Treball final de grau

Desenvolupament de codis d'algebra lineal amb PyCOMPSs

Tutora del projecte

Rosa M. BADIA SALA

Arquitectura de Computadors

Estudiant

Ramon AMELA MILIAN

Grau en Enginyeria Informàtica

Especialitat de Computació

17 de gener de 2017

Resum

Arrel de l'interès per ser capaç d'utilitzar les llibreries de *NumPy* amb *MKL* en entorns distribuïts, va sorgir una col·laboració entre *Intel*® i el departament de *Workflows and distributed computing* del *BSC*.
Aquest treball explora un seguit de possibles línies d'investigació que tenen per objectiu millorar el rendiment i facilitar la utilització simultània de *MKL* a través de *NumPy* i *PyCOMPSs* per a realitzar operacions d'àlgebra lineal en entorns distribuïts.
Pel que fa als algoritmes matemàtics utilitzats, en una primera instància es busquen solucions per aconseguir un rendiment màxim de l'algoritme de multiplicació de matrius mitjançant la descomposició de les mateixes en blocs quadrats. A continuació, s'ha adaptat el codi ja present al *BSC* per calcular la factorització de *Cholesky* amb inicialització distribuïda i augmentar-ne el paral·lelisme. L'última tasca realitzada en aquest apartat consisteix en la implementació d'un algoritme per al càlcul de la descomposició *QR* mitjançant la descomposició en matrius quadrades i inicialització distribuïda.
Tenint en compte la importància de la planificació per obtenir un bon rendiment de les aplicacions, a continuació s'ha procedit a fer una refactorització del planificador per introduir-hi un conjunt de noves polítiques. Concretament, s'han afegit planificadors que segueixen una política *FIFO*, una *LIFO* i una *FIFO* modificada per prioritzar la localitat de les dades i minimitzar així la quantitat de transferències.
Finalment, s'ha dissenyat i implementat una llibreria que utilitza un sistema de *wrapping* sobre la llibreria *NumPy* que permet la introducció progressiva dels algoritmes distribuïts sense la necessitat d'implementar-la tota de cop. En tot moment es garanteix, però, que l'usuari podrà accedir a totes les funcionalitats de la llibreria original.

Abstract

In the wake of the interest in being capable to use *MKL* through *NumPy* in distributed systems, a collaboration between *Intel*® and the *BSC*'s *Workflows and distributed computing* department has been created.
This work explores several possible investigation lines that aims to improve the behaviour and ease the use of *NumPy* with *MKL* and *PyCOMPSs* simultaneously to carry out linear algebra operations in distributed systems.
Regarding the used algorithms, first of all some solutions are explored to achieve a better performance in the blocked matrix multiplication. Next, the code already present in the center that computes the Cholesky factorisation has been modified in order to improve his parallelisation level and initialise the matrix in a distributed way. Finally, an algorithm to compute a *QR* decomposition through the matrix decomposition in smaller square matrix initialized in a distributed way has been implemented.
Next, a refactor in the *COMPSs* scheduler has been done in order to ease the creation of new scheduling policies. Once this work has been done, some new schemes has been added. In particular, schedulers with *FIFO*, *LIFO* and *FIFO* modified to schedule child tasks in the father's worker to minimize the amount of data transfers has been added.
Finally, a library that wraps the entire *NumPy* library has been implemented. This new environment allows the team to introduce the distributed implementations progressively assuring that importing it the user will have access to the full stack of *NumPy*'s functionalities, even those not still implemented.

Resumen

A raíz del interés por ser capaces de utilitzar las librerias de *NumPy* con *MKL* en entornos distribuidos, surgió una colaboración entre *Intel*® i el departamento de *Workflows and distributed computing* del *BSC*.
Este trabajo explora un conjunto de posibles lineas de investigación que tienen por objetivo mejorar el rendimiento y facilitar la utilización simultánea de *MKL* a través de *NumPy* i *PyCOMPSs* para realizar operaciones de álgebra lineal en entornos distribuidos.
En lo que afecta a los algoritmos matemáticos utilizados, en una primera instancia se buscan soluciones para conseguir un rendimiento máximo de la multiplicación de matrices mediante la descomposicion de las mismas en bloques cuadrados. A continuación, se ha adaptado el código ya existente en el *BSC* para calcular la factorización de *Cholesky* con la inicialización distribuida i aumentar su grado de paralelismo. La última tarea realizada consiste en la implementación de un algoritmo para el cálculo de la descomposición *QR* mediante la descomposición de la matriz en bloques cuadrados realizando la inicialización de forma distribuida.
Considerando el impacto de la planificación para obtener un buen rendimiento de las aplicaciones, a continuación se ha procedido a realizar una refactorización del planificador con el objetivo de introducir nuevas políticas. Concretamente, se han añadido planificadores que siguen una política *FIFO*, una *LIFO* i una *FIFO* modificada para priorizar la localidad de los datos y evitar así el número de transferencias.
Finalmente, se ha diseñado e implementado una librería que utiliza un sistema de *wrapping* sobre la librería *NumPy* que permite la introducción progresiva de los algoritmos distribuidos sin la necesidad de implementar-la toda de golpe a la vez que el usuario sigue teniendo acceso a todas las funcionalidades de la librería original.

Agraïments

En primer lloc, voldria agrair a la Dra. Rosa M. Badia Sala tant la confiança depositada en mi per a la realització d'aquest projecte com els seus inestimables consells i suggeriments.
M'agradaria també agrair a Adrià Àguila, Pol Álvarez, Javier Conejero, Sandra Corella, Jorge Ejarque, Daniele Lezzi, Francesc Lordan, Cristian Ramon-Cortés, Albert Serven i Raül Sirvent, membres del grup de *Workflows and Distributed Computing* del *Barcelona Supercomputing Center* per la seva ajuda durant la realització del projecte.
Finalment voldria agrair tant als meus pares, Ramon i Maria, com al meu germà, Josep, el suport incondicional al llarg de tot el viatge que m'han portat a la realització d'aquest projecte final de carrera.

Índex

Índex de figures

Índex de taules

Glossari

ComputingUnits en aquest context, unitats de còmput del worker assignades a una determinada tasca. 24

GFlops milions d'operacions de coma flotant per segon. 15

graf de dependències graf unidireccional acíclic on cada node representa una tasca i cada aresta representa l'existència d'una dependència de dades entre les dues tasques relacionades. 10

internode referent a una execució que comprèn diversos nodes. 9

intranode referent a una execució dintre d'un sol node. 9

lazy evaluation mecanisme gràcies al qual una dada sol es calcula quan es necessita i que permet seguir el flux de còmput tot i que alguna crida a funció no hagi acabat. 9

llenguatge funcionals llenguatge que tracta la programació com l'avaluació de funcions matemàtiques evitant els canvis d'estat i l'ús dels objectes mutables. 9

master node encarregat d'orquestrar l'execució del programa planificant l'execució de les tasques als diferents workers disponibles així com d'organitzar-ne les transferències. 18

mida de bloc en aquest document, nombre de reals per fila/columna dintre de cada bloc. 27

mida de matriu en aquest document, nombre de blocs per fila/columna. 27

MKLProc en aquest context, quantitat de threads creats per una determinada crida *MKL* per a realitzar el còmput corresponent. 24

node en un sistema distribuït, cadascun dels elements individuals vists per la xarxa. 13

overhead temps de càlcul afegit durant el procès de paral·lelització d'una determinada execució. 26

oversubscriving creació d'un nombre de threads superior al nombre de CPU's presents a una determinada màquina. 35

paral·lelització tècnica que consisteix en segmentar una execució amb la finalitat d'aprofitar tots els recursos disponibles. 9

potència en aquest context, es considera que la potència es el nombre d'operacions en coma flotant per segon que pot realitzar un determinat equip. 8

release versió pública d'un programa. 9

sistema distribuït conjunt d'equips que disposen d'una connexió de xarxa que els permet comunicar-se per a realitzar una determinada acció de forma coordinada. 8

speed up en aquest document, temps d'execució amb N workers / temps d'execució amb 1 worker. 28

strong scaling augment del nombre de recursos sense variar la mida del problema. 15

tasca unitat mínima de càlcul en la que es descompon el programa a paral·lelitzar. 13

thread unitat mínima d'execució que pot gestionar el planificador d'un sistema operatiu. 54

weak scaling augment del nombre de recursos variant proporcionalment la mida del problema. 15

worker node encarregat de realitzar l'execució de tasques. 18

zoom sincronitzat referent a dues vistes d'una mateixa traça, ampliació que engloba el mateix període de temps amb la mateixa escala. 30

Introducció

Índex

A l'hora de dissenyar un ordinador, es pot augmentar la potència del mateix augmentant la capacitat de càlcul de cadascun dels seus processadors o augmentant el número d'unitats de còmput. Històricament, l'augment de la potència dels equips s'ha realitzat mitjançant la primera opció. L'ús de múltiples processadors per executar una mateixa aplicació es reservava, doncs, a aplicacions d'alt rendiment que tot sovint han estat lligades a la recerca i el desenvolupament.

La llei de Moore prediu que, cada dos anys, el nombre de transistors dintre d'un processador es duplica, fent possible que, en general, la capacitat de còmput dels ordinadors es dupliqui cada dos anys. No obstant, darrerament s'ha comprovat que ja no es podia augmentar el nombre de transistors dintre de cadascun dels dispositius. Per continuar augmentant la capacitat de càlcul, s'ha procedit a la incorporació múltiples processadors dins d'un mateix equip. Aquest fenomen es pot apreciar a la figura 1.

A més a més, la popularització de l'Internet i l'augment de la seva velocitat ha fet créixer l'interès en els sistemes distribuïts, que permeten realitzar un cert còmput en màquines deslocalitzades geogràficament.

En aquest projecte es presenta una problemàtica relacionada amb aquesta tema. L'objectiu del mateix es trobar una estratègia per a resoldre-la i, posteriorment, dur-la a terme.

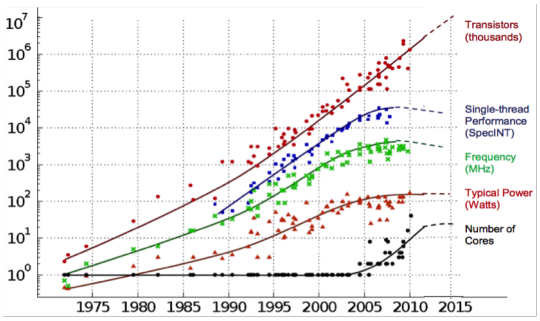

Figura 1 – Evolució de les característiques dels ordinadors (1970-2015) [1]

0.1 Estat de l'art

Dintre del món de la paral·lelització, es pot realitzar, d'entrada, una separació molt clara entre paral·lelització intranode i internode. En l'actualitat ja hi ha múltiples estàndards que regeixen ja sigui les comunicacions internode (MPI) [2] com la paral·lelització d'aplicacions intranode (OpenMP)[3].
Aquests dos estàndards són el referent per a tots els centres de programació d'alt rendiment del món en el que a paral·lelització de programes que utilitzen llenguatges imperatius es refereix. Els llenguatges funcionals com el *Javascript* [4] no necessiten aquest tipus d'eina gràcies al fet que siguin paral·lels de forma implícita gràcies a la lazy evaluation.
Tot i que els usuaris especialitzats coneixen perfectament tant *MPI* com *OpenMP*, hi ha llibres i cursos sencers dedicats a la seva completa coneixença donada la seva complexitat [5] [6]. Aquest fet implica que per a implementar un programa sobre una determinada disciplina que s'executi en paral·lel cal, en el cas general, un ampli coneixement sobre la temàtica en qüestió i sobre els estàndards de paral·lelització.
Cal remarcar el fet que fins fa poc la programació intensiva en sistemes distribuïts estava reservada a un nombre molt limitat d'empreses privades i centres de recerca. La manca d'interès general, feia que cada actor implementés la seva solució particular. De fet, algunes de les solucions trobades per aquests actors privats han esdevingut estàndards a la llarga i han donat peu a projectes de programari lliure. En són dos exemples *Apache Hadoop* [7], que va llençar la seva primera release estable el 2011[8] i basat en els papers publicats per *Google* sobre l'execució de l'esquema *Map-Reduce* en sistemes distribuïts i *Apache Spark* [9], llençat el 2014 i resultat de treballs d'investigació realitzats a la Universitat de Califòrnia, Berkeley, sobre sistemes distribuïts amb memòria distribuïda.
Com es pot veure, es tracta d'una àrea molt nova que ha despertat molt interès en els darrers anys però sobre el que hi ha poca bibliografia i molt treball per fer ja que es troba en els seus estadis inicials.

0.2 Contextualització

Clàssicament la programació d'alt rendiment ha estat utilitzada des de les previsions meteorològiques al càlcul d'interaccions entre proteïnes per a la indústria farmacèutica però sempre en centres d'investigació o per un nombre molt reduït d'empreses privades. A mesura que creixien les potencials aplicacions d'aquestes tecnologies va anar creixent en el nostre territori un interès per aquesta temàtica, naixent el 2004 el BSC-CNS (Barcelona Supercomputing Center - Centro Nacional de Supercomputación).

BSC-CNS

Aquesta institució té com a finalitat la investigació, desenvolupament i gestió de les tecnologies de la informació per tal de facilitar l'avenç científic. Amb aquest objectiu, es realitzen treballs en els camps de les ciències de la computació, ciències de la terra, ciències de la vida i enginyeria computacional al voltant del superordinador més potent de l'estat, *MareNostrum*.
Per tal d'aprofitar al màxim la capacitat de les instal·lacions, un enfoc pluridisciplinar es indispensable. Com s'ha explicat a l'estat de l'art, la comunitat informàtica ha desenvolupat dos estàndards per a la paral·lelització de processos seqüencials. El problema principal d'aquests estàndards és la seva alta complexitat. Per tant, sol els usuaris experts són capaços d'aconseguir executar els seus programes en entorns distribuïts. Una vegada detectada aquesta problemàtica, el BSC ha desenvolupat dos models de programació anomenats *OmpSs* i *COMPSs* per facilitar la paral·lelització intranode i internode de programes als usuaris d'altres disciplines.

COMPSs

COMPSs (COMP Superscalar) es un model de programació per permet la paral·lelització internode de programes seqüencials en sistemes distribuïts mitjançant la definició de tasques. Una vegada aquestes han estat definides, COMPSs s'encarrega de gestionar tot el graf de dependències i les transferències entre els diferents equips.
Aquest model ha estat programat i actualment mantingut pel grup *Workflows and Distributed Computing*[10] del departament *Computer Science*. Amb aquesta eina, els departaments ciències de la terra, ciències de la vida i aplicacions informàtiques són capaços d'executar les seves aplicacions en *Java*, *Python* (*PyCOMPSs*) i *C++* en paral·lel amb un esforç molt baix.
Una vegada realitzada la paral·lelització internode, caldrà triar quin mecanisme s'utilitza per a tractar la paral·lelització intranode. En aquest punt es podria, per exemple, utilitzar *OmpSs*[11]. No obstant, hi ha certes llibreries disponibles al mercat que ja incorporen una paral·lelització intranode sub-

jacent. Aquestes, en molts casos, són més òptimes que si s'utilitza *OmpSs* o *OpenMP* ja que han estat optimitzades per a les particularitats de cada aplicació. És el cas de MKL.

MKL

MKL (Math Kernel Library [12]) és una llibreria C/C++/Fortran. Degut a la popularitat d'aquesta llibreria, *Intel*® comercialitza la seva propia versió amb una optimització multiprocessador de rutines matemàtiques basada en *OpenMP*[13]. La optimització ha estat feta per als processadors de la mateixa marca. Per tant, el rendiment esperat si s'utilitza el seu maquinari és molt alt. Aquesta llibreria permet realitzar càlculs d'àlgebra lineal, transformades de Fourier, anàlisi vectorial, estadística i tractament de dades.
Considerant que Intel® és el líder indiscutible en el mercat de la fabricació de processadors per ordinadors [14] (Samsung fabrica sobretot processadors per aparells mòbils), l'ús d'aquesta llibreria es fa quasi imprescindible a l'hora de paral·lelitzar càlculs de tipologia matemàtica. Aquest és el motiu principal pel qual *NumPy*[15], llibreria de càlcul numèric de *Python*, incorpora MKL [16].
Cal remarcar que l'esforç de la comunitat per optimitzar les rutines científiques de *Python* ha provocat que aquest llenguatge s'utilitzi cada vegada més com alternativa gratuïta a *Matlab*.
D'aquesta manera, aquesta llibreria és utilitzada des de laboratoris de recerca científica fins a empreses on el càlcul numèric té una especial importància. En són un exemple les d'inversió financera o anàlisi de riscos.

Actors

L'objectiu principal d'aquest projecte és explotar la potència de *PyCOMPSs* per a realitzar la paral·lelització internode i *MKL* per a la intranode per implementar càlculs d'algebra lineal utilitzant *Python*. D'aquesta manera es busca que usuaris no experts en paral·lelisme siguin capaços de programar aplicacions que continguin càlculs d'àlgebra lineal que seràn executats en sistemes distribuïts.
En un primer terme, aquests càlculs seran aprofitats pel *BSC-CNS*. No obstant, el fet que aquesta disciplina sigui tant utilitzada en nombroses aplicacions, fa pensar que el salt a la seva utilització per part de les empreses privades no tardarà en donar-se. Concretament, l'àlgebra lineal s'utilitza en àmbits tan diversos com la optimització de rutes o el càlcul d'estructures.

0.3 Organització del document

Per tal de facilitar la comprensió del treball realitzat, s'ha intentat que el document segueixi una estructura clara i lògica. Concretament, aquest disposa de les següents parts:

- Formulació del problema
  A l'inici es fa una descripció de la problemàtica abordada, descrivint tant l'estat actual com els objectius del projecte i la metodologia utilitzada.

- Planificació
  Una vegada presentats els objectius del projecte, es fa una pinzellada sobre les seves característiques temporals, pressupostaries i de sostenibilitat envers la societat.

- Millora del rendiment de la multiplicació de matrius
  La primera feina realitzada a estat l'anàlisi de l'execució de l'algoritme de multiplicació de matrius per blocs. En aquest capítol es presenten els resultats obtinguts així com les decisions preses per tal de millorar-ne el comportament al màxim.

- Implementació de nous algoritmes
  Amb els resultats de l'anàlisi de la multiplicació de matrius, s'ha procedit a la modificació de l'algoritme de factorització de *Cholesky* i la implementació de la factorització QR.

- Implementació de nous planificadors
  En els capítols anteriors s'ha constatat la importància d'una bona planificació per a l'obtenció d'un bon rendiment. En aquest capítol es presenten les solucions adoptades per dotar l'usuari de més polítiques de planificació.

- Implementació de la llibreria matemàtica distribuïda
  Després de constatar la dificultat a l'hora de programar algoritmes distribuïts eficients, s'han estudiat les possibles alternatives per a facilitar el treball d'un usuari no expert.

- Conclusions
  Finalment s'adjunta un resum dels resultats obtinguts així com una sèrie de propostes de treball futur.

Capítol 1

Formulació del problema

Índex

1.1 Descripció del problema

Tenint en compte que, a priori, *MKL* realitza la millor paral·lelització intra-node possible en arquitectures Intel®, els desenvolupadors de *COMPSs* van decidir comprovar si aquest entorn de treball és completament compatible amb *MKL* i aprofita totes els seves funcionalitats a través de la citada llibreria *NumPy*.
Els resultats obtinguts fan pensar que hi pot haver problemes a l'hora d'assignar els recursos d'un node de forma que les diferents instàncies *MKL* que s'hi executen ho fan sol en els primers N processadors assignats. Així, si a un node de 16 processadors s'assignen M tasques on cada tasca disposa de N/M processadors, enlloc de repartir-se els N processadors i aprofitar tota la capacitat del node, les M tasques s'executen en els N processadors amb un identificador més baix.
El projecte té com a finalitat principal la confirmació d'aquesta hipòtesi, la seva solució i l'obertura d'altres línies d'investigació que permetin augmentar l'eficiència de càlcul actual i la facilitat d'utilització de la combinació *PyCOMPSs/MKL*.

1.2 Abast del projecte

El projecte es pot dividir en cicles que continguin els següents elements:

1. Detecció del problema
   En aquesta fase cal llançar una bateria completa de testos per detectar les anomalies en l'execució que fan que aquesta no tingui el comportament desitjat o, en el cas d'afegir noves funcionalitats, definir correctament tots els requeriments del que es desitja implementar.

2. Solució
   En aquesta fase cal trobar una solució al problema detectat. Aquesta solució pot ser de diferents tipologies:

   - Modificació del codi de *COMPSs* per aconseguir aprofitar tots els recursos d'un node quan diferents instàncies de *MKL* s'hi estan executant.
   - Implementació de noves funcionalitats dintre de *COMPSs* per tal d'obtenir un millor rendiment.
   - Implementació de noves funcionalitats dintre de *PyCOMPSs* de forma que el seu ús sigui més intuïtiu per a un desenvolupador no expert.

3. Resultats
   Aquesta fase consisteix en la repetició dels testos executats a la primera fase per comprovar que les anormalitats trobades han desaparegut i els resultats obtinguts són els esperats. En el cas de funcionalitats noves, caldrà assegurar la robustesa i correctesa de les mateixes.

En aquest projecte es tracten, doncs, totes les fases d'un procés de recerca. Des de la detecció del problema fins l'avaluació dels resultats que permet confirmar que aquest problema ha estat resolt.
A més a més, es concep el mateix en cicles curts per tal d'anar millorant la solució gradualment. D'aquesta manera, es pot revisar la planificació periòdicament per tal d'adaptar-la als resultats obtinguts fins al moment. Finalment, es fa possible abordar diferents problemàtiques sense comprometre l'acabament del projecte.

1.3 Metodologia utilitzada

Com s'ha pogut comprovar, els anàlisis de rendiment tenen un gran pes en aquest projecte. Aquests indiquen si els programes es comporten com un podria esperar o si, per contra, existeixen certes anormalitats.
Per tal d'avaluar les solucions implementades, s'utilitzaran testos del tipus strong scaling en funció de les unitats de còmput assignades. Segons la llei d'Amdahl[17], en una execució ideal el temps d'execució és inversament proporcional al nombre d'unitats de còmput assignades. No obstant, hi ha una

certa sobrecàrrega (temps perdut en tasques de transmissió de dades i sincronització entre els diferents nodes i processadors) que és impossible evitar. Per tant, aquest creixement mai serà perfectament lineal.
L'algoritme utilitzat com referència serà la multiplicació de matrius per blocs. S'ha pres aquesta decisió ja que és un algoritme molt conegut per la comunitat científica i àmpliament utilitzat en la valoració de l'eficiència. A més a més, per confirmar els resultats s'utilitzaran altres algoritmes com la factorització de *Cholesky* i la descomposició *QR* [18].
Com a suport als testos de tipus *strong scaling*, s'utilitzaran les traces (informació del treball realitzat per cada component al llarg de l'execució) obtingudes amb el paquet *Extrae* proporcionat pel mateix *BSC*.
En aquest punt cal afegir que degut a limitacions de memòria, els anàlisis de tipus *strong scaling* no permeten obtenir programes que siguin prou grans com per escalar amb molts nodes i alhora siguin prou petits per poder-se executar en un sol node sense deixar al node sense memòria. Aquest fet ha provocat que també s'incloguin els anàlisis de tipus weak scaling. Cal remarcar que, en el cas ideal, el temps d'execució en aquest tipus de test es manté constant.
Tots els testos seran executats al superordinador *MareNostrum* per garantir un entorn aïllat i estable que doni credibilitat als resultats obtinguts.
Com a mètode de validació es compararà el rendiment i la facilitat d'ús de l'aplicació en el seu estat abans del projecte i el final obtingut.
Pel que fa a les eines de seguiment, es realitza una reunió amb tot el grup cada divendres a més a més de reunions periòdiques cada 1-2 setmanes amb la tutora del projecte que permetin un intercanvi d'idees i una validació dels avenços obtinguts. Per a la gestió de les iteracions, s'utilitza la plataforma *Trello*, que permet definir objectius així com tasques pendents i en curs.
Per a la comunicació del dia a dia entre els components de l'equip, s'utilitza el correu electrònic, *Skype* i *Slack*.
Pel que fa a l'intercanvi de codi i el control de versions, s'utilitza *Subversion*.

1.4 Entorn d'execució

Tots els resultats presentats en aquest document han estat obtinguts a *MareNostrum* utilitzant matrius emplenades amb nombres de tipus real. D'aquesta forma, s'obté el rendiment de les aplicacions en `GFlops`. Els nodes utilitzats disposen de dos processadors Intel® Xeon® E5-2670 amb una capacitat total de càlcul de 332 GFlops/s/node [19]. Pel que fa a la memòria, la majoria de nodes disposen de 32GB de RAM, tot i que hi ha nodes més grans amb 64GB i 128GB respectivament. Si no s'indica el contrari, en totes les execucions considerades s'utilitzen nodes amb 32GB de memòria.

Capítol 2

Planificació

Índex

2.1 Planificació temporal

Tenint en compte la metodologia utilitzada, a l'inici del projecte es va realitzar una planificació. En aquesta secció es presenten les diferències més significatives respecte a la citada planificació inicial.

Planificació inicial

La durada aproximada del projecte és de 496 hores. A la taula 1 es pot veure un resum de la durada de cadascuna de les tasques. Aquest desgloçament

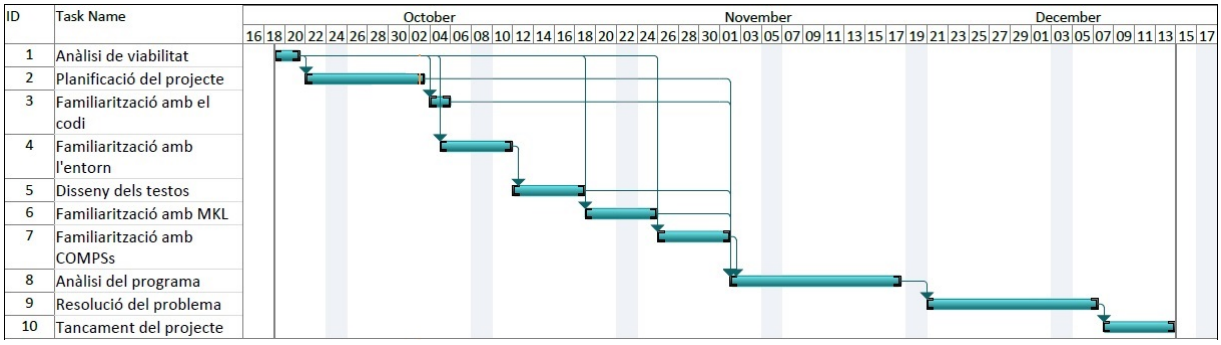

Figura 2 – Diagrama de Gantt del projecte

dóna com resultat el diagrama de Gantt presentat a la figura 2. En aquest diagrama es pot apreciar una distribució temporal de les tasques així com una representació de les dependències existents entre les mateixes.

Tasca	Estimació de la durada (hores)	Rol
Anàlisi de viabilitat	20	Cap de projecte
Planificació del projecte	60	Cap de projecte
Familiarització amb el codi	16	Programador
Familiarització amb l'entorn	20	Programador
Disseny dels testos	20	Programador
Familiarització amb MKL	20	Programador
Familiarització amb COMPSs	20	Programador
Millora de l'aplicació actual	280	Programador
Tancament del projecte	40	Programador
Total	496	

Taula 1 – Taula resum de la planificació temporal

Planificació seguida

No hi ha cap diferència significativa entre la planificació inicial i la planificació seguida. Aquest fet es deu a la llibertat donada en la resolució del problema. El fet de realitzar cicles curts d'anàlisi i resolució a fet possible anar millorant paulatinament els resultats obtinguts sense tenir que fer modificacions majors ni posar en risc la correcta finalització del projecte.
Cal esmentar en aquest punt que tot i que la planificació no ha patit cap modificació, si que ho han fet les tasques previstes per a realitzar. En un primer moment es va plantejar la possibilitat de valorar el correcte funcionament de les solucions implementades mitjançant la multiplicació de matrius i la descomposició de *Cholesky*. Cal esmentar que, a més a més, s'ha afegit una implementació de la descomposició *QR*.

A més a més, degut als resultats observats, s'han realitzat modificacions en el codi de *COMPSs* per tal d'obtenir un rendiment superior. Aquestes modificacions afecten la crida utilitzada per llençar les tasques als workers i a la planificació realitzada per part del master.
Finalment, s'ha programat una llibreria per a connectar *numpy* i *PyCOMPSs* de forma totalment transparent per a l'usuari.

2.2 Pressupost

Una vegada definit el temps necessari per a desenvolupar totes les tasques relacionades amb el projecte, el següent pas consisteix en fer una valoració del seu cost econòmic.
A part de l'ordinador necessari per a realitzar el treball i l'adquisició del superordinador *Mare Nostrum*, totes les altres despeses estan lligades al nombre d'hores treballades. Aquest fet es degut a la utilització de programari lliure al llarg de tot el projecte.
Per tant, en aquest cas el correcte disseny del diagrama de Gantt que permeti un càlcul acurat del nombre d'hores necessàries per al desenvolupament del projecte tindrà un alt impacte en el cost final del mateix.

Recursos humans

S'ha considerat que per a dur a terme aquest projecte caldrà un total de 496 hores. D'aquestes hores, 416 corresponen a un sou de programador mentre que 80 són realitzades per un cap de projectes.

En aquest punt, s'ha considerat que el cost d'un enginyer informàtic amb les capacitats suficients per al correcte desenvolupament de les tasques programades és d'aproximadament 15 €/hora per a l'empresa. El cost del cap de projectes serà de 35 €/hora [20]. Cal notar que aquest sou no és el que percebrà finalment el treballador. En el mateix s'inclouen tots els impostos i cotitzacions imposades pel marc legal actual.
Donat el preu de l'hora i el nombre d'hores necessàries, es pot arribar a la conclusió que el cost en recursos humans d'aquest projecte és d'aproximadament $416 \cdot 15 + 80 \cdot 35 = 9040$ €.

Recursos *hardware*

En el càlcul del cost dels equips utilitzats, s'ha tingut en compte que *Mare-Nostrum* és de propietat pública. Concretament, el grup de *Workflows and distributed computing* té un cert nombre d'hores assignades per a l'execució de còmputs de forma gratuïta pel fet de realitzar tasques de recerca dins del

BSC. El cost que s'ha de pagar per l'accés a la màquina serà de 0 €. Considerant l'exposat anteriorment, els costos d'amortització del mateix són de 0 €.
Cal considerar en aquest punt que en el cas de que un usuari extern desitgi accedir a les instal·lacions, les tarifes són de 0.8-1 €/hora/CPU.
Posteriorment, s'han considerat els costos d'amortització de l'ordinador portàtil [21]. El temps d'ús dels mateixos, pot oscilar entre 3 i 5 anys. Per tant, s'ha decidit de prendre un temps de servei intermig.
El resultat d'aquest estudi es pot veure a la taula 2.

Concepte	Preu	Vida útil	Amortització
MareNostrum	22700000 €	4 anys	0 €
Ordinador portàtil	900 €	4 anys	56,25 €

Taula 2 – Taula resum del *hardware* necessari per a desenvolupar el projecte

Recursos *software*

Com ja s'ha dit anteriorment, tot el programari utilitzat és gratuït. Per tant, en aquest projecte no hi ha costos associats a aquest apartat.

Pressupost total

Tenint en compte els comentaris fets en les seccions anteriors, es pot concloure que el cost econòmic d'aquest projecte és de 9096.25 €. A aquest valor caldrà sumar un 21% d'IVA, obtenint un import de contracte de 11.006,46 €.

2.3 Informe de sostenibilitat

En aquest projecte s'ha tingut molta cura de que aquest sigui sostenible des del punt de vista ambiental, social i econòmic. A la taula 3 es mostra una taula resum obtinguda a partir de tots els raonaments fets dintre d'aquest capítol.

Sostenibilitat ambiental

Aquest projecte té per objectiu l'ús eficient dels recursos disponibles per a realitzar un determinat càlcul. Una de les conseqüències del mateix serà, doncs, la capacitat de realitzar un càlcul amb una despesa de recursos inferior. Si es

	PPP	Vida útil	Riscos	Total
Ambiental	8	15	0	23
Econòmic	5	10	-10	5
Social	7	15	-5	17
Total	20	40	-15	45

Taula 3 – Taula de sostenibilitat

considera que en una situació qualsevol ja s'hauran comprat els equips, aquest estalvi afecta sobretot al consum elèctric.
Cal veure aquí que no hi ha cap factor que pugui fer que el projecte augmenti la seva petjada durant la realització del mateix.

Sostenibilitat social

Com s'ha comentat a la contextualització, aquest projecte té moltes aplicacions directes a la vida real. L'augment del rendiment a l'hora de realitzar certs càlculs pot implicar, per exemple, que certes optimitzacions que fins ara no es podien fer passin a ser viables. Un exemple en podria ser la optimització de la distribució del tràfic d'internet a través de la xarxa en temps real. El fet que aquest càlcul hagi de fer-se al moment, implica que el temps de còmput sigui crític. La millora d'aquest temps de còmput pot fer que, sense necessitat de canviar la infraestructura física, els usuaris vegin com la seva connexió internet augmenta de velocitat.

Sostenibilitat econòmica

Com s'ha raonat a la secció anterior, tot el programari utilitzat és de tipus gratuït. De fet, els costos es redueixen a la quantitat d'hores treballades i a l'equip informàtic bàsic per a desenvolupar el projecte. Tenint aquest fet en compte, l'única font de sobrecostos que podria aparèixer és la pròpia derivada d'un augment de les hores necessàries per a realitzar el treball. Tot i haver-se considerat períodes de temps que es consideren amplis per a realitzar cadascuna de les tasques, aquest apartat resta una incògnita. El fet de tractar-se d'un treball de recerca no garanteix la correcta finalització del projecte dins del terminis. No obstant, cal notar que es pot donar el cas en el que les tasques descrites es realitzin en un temps inferior a l'especificat.
Donat el context del projecte i les múltiples repercussions que pot tenir, el cost del projecte es molt baix en comparació als avantatges que es poden obtenir. No obstant, com tot projecte de recerca, té el risc de no trobar resultats concloents. En aquest cas, tota la inversió pot considerar-se inútil des d'un

punt de vista empresarial. Des d'un punt de vista del coneixement, però, pot aportar pistes per a la tria de pròximes línies de recerca.

Capítol 3

Millora del rendiment de la multiplicació de matrius

Índex

El primer problema que s'ha abordat en aquest projecte es la millora del rendiment de l'algoritme disponible de multiplicació de matrius. Aquest considera matrius quadrades de dimensions idèntiques a l'entrada. La sortida estarà descomposta el mateix nombre de blocs que les matrius d'entrada. El resultat de multiplicar dues matrius A i B es calcula com es mostra a l'equació 3.1. Cal tenir en compte que l'acumulació es realitza utilitzant la comanda `C[i][j] += A[i][k] * B[k][j]`, creant de forma implícita una dependència de dades entre dos sumands consecutius.

$$C_{i,j} = \sum_k A_{i,k} \cdot B_{k,j} \tag{3.1}$$

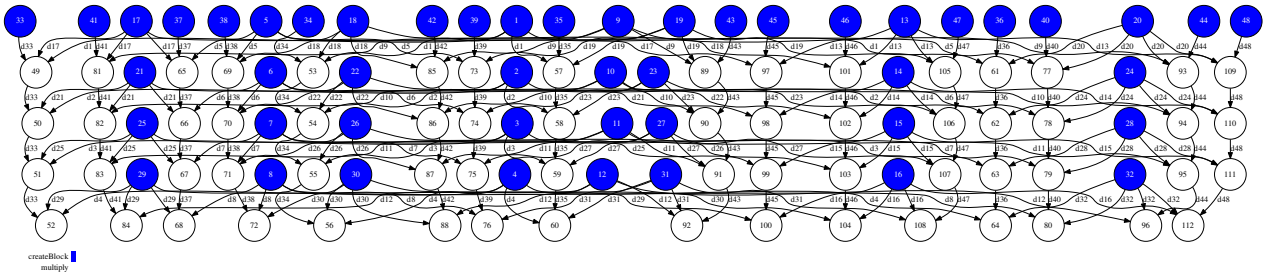

Figura 3 – Graf de dependències corresponent a la multiplicació de matrius

Les consideracions anteriors aplicades a la multiplicació de dues matrius de mida 4 donen com resultat el graf de dependències presentat a la figura 3. Els nodes blaus representen les inicialitzacions de les matrius mentre que els nodes blancs representen la multiplicació de dos blocs.

3.1 Introducció

A l'hora d'interpretar les traces obtingudes, cal tenir en compte que actualment *Paraver* només es capaç de capturar els esdeveniments generats pel thread principal on es troba el codi de la tasca. Entendre bé aquest fenomen és capital per tal d'analitzar correctament les traces de les execucions que utilitzen *MKL*.
A l'apèndix A es realitza una breu descripció del funcionament d'aquest programa i les vistes utilitzades a continuació. Es recomana la lectura del nombrat apèndix en aquest punt a tots els lectors no familiaritzats amb *Paraver* i les vistes utilitzades a continuació.

Format de les crides

Una tasca tipus tindrà l'estructura mostrada a la figura 4. A continuació es detalla el significat de cadascuna de les línies:

1. @constraint (ComputingUnits="$ComputingUnits")
   Aquesta línia indica el nombre d'unitats de còmput del worker que *PyCOMPSs* li assignarà a cadascuna de les tasques d'aquest tipus.

2. @task(returns=list)
   Aquesta línia marca la funció que es trobi a continuació com una tasca.

3. def foo(*args, **kwargs, MKLProc):
   Capçalera de la funció. El nombre de threads que crearà la funció *MKL* invocada en les línies posteriors.

```
1 @constraint (ComputingUnits="${ComputingUnits}")
2 @task(returns=list)
3 def foo(*args, **kwargs, MKLProc):
4   os.environ["MKL_NUM_THREADS"]=str(MKLProc)
5   -- instruccions que inclouen una crida a la llibreria MKL
6 return ret
```

Figura 4 – Exemple de tasca MKL

4. os.environ["MKL_NUM_THREADS"]=str(MKLProc)
   Com s'ha dit anteriorment, *MKL* utilitza *OpenMP* per a realitzar la paral·lelització intranode. La forma més típica de modificar el comportament dels programes que utilitzen aquest model de programació és mitjançant l'ús de variables d'entorn. Cal notar en aquest punt que la crida `os.environ["OMP\_NUM\_THREADS"]=str(MKLProc)` tindria exactament el mateix efecte que la instrucció utilitzada. S'ha decidit utilitzar `MKL\_NUM\_THREADS` ja que es considera que d'aquesta manera el codi en aquest context s'entén millor.

5. – instruccions que inclouen una crida a la llibreria MKL
   En aquest bloc d'instruccions es realitza la crida a la llibreria *MKL* que crearà *MKLProc* threads que realitzaran el còmput en paral·lel.

6. return ret
   Aquesta línia és opcional i sol apareixerà en els casos en els que la funció retorni algun valor de forma explícita. És important notar que es possible retornar els paràmetres mitjançant variables d'entrada-sortida. En aquest cas, la línia 6 no existirà però la línia 2 serà del tipus `@task(c=INOUT)`, on `c` és el paràmetre d'entrada-sortida.

Graus de paral·lelisme

Com s'ha vist en l'explicació anterior, hi ha dos paràmetres que marcaran el nombre de threads que, en cada moment, s'executaran en cada worker. Es tracta de `ComputingUnits` i `MKLProc`.
Paraver només és capaç de capturar els esdeveniments que s'emeten des del codi de *Python*. Aquest fet implica que a les traces no hi haurà diferència aparent entre dos execucions amb tots els paràmetres iguals excepte `MKLProc`, ja que en cap moment s'emeten esdeveniments des dels threads creats dins de les rutines *MKL*. Tenint en compte que l'únic indicador per extreure conclusions respecte aquest paràmetre és el temps d'execució, han calgut una bateria d'experiments molt àmplia per tal d'arribar a conclusions interessants.
Pel que fa a les `ComputingUnits`, les figures 5, 6 i 7 mostren les traces resultants d'execucions amb el paràmetre `ComputingUnits` de les tasques igual a

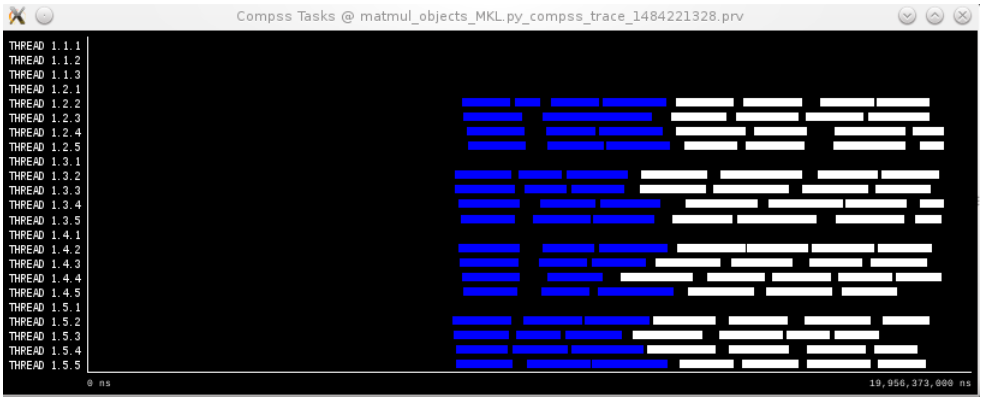

Figura 5 – Vista Compss Tasks amb `ComputingUnits = 1`

1, 2 i 4 respectivament. Aparentment, el grau de paral·lelisme és molt diferent entre elles. Emperò, aquesta conclusió és precipitada. Cal tenir en compte que cadascuna de les crides *MKL* crearà `MKLProc` threads. Que segons les traces una unitat de còmput estigui inactiva no implica necessàriament que aquest fet sigui veritat com es mostrarà en les següents seccions. Concretament, el nombre de threads de càlcul teòricament presents en un worker es pot calcular mitjançant la formula 3.2.

$$N = \frac{UC}{comp} MKLProc$$

on :

N = total de threads de còmput presents al worker
$comp$ = `ComputingUnits`
UC = unitats de càlcul al worker
$MKLProc$ = nombre de threads creats per cada crida *MKL* (3.2)

En els següents exemples s'utilitza un sol worker amb 4 unitats de còmput. En un extrem es troba la figura 5, on cada tasca en requereix 1. Es poden executar, doncs, quatre tasques alhora. D'altra banda, la figura 7 mostra una execució on cada tasca requereix 4 unitats de còmput. El nombre màxim de tasques que es poden executar en cada instant és de 1.
Finalment, cal tenir en compte que els threads creats per *MKL* sol es trobaran actiu durant la part taronja de la vista *Events Inside Tasks*. Triar bé la granularitat té, doncs, un alt impacte en el rendiment de les aplicacions.

Organització del capítol

S'ha decidit articular aquest capítol detallant totes les iteracions realitzades. Per a cada iteració s'indica:

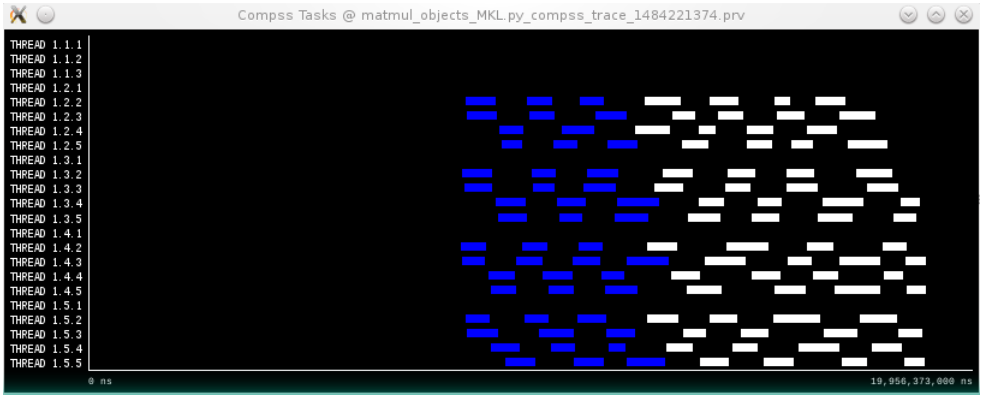

Figura 6 – Vista Compss Tasks amb `ComputingUnits = 2`

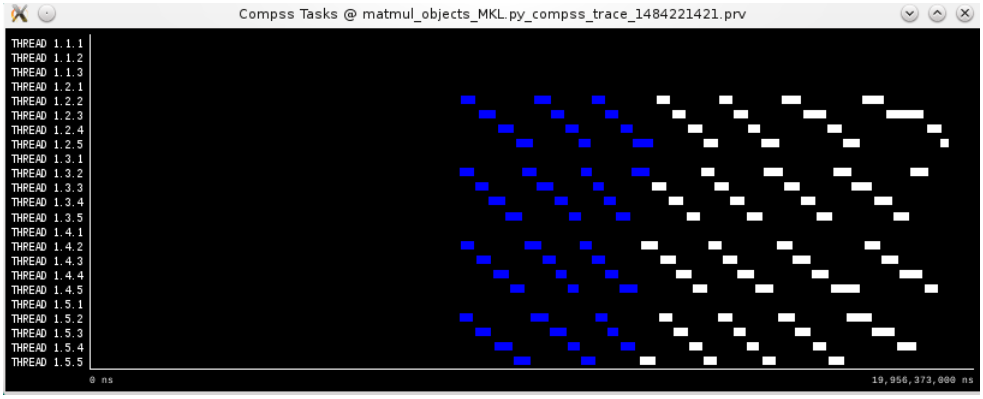

Figura 7 – Vista Compss Tasks amb `ComputingUnits = 4`

1. Anàlisi
2. Solució
3. Resultats

S'ha cregut que era la forma més lògica ja que d'aquesta forma s'explica la raó de totes les implementacions. A més a més, el document està organitzat segons l'ordre en el que cadascuna de les decisions ha estat presa.

3.2 Primera iteració

Anàlisi

L'objectiu d'aquesta fase consisteix en la detecció de les fonts d'overhead i la proposició d'actuacions que milloren el rendiment de la multiplicació de matrius. Els resultats a l'inici del projecte es mostren a la taula 4. Després de realitzar molts tests, es va trobar que el rendiment màxim s'obtenia amb `ComputingUnits=N` on `N` es el nombre d'unitats de còmput de cada node i `MKLProc = 1`. Aquests resultats inviten a pensar que *MKL* no es tan eficient com es podria pensar, ja que els resultats obtinguts són millors paral·lelitzant amb *PyCOMPSs* que amb *MKL*. Per limitar el nombre de variables analitzades, s'utilitza un sol worker, assegurant que no hi haurà transferències entre workers durant l'execució.
Per calcular el temps d'execució de la multiplicació s'utilitzava la comanda `compss_wait_on(c)` de l'*API* de `PyCOMPSs` on `c` és la matriu resultat. Aquesta comanda espera la finalització del càlcul de `c` i transfereix aquest paràmetre al màster. No es van executar tests més grans ja que amb matrius més grans s'obtenia un error indicant que es superava l'espai disponible en el node master.
Pel que fa al nombre d'operacions de coma flotant efectuades en cada execució, considerant que la matrius resultat s'inicialitza a zero i cal acumular-hi el resultat de la primera multiplicació de bloc, el nombre total d'operacions es el mostrat a l'equació 3.2, on `N` és el nombre de blocs presents a cada fila/columna de la matriu i `M` és el nombre de reals per fila/columna.
És important recordar que durant tot el document es tractarà amb matrius i blocs quadrats.

$$flops = 2 \cdot (N \cdot M)^3 \tag{3.3}$$

[1] No es té en compte el temps d'inicialització de les matrius
[2] El rendiment màxim teòric del nodes utilitzats és de 332 GFlops/s

Mida de la matriu	Mida dels blocs	Temps de multiplicació[1](s)	GFlops/s	Eficiència[2]
8	64	41.04	0.0065	0.0020%
16	64	231.41	0.0093	0.0028%
32	64	1767.46	0.0097	0.0029%
16	128	242.03	0.0710	0.0214%
16	256	245.06	0.5608	0.1689%
16	512	231.43	4.7509	1.4310%

Taula 4 – Rendiment de la multiplicació de matrius a l'inici del projecte

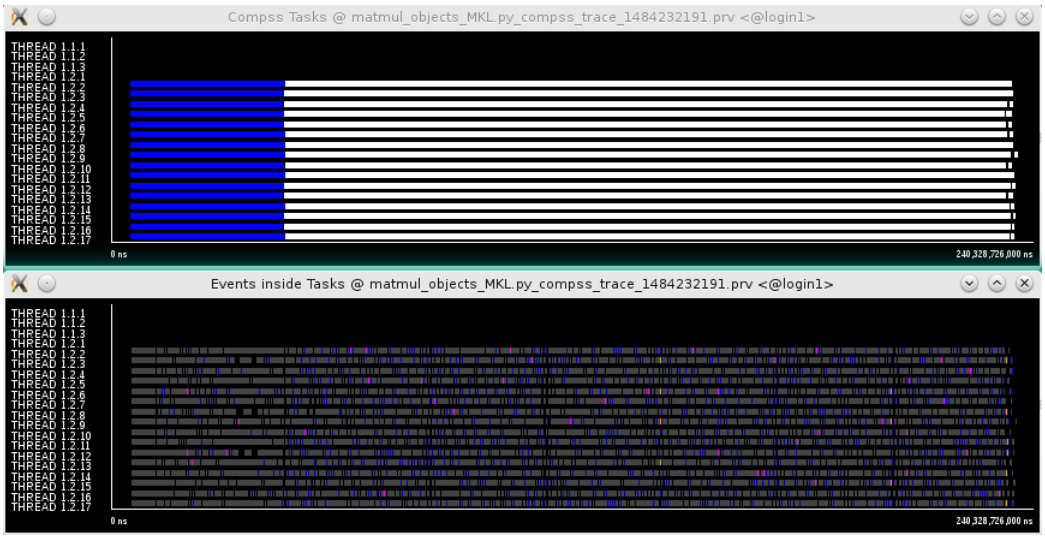

Figura 8 – Vistes *Compss Tasks* i *Events Inside Tasks* per a la multiplicació de matrius amb mida de matriu 16 i mida de bloc 64

A més a més, es va observar que l'aplicació escalava quasi idealment amb un estudi de tipus *strong scaling*, `mida de matriu` 32 i `mida de bloc` 64.
Per explicar aquest comportament s'utilitza la traça obtinguda per al cas amb mida de matrius 16 i mida de bloc 64. A la figura 8 es mostra com primer de tot s'inicialitzen les matrius per a, posteriorment, calcular la multiplicació. Degut a la granularitat de les tasques, la vista *Events Inside Tasks* no resulta especialment útil. Tenint en compte que es tracta del primer anàlisi, es mostra la vista general de les dues vistes. Als pròxims anàlisis, sol es mostrarà si fent-ho s'aporta alguna informació addicional.

A la figura 9 es mostra una ampliació de les traces anteriors. Les dues vistes es troben sincronitzades, per tant, mostren el mateix lapse de temps a idèntica escala.
La primera conclusió que podem extreure és que, de tot el temps d'execució, es passa molt poc temps realitzant càlculs[3]. Aquesta conclusió explica, alhora,

l'escalabilitat quasi perfecta trobada. Tenint en compte que els blocs són tant petits, el temps de transferència entre nodes es quasi negligible. Per tant, l'únic que estem fent quan augmentem els recursos dels que disposa l'execució es paralelitzar el temps d'inicialització de les tasques que, com es pot veure, ocupa la majoria del temps d'execució relacionat amb cada tasca.

Aquest primer cas ens indica que cal ser extremadament caut alhora d'utilitzar la vista *Compss Tasks*. Un codi que aparentment té un paral·lelisme quasi perfecte pot tenir en realitat un overhead desproporcionat respecte el temps d'execució seqüencial. Aquest fet també indica que cal utilitzar els `speed up` amb precaució. Caldrà, primer de tot, verificar que l'aplicació té un rendiment acceptable amb un sol worker. Quan sigui el cas, es podrà parlar de speed up per assenyalar que l'algoritme és bo. Altrament, no podrem dir res de l'algoritme ja que existeix la possibilitat de trobar-nos en una situació anàloga a l'anterior.
La primera decisió que es va prendre va ser, doncs, la d'augmentar el la mida dels blocs de forma que el temps de càlcul fos significatiu enfront l'overhead introduït per a paral·lelitzar les tasques.

Solució

Per a poder modificar la granularitat de les tasques augmentant el temps de càlcul de les mateixes sense trobar l'esmentat error de falta de memòria al node master, els desenvolupadors de *COMPSs* van afegir una nova crida a l'*API* de *PyCOMPSs* anomenada `wait_for_all_tasks()`. Quan es troba aquesta crida, es bloqueja l'execució de les instruccions que es troben a continuació fins que l'execució de totes les tasques hagi finalitzat. Cal tenir en compte, però, que la utilització d'aquesta crida no implica cap tipus de transferència de dades.

Resultats

Després de modificar el codi per a realitzar les esperes mitjançant la crida `wait_for_all_tasks()` enlloc d'utilitzar la crida `compss_wait_on(c)` es va procedir a la realització d'un estudi de granularitat. Per al disseny d'aquests tests cal garantir que el potencial paral·lelisme de les aplicacions sigui d'almenys el nombre d'unitats de còmput de cada node. Per aconseguir-ho, es van realitzar les execucions presentades a la taula 5.
Es tracta d'una multiplicació de dues matrius de mida total 32768x32738. A més a més, es va decidir que `ComputingUnits=2` i `MKLProc=2` per garantir que

[3]Els fragments taronges es indiquen l'execució del codi de la tasca

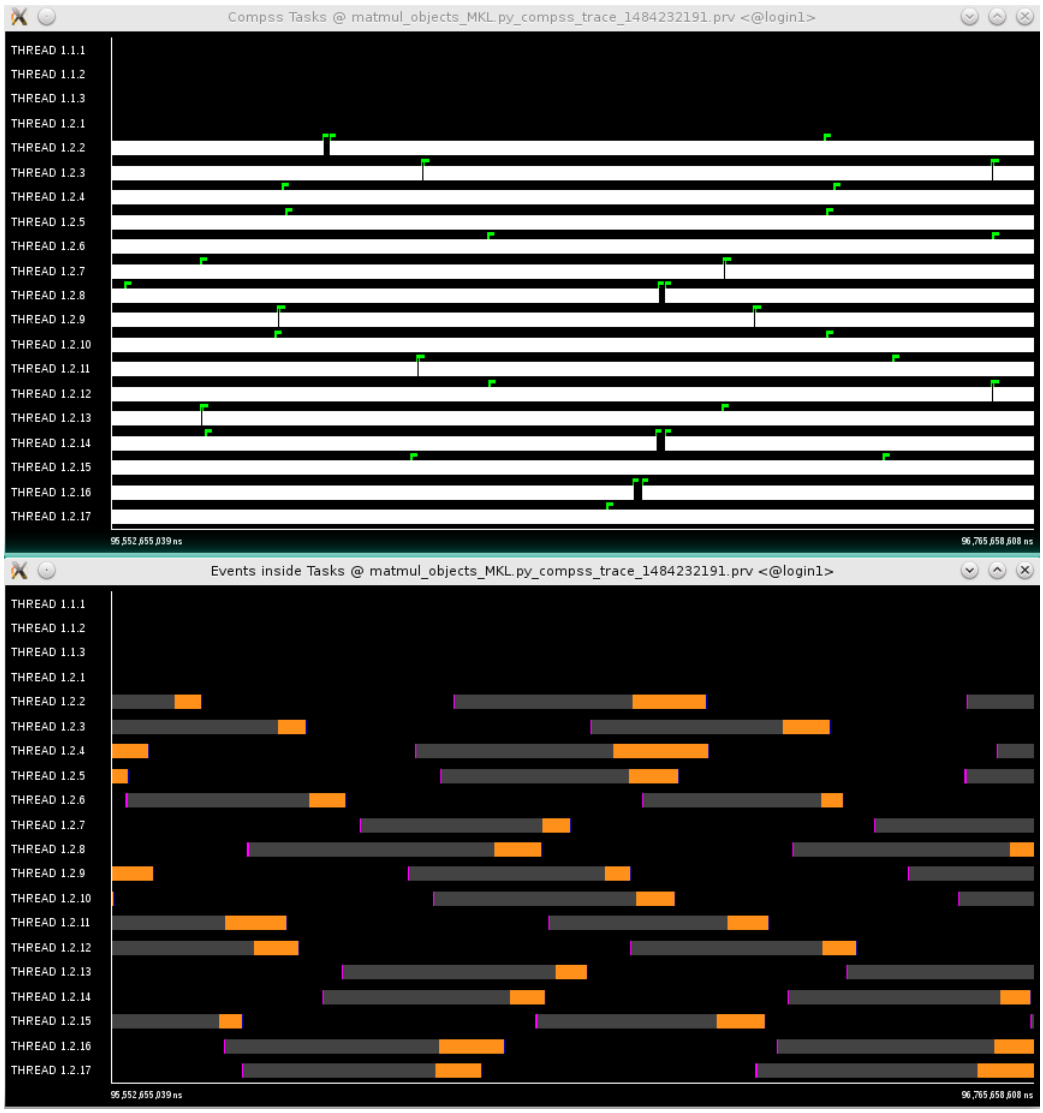

Figura 9 – Ampliació de les vistes *Compss Tasks* i *Events Inside Tasks* per a la multiplicació de matrius amb mida de matriu 16 i mida de bloc 64

s'assolís el potencial paral·lelisme del worker.

Mida de la matriu	Mida dels blocs	Temps de multiplicació[4](s)	GFlops/s	Eficiència[5]
4	8192	636.59	110.54	33.30%
8	4096	1169.84	60.15	18.12%
16	2048	2540.53	27.70	8.34%

Taula 5 – Rendiment de la multiplicació de matrius disminuint la granularitat

Com es pot observar, l'eficiència de la multiplicació passa d'un 1.43% a un 33.3%. A més a més, s'observa que mantenint la mida total de la matrius constant, dividint per dos la granularitat s'augmenta l'eficiència amb un factor igual. En aquest punt cal tenir en compte que si es divideix per dos la granularitat es divideix per 8 el nombre de tasques creades. Si es pren la hipòtesi que l'overhead d'inicialització de cada tasca es constant, s'està dividint l'overhead total per un factor 8. Cal tenir en compte, però, que aquest overhead esdevé negligible a partir d'una certa mida de bloc.
Un anàlisi de les traces obtingudes permet corroborar que les hipòtesi fetes durant la fase d'anàlisi eren correctes. Concretament, la figura 10 mostra un esquema més semblant al que es podria esperar, amb un temps d'inicialització de blocs molt inferior al temps que s'inverteix en calcular les multiplicacions. A més a més, realitzant un `zoom sincronitzat` com el mostrat a la figura 11 es pot comprovar que amb aquesta granularitat quasi la totalitat del temps de les tasques de multiplicació s'inverteix realitzant el còmput i no en operacions d'overhead.
A més, s'ha trobat que la mida de bloc mínima a partir de la qual el temps de còmput passa a ser significatiu envers l'overhead és de 2048, tot i que es recomanable utilitzar una mica de bloc de 4096. Es tracta de mides que permeten realitzar comparacions vàlides entre diferents execucions. Si el que es desitja es obtenir màxima eficiència, caldrà realitzar tots els càlculs amb mida de bloc de 8192. Es fa aquesta remarca ja que per tal d'evitar execucions massa llargues, al llarg d'aquest projecte s'han realitzat tests amb mida de bloc de 4096.

3.3 Segona iteració

Anàlisi

Per comprovar el potencial guany obtingut utilitzant *MKL* en aquesta fase del projecte, es va procedir a l'execució del mateix càlcul amb la següent parella de paràmetres:

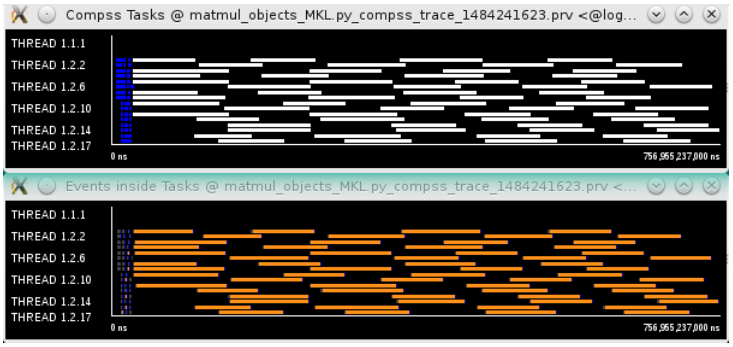

Figura 10 – Vistes *Compss Tasks* i *Events Inside Tasks* per a la multiplicació de matrius amb mida de matriu 4 i mida de bloc 8192

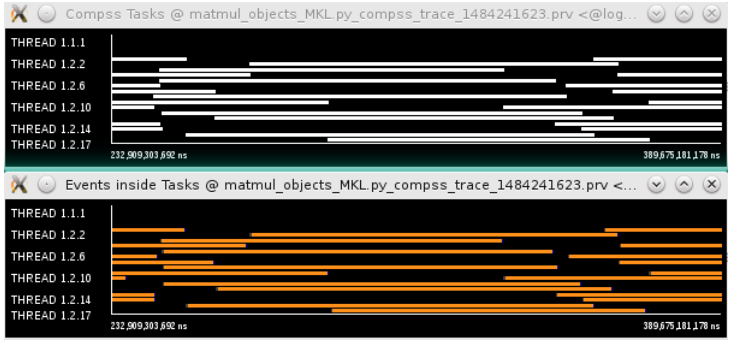

Figura 11 – Ampliació de les vistes *Compss Tasks* i *Events Inside Tasks* per a la multiplicació de matrius amb mida de matriu 4 i mida de bloc 8192

1. MKL seqüencial
   `ComputingUnits=1`, `MKLProc=1`

2. Quantitat òptima teòrica de threads MKL per a l'arquitectura de *MareNostrum*
   `ComputingUnits=16`, `MKLProc=16`

En els dos casos s'utilitza un sol worker. Els resultats obtinguts es mostren a la taula 6.

ComputingUnits	MKLProc	Mida de la matriu	Mida dels blocs	Temps de multiplicació (s)	GFlops/s	Eficiència
1	1	16	1024	468.58	18.7718063558	5.65%
1	1	8	2048	314.64	27.9560546091	8.42%
1	1	4	4096	160.06	54.9549732738	16.55%
16	16	16	1024	2688.56	3.2716744362	0.99%
16	16	8	2048	1065.3	8.2569163824	2.49%
16	16	4	4096	297.19	29.5975403688	8.91%

Taula 6 – Rendiment de la multiplicació de matrius amb MKL seqüencial o paral·lel

En aquest cas s'observa l'efecte de falsa escalabilitat comentat anteriorment per a les granularitats altes. Si es realitza la paral·lelització amb *PyCOMPSs* s'obté un rendiment molt més alt ja que s'està executant l'overhead en paral·lel. No obstant, aquest fet no explica perquè el rendiment segueix sent molt millor per a les granularitats baixes.
Per tal d'entendre millor aquest fenomen es va procedir a la lectura acurada de la documentació tant d'*MKL* com dels fòrums oficials per a desenvolupadors. Aquesta recerca va donar com resultat el fet que quan hi ha més d'un processador en un node, els threads es queden encapsulats al socket que els crea [22].

Solució

Per a solucionar aquest problema es van trobar dues alternatives. La primera consisteix a definir manualment a dintre de cada tasca quins processadors executaran els threads creats per *MKL* mitjançant la variable d'entorn `KMP_AFFINITY`[23]. A la figura 12 es mostra com queda el codi de la tasca utilitzant aquest solució.
Aquesta solució tot i ser la més rapida d'implementar presenta els següents problemes:

- Cal que l'usuari la modifiqui manualment
- Es col·loca un codi dintre de la tasca que es depenent de l'arquitectura

```
@constraint (ComputingUnits="${ComputingUnits}")
@task(c=INOUT)
def multiply(a, b, c, MKLProc):
  os.environ["MKL_NUM_THREADS"]=str(MKLProc)
  os.environ["KMP_AFFINITY"]="verbose,granularity=thread,proclist
    =[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],explicit"
c += a * b
```

Figura 12 – Thread affinity amb la variable d'entorn `KMP_AFFINITY`

```
tasksetPython.append("taskset -c ");
if(assignedCoreUnits.length > 8){
  tasksetPython.append("0-15 ");
}
else{
  if(assignedCoreUnits[0] < 8){
    tasksetPython.append("0-7 ");
  }
  else{
    tasksetPython.append("8-15 ");
  }
}
lArgs.add(tasksetPython.toString());
```

Figura 13 – Thread affinity amb la crida `taskset`

Tenint en compte els anteriors inconvenients, es va utilitzar aquesta solució per a comprovar que hi havia una diferència de rendiment introduint aquesta modificació. Una vegada realitzada aquesta comprovació ràpida, es va procedir a la recerca d'una solució més elegant per integrar-la al codi intern de *COMPSs* de forma que es pugui aprofitar per a altres aplicacions.
Com s'ha dit anteriorment, un dels problemes trobats amb granularitats altes és que es paga el temps d'iniciar un intèrpret de *Python* cada vegada que s'executa una tasca. Aquesta acció es realitza des d'un script en `bash` de la forma `python [nom de l'script] [paràmetres]`. Un anàlisi de les funcionalitats proporcionades pel sistema operatiu `linux` va portar al descobriment de les comandes `taskset` i `numactl`. Aquestes comandes tenen una altra comanda com paràmetre. Entre les seves funcionalitats es troba el fet que permeten definir l'afinitat de tots els threads creats pel procès invocat a través de la crida. Tenint aquest fet en compte, es va modificar el codi del l'invocador del worker dintre del codi de *COMPSs* com es mostra a la figura 13.
Cal remarcar que es tracta d'una implementació que sol funciona per a les arquitectures amb dos socket on cadascun dels sockets disposa de 8 *CPU*'s. A més a més, es fa la hipòtesi de que el nombre de `ComputingUnits` de cada tasca serà un divisor del nombre de *CPU* present a cadascun dels sockets,

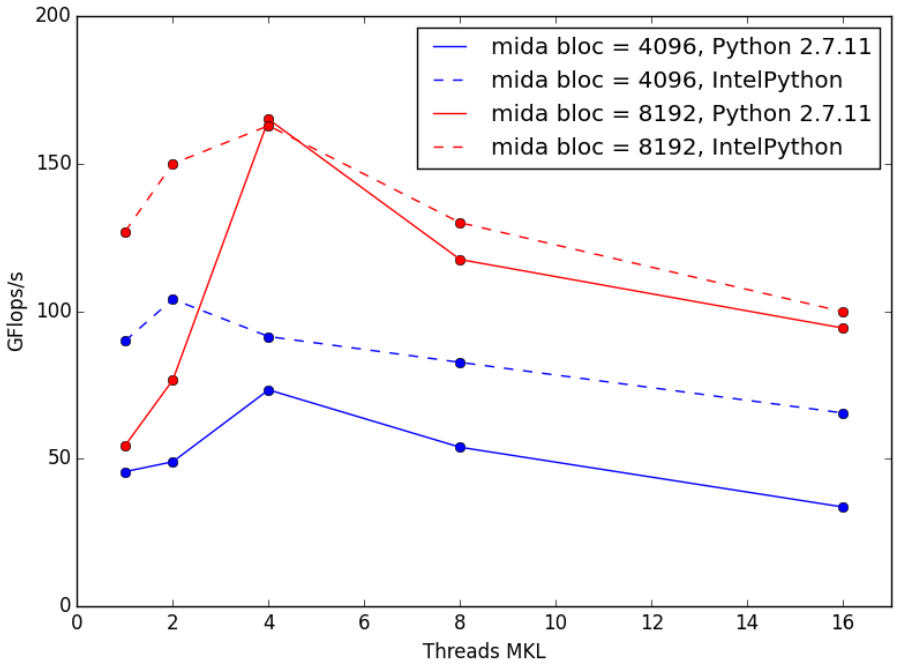

Figura 14 – GFlops/s en funció del nombre de threads `MKL`

assegurant que sempre es possible que una tasca sempre es pugui planificar dintre d'un sol socket independentment de la planificació de tasques anterior.

Resultats

Una vegada introduïdes les modificacions, s'han realitzat diversos tests per trobar la configuració que dona un millor rendiment. Primer de tot, s'ha decidit variar els valors de `ComputingUnits` i `MKLProc` de forma que la multiplicació dels dos fos igual a 16, es a dir, al nombre de *CPU*'s del node. D'aquesta manera es pretén crear tants threads de còmput com *CPU*'s disponibles.
D'altra banda, durant les fases d'anàlisi i recerca de la solució es va instal·lar a *MareNostrum* una versió de *IntelPython*. Segons *Intel*® aquesta implementació afegeix una capa d'optimització a la ja present dintre de *MKL*. Segons el fabricant de processadors, les diferencies més notables es troben, precisament, en la gestió dels threads.
Els resultats obtinguts es poden consultar a la figura 14. S'han multiplicat dues matrius de mida total 32768x32768.

Com es pot veure, per primera vegada s'obté un millor rendiment amb la utilització de *MKL*. Concretament, amb una mida de bloc de 4096 s'arriba fins als 73.37 GFlops/s (22.09% d'eficiència) enfront els 54.95 GFlops/s (16.55% d'eficiència) obtinguts amb la paral·lelització intranode realitzada íntegrament amb *PyCOMPSs*.
Pel que fa al cas en el que s'utilitza una mida de bloc igual a 8192, s'arriba fins als 165.05 GFlops/s, es a dir, una eficiència del 49.71%.
La darrera conclusió que es pot extreure d'aquesta bateria d'execucions és que, efectivament, la implementació *IntelPython* aconsegueix un rendiment superior a la implementació per defecte.
A partir de les consideracions anteriors, es fixen els següents paràmetres per a les execucions futures:

- `ComputingUnits=4`
- Utilització d'*IntelPython*

Des del principi s'ha assumit que el més òptim seria crear un thread per *CPU*. No obstant, una vegada es comencen a utilitzar matrius grans s'augmenta el nombre de fallades de cache, el temps de transferència entre memòria i disc i el temps de serialització (entre dues tasques, totes les dades es serialitzen i es guarden a disc per a ser carregades una altra vegada amb posteritat). A aquest fet cal sumar la disponibilitat d'*IntelPython*, que assegura una òptima gestió dels threads.
A partir de l'exposat al paràgraf anterior, es va explorar la idea de provocar `oversubscriving` per veure com variava el rendiment de les operacions. Els resultats es poden veure a la figura 15. S'ha utilitzat una mida de matriu de 4 per als blocs amb mida 8192 i de 8 per als blocs de mida 4092 (és a dir, una mida constant de 32768). La mida total de la matrius es manté, doncs, constant. A més a més, s'ha considerat `ComputingUnits=4`. Per variar el grau de l'oversubscriving s'ha fet que el valor de `MKLProc` vagi de 4 fins a 32.

Com es pot observar, s'arriba a un rendiment de 203 GFlops/s (61,21% d'eficiència) per als blocs de mida 8192 i de 97.5 GFlops/s (29.37% d'eficiència) per als blocs de mida 4096.
A continuació es va procedir a realitzar l'anàlisi d'strong scaling. Els resultats es mostren a la figura 16. S'ha utilitzat una mida de matriu igual a 16, de forma que el grau de paral·lelisme sigui suficient com per a, teòricament, aprofitar els 64 workers.

Com es pot veure, l'aplicació escala molt bé fins als 16 workers. Aquest estudi ha permès extreure conclusions molt interessants. S'ha vist que augmentar la mida de la matrius fa guanyar rendiment, però augmentar-la massa en fa perdre. En el cas d'un sol worker, s'ha realitzat el càlcul a un ritme de 76.56 GFlops/s. Aquest fet suposa una davallada notable del rendiment màxim assolit anteriorment.

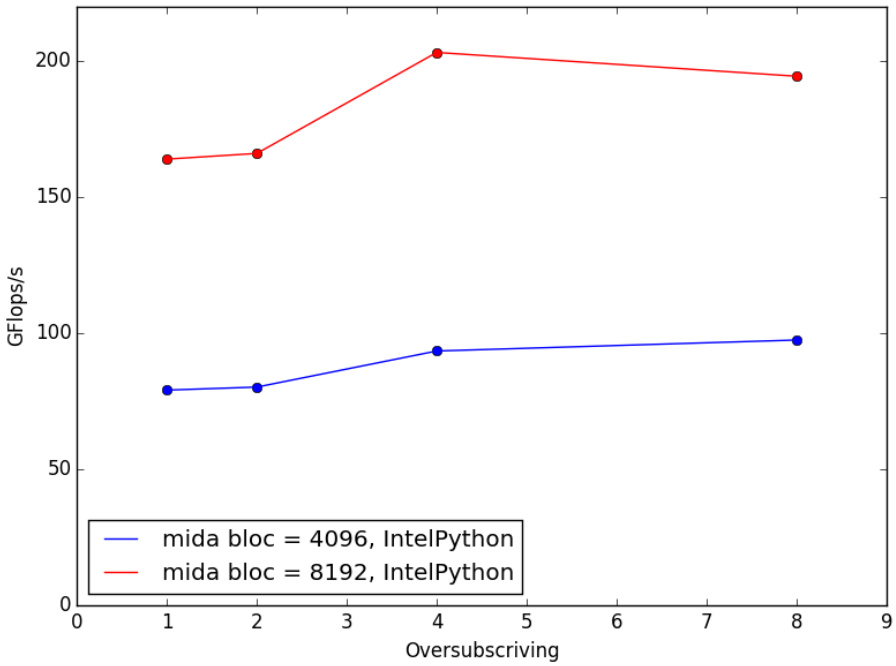

Figura 15 – GFlops/s en funció del grau d'oversubscriving

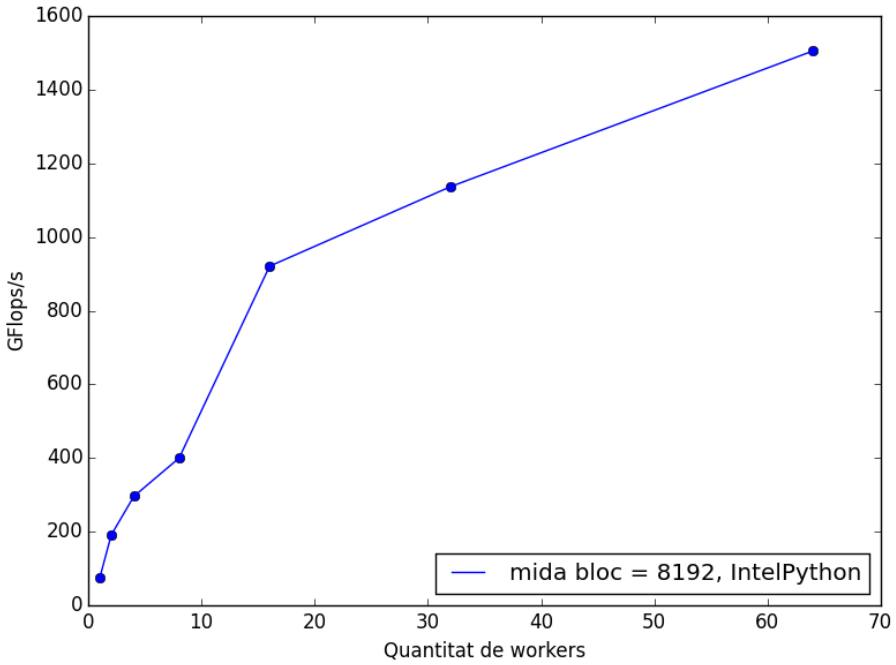

Figura 16 – GFlops/s en funció del nombre de workers

Mida total de la matriu	Espai utilitzat[6](GB)
8192	8
16384	16
32768	32
65536	64
131072	128

Taula 7 – Espai ocupat per les matrius en funció de la seva mida

Analitzant l'arquitectura en la qual s'han realitzat els tests (la memòria dels nodes utilitzats és de 64 o 128 GB) i el comportament del sistema operatiu utilitzat[24] s'ha arribat a la conclusió que si s'obtenia tan bon rendiment era gràcies al fet que en cap moment es passava per disc per carregar els blocs. Tot i que *COMPSs* realitzava l'operació de llegir de disc, el sistema operatiu ja tenia els fitxers en memòria. Linux sempre emplena la *RAM* seguint un comportament similar al d'una *cache* per tal d'obtenir el màxim rendiment possible. Aquest fet fa que si s'utilitza la comanda `top` en un equip, el més probable és que la memòria estigui quasi plena. No obstant, aquest fet no suposa un problema si una part significativa d'aquest espai es troba en l'apartat de `cached Mem`.
En un segon terme, s'ha apreciat que a partir dels 16 workers l'escalabilitat de l'aplicació deixa de ser tan bona com amb menys workers. S'ha associat aquest fet a l'augment de les col·lisions en la consulta de dades. El grau màxim de paral·lelisme per a una multiplicació de matriu de mida 16 és de 256 (cada unitat de còmput calcula una posició de la matriu resultat). No obstant, si es treballa en aquestes condicions hi ha moltes col·lisions de dades. Aquest fet implica un augment de les comunicacions. A més a més, cal tenir en compte que cada worker sol té un port de comunicacions. Per tant, si les 4 unitats de càlcul presents (en aquesta bateria d'execucions, s'ha utlitzat `ComputingUnits=4`) necessiten cadascun una dada en un moment donat, les 4 transferències es realitzaran de forma seqüencial.
Aquest últim test ha permès comprovar la importància de la tria de la mida dels blocs de forma que sigui òptima per a l'arquitectura disponible i de la planificació per tal d'evitar al màxim el nombre de transferències entre workers.

[6]Es considera que en *Python* un nombre real ocupa 8 bytes

Capítol 4

Implementació de nous algoritmes

Índex

Una vegada optimitzat el codi de la multiplicació de matrius, es passa a l'estudi de codis més complexos.

4.1 Factorització de Cholesky

Introducció

La factorització de *Cholesky*, inventada pel matemàtic francès que li dona nom, es pot aplicar en matrius hermitianes definides positives[18]. Aquesta descomposició és un cas pas particular de la descomposició $\mathscr{L}\mathscr{U}$. Concretament, es busquen dues matrius de la forma $\mathscr{U} = \mathscr{L}^t$.

Implementació

Per a la implementació es va modificar la versió existent de l'algoritme per realitzar la inicialització de forma distribuïda i es van modificar algunes crides per augmentar el grau de paral·lelisme. Aquesta versió utilitzava el mètode clàssic[25].

```
def cholesky_blocked(A):
  n = len(A)
  bSize = len(A[0][0])
  cont = 0
  for k in range(n):
    # Diagonal block factorization
    A[k][k] = potrf(A[k][k])
    # Triangular systems
    for i in range(k+1, n):
      A[i][k] = trsm(A[k][k], customTranspose(A[i][k]))
      A[k][i] = np.zeros((bsize,bsize))

    #Update trailing matrix
    for i in range(k+1, n):
      for j in range(i, n):
        A[j][i] = gemm(-1.0, A[j][k], A[i][k], A[j][i], 1.0)

  return A
```

Figura 17 – Algoritme per obtenir la descomposició de Cholesky per blocs

A la figura 17 es presenta la implementació resultant de la descomposició de *Cholesky* per blocs. Cal tenir en compte que s'han creat tasques amb el mateix nom de les funcions *NumPy* per a que el codi paral·lelitzat sigui el més semblant possible al codi seqüencial. Concretament, les següents crides generen tasques:

1. `potrf`: calcula la factorització de Cholesky
2. `trsm`: resol un sistema triangular de la forma $A \cdot X = B$
3. `customTranspose`: transposa una matriu
4. `gemm`: retorna el resultat de l'operació $\alpha \cdot A \cdot B + \beta C$

Una anàlisi acurat de l'algoritme i la consulta de la documentació oficial de *NumPy* permet construir la taula 8[1]. Aquesta mostra el nombre de crides a cadascuna de les tasques creades així com la seva complexitat interna.

A partir del que es mostra a la taula 8 es mot construir l'equació 4.1. En aquesta es mostra el càlcul de la complexitat total de l'algoritme. Com es pot veure, la complexitat respecte a l'algoritme que no utilitza blocs no es veu augmentada.

[1] M=mida de la matriu, N=mida de bloc

Funció	Crides	Complexitat
`potrf`	M	$\frac{N^3}{3}$
`trsm`	$\frac{M \cdot (M+1)}{2}$	$2 \cdot N^3$
`gemm`	$M \cdot \frac{M \cdot (M-1)}{6}$	$2 \cdot N^3$

Taula 8 – Crides fetes a la llibreria *NumPy* i la seva complexitat en la descomposició de *Cholesky* per blocs

$$\begin{aligned} \text{Complexitat Cholesky} &= M\frac{N^3}{3} + \frac{M(M+1)}{2}2N^3 + M\frac{M(M-1)}{6}2N^3 \\ &= \frac{MN^3}{3} + M^2N^3 + MN^3 + \frac{M^3N^3 - M^2N^3}{3} \approx \frac{(MN)^3}{3} \end{aligned} \tag{4.1}$$

Una execució d'aquest algoritme utilitzant una mida de matriu igual a 8 genera el graf de dependències presentat a la figura 18. les tasques blaves es corresponen amb les inicialitzacions de les matrius, les blanques amb `potrf`, les vermelles amb `trsm` i les roses amb `gemm`.

La figura 19 mostra la traça obtinguda amb una mica de matriu igual a 8 i una mida de bloc de 4096. A més a més, s'ha utilitzat `ComputingUnits=4` i `MKLProc=16`, es a dir, un grau d'oversubscriving igual a 4. L'anàlisi de la mateixa mostra el que ja es podia intuir en el graf de dependències: l'algoritme té un bon grau de paral·lelisme. A més a més, la granularitat és el suficientment baixa, ja que el temps de càlcul és significatiu respecte el temps total d'execució de la mateixa.

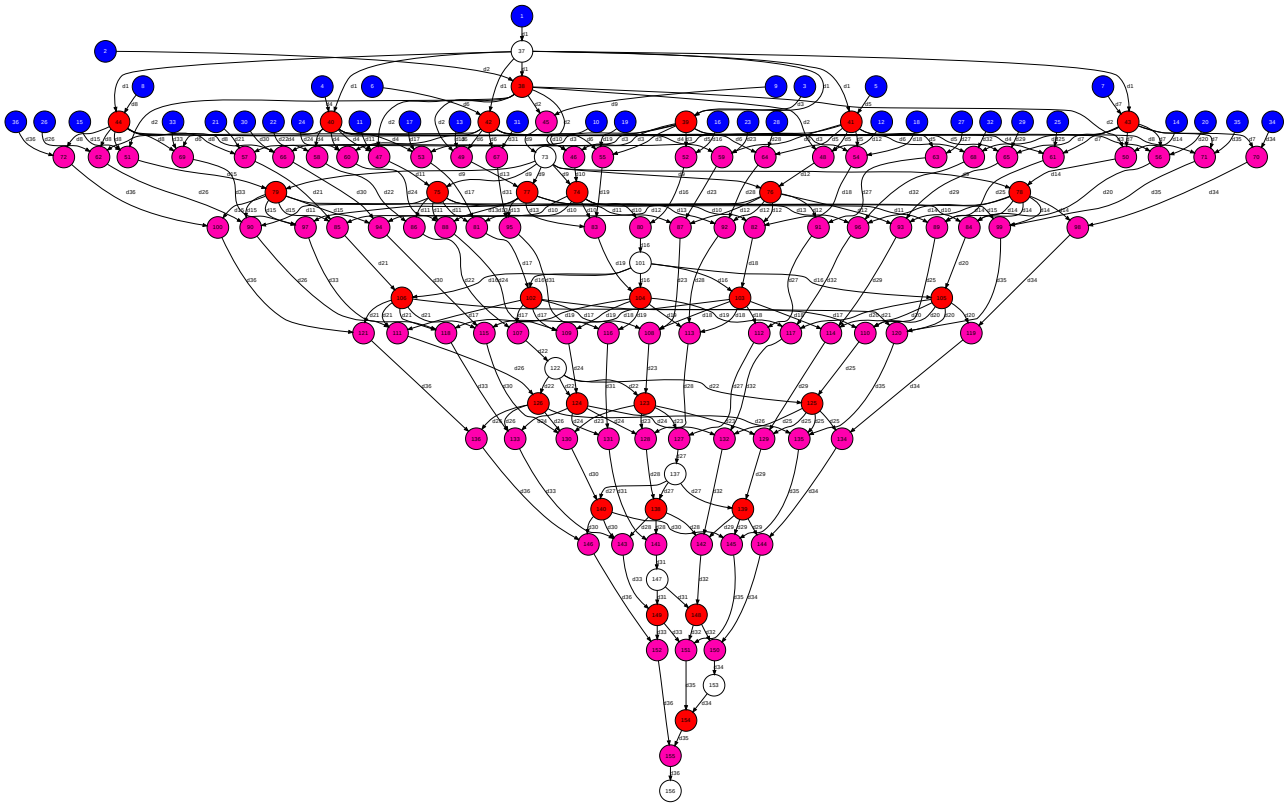

Figura 18 – Graf de dependències corresponent a la factorització de Cholesky

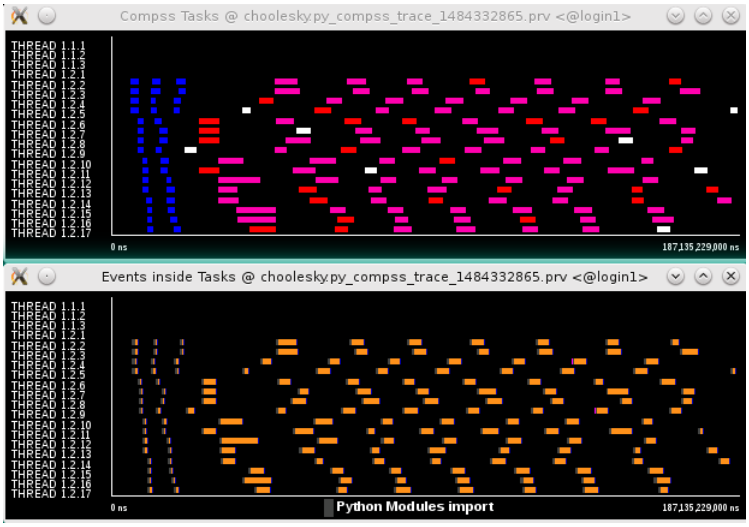

Figura 19 – Vistes *Compss Tasks* i *Events Inside Tasks* per a la factorització de Cholesky amb mida de matriu 8 i mida de bloc 4096

4.2 Descomposició QR

Introducció

La factorització QR consisteix en trobar la descomposició d'una matriu com producte d'una matriu ortogonal multiplicada per una triangular superior. Concretament, la descomposició serà de la forma $A = QR$ tal que $Q^tQ = I$ i R triangular superior. S'utilitza sobretot per al càlcul dels valors propis d'una matriu i, posteriorment, els seus vectors propis.
S'ha escollit aquest algoritme ja que és una de les implementacions d'àlgebra lineal més complicades. A més a més, les solucions clàssiques proposen solucions on la matriu es descompon per columnes. Aquesta decisió es pren per raons d'eficiència. Tenint en compte que tots els algoritmes van transformant la matriu d'origen per posar-hi zeros sota la diagonal, el fet de treballar per columnes fa que cadascuna de les transformacions aplicades s'apliqui sol a elements que interessi. Aquest és el cas de l'algoritme de *Householder*[26], utilitzat en la implementació interna de *MKL*.
En aquest cas, hi ha un gran interès per emmagatzemar la matriu per blocs quadrats per a integrar-la més fàcilment amb les altres operacions ja programades.

Implementació

Per a implementar aquesta solució es va realitzar una recerca bibliogràfica. Com s'ha assenyalat a la introducció, en general tots els algoritmes distribuïts utilitzen la descomposició mitjançant columnes enlloc de blocs quadrats. Tenint en compte el posterior interès d'integrar aquesta funció en una llibreria matemàtica, el més interessant és utilitzar un sol tipus de descomposició per evitar conversions innecessàries. S'ha decidit d'emmagatzemar les matrius en blocs quadrats.
La solució trobada realitza una sèrie de rotacions de Givens a nivell de bloc fins a obtenir una matriu triangular superior[27]. A continuació es mostra una iteració completa de l'algoritme:

1. Descomposició de la matriu en blocs[2]

$$A = \begin{pmatrix} M^{0,0} & M^{0,1} & \dots & M^{0,N} \\ M^{1,0} & M^{1,1} & \dots & M^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M^{N,0} & M^{N,1} & \dots & M^{N,N} \end{pmatrix}$$

[2]N=mida de la matriu

2. Càlcul de la descomposició QR del bloc situat a la fila i columna amb índexs més baixos

$$M^{i,i} = Q^{i,i}\tilde{R}_i^{i,i} \rightarrow A' = \begin{pmatrix} (Q^{0,0})^t & 0 & \dots & 0 \\ 0 & I & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & I \end{pmatrix} \begin{pmatrix} M^{0,0} & M^{0,1} & \dots & M^{0,N} \\ M^{1,0} & M^{1,1} & \dots & M^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M^{N,0} & M^{N,1} & \dots & M^{N,N} \end{pmatrix} = \begin{pmatrix} \tilde{R}_0^{0,0} & \tilde{M}_0^{0,1} & \dots & \tilde{M}_0^{0,N} \\ M^{1,0} & M^{1,1} & \dots & M^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M^{N,0} & M^{N,1} & \dots & M^{N,N} \end{pmatrix} \tag{4.2}$$

3. Canvi de tots els blocs de la mateixa fila per zeros
   Per cadascun dels valors de la columna, caldrà seguir el següent procediment:

   (a) Construcció de la matriu rotació mitjançant una descomposició parcial QR

$$\begin{pmatrix} \tilde{R}_j^{i,i} \\ \tilde{M}^{j+1,i} \end{pmatrix} = \begin{pmatrix} \Sigma^{0,0} & \Gamma^{0,1} \\ \Gamma^{1,0} & \Sigma^{1,1} \end{pmatrix} \begin{pmatrix} \tilde{R}_{j+1}^{i,i} \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} \tilde{R}_{j+1}^{i,i} \\ 0 \end{pmatrix} = \begin{pmatrix} (\Sigma^{0,0})^t & (\Gamma^{1,0})^t \\ (\Gamma^{0,1})^t & (\Sigma^{1,1})^t \end{pmatrix} \begin{pmatrix} \tilde{R}_j^{i,i} \\ \tilde{M}^{j+1,i} \end{pmatrix} \tag{4.3}$$

   (b) Afegir la rotació a la matriu Q parcial per a introduir un nou zero a la matriu R parcial Seguint el procediment anterior, es realitza la següent operació:

$$A'' = \begin{pmatrix} (\Sigma^{0,0})^t & (\Gamma^{1,0})^t & \dots & 0 \\ (\Gamma^{0,1})^t & (\Sigma^{1,1})^t & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & I \end{pmatrix} \begin{pmatrix} (Q^{0,0})^t & 0 & \dots & 0 \\ 0 & I & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & I \end{pmatrix} \begin{pmatrix} M^{0,0} & M^{0,1} & \dots & M^{0,N} \\ M^{1,0} & M^{1,1} & \dots & M^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M^{N,0} & M^{N,1} & \dots & M^{N,N} \end{pmatrix} = \begin{pmatrix} \tilde{R}_1^{0,0} & \tilde{M}_1^{0,1} & \dots & \tilde{M}_1^{0,N} \\ 0 & \tilde{M}^{1,1} & \dots & \tilde{M}^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M^{N,0} & M^{N,1} & \dots & M^{N,N} \end{pmatrix} \tag{4.4}$$

   (c) Repetir el pas anterior per a tots els elements de la fila

   Aplicant els passos anterior, s'obté una expressió com la següent:

$$A^{N)} = \tilde{Q}^0 A = \begin{pmatrix} \tilde{R}_N^{0,0} & \tilde{M}_N^{0,1} & \dots & \tilde{M}_N^{0,N} \\ 0 & \tilde{M}^{1,1} & \dots & \tilde{M}^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & M^{N,1} & \dots & M^{N,N} \end{pmatrix} = \begin{pmatrix} \check{R}^{0,0} & [\check{M}^0] \\ [0] & \check{A}_0 \end{pmatrix} \tag{4.5}$$

Posteriorment, caldrà aplicar l'algoritme de forma recursiva a totes les $\check{A}_i$ fins obtenir una matriu triangular superior.
Com es pot comprovar en l'operació anterior, el procediment per arribar a la matriu R final consisteix a realitzar una sèrie de canvis de base ortogonals sobre la matriu original. Al final del procès, es tindrà una expressió com la presentada a l'equació número 4.6.

$$R = \tilde{Q}^N \tilde{Q}^{N-1} \dots \tilde{Q}^1 \tilde{Q}^0 A \tag{4.6}$$

A partir de l'expressió 4.6 i considerant que el producte de transformacions ortogonals és una transformació ortogonal, es pot seguir el raonament presentat a l'expressió 4.7. Segons el mateix, el procediment anterior obté una descomposició de la matriu original en el producte d'una matriu ortogonal per un altra de triangular superior.

$$\hat{Q} = \tilde{Q}^N \tilde{Q}^{N-1} \dots \tilde{Q}^1 \tilde{Q}^0 \rightarrow R = \hat{Q}A \rightarrow {}^3A = \hat{Q}^t R \rightarrow Q = \hat{Q}^t \tag{4.7}$$

En base a l'explicat en aquesta secció, queda perfectament definit l'algoritme per obtenir les matrius Q i R a partir d'una matriu donada. La implementació del mateix es pot veure a la figura 20.

[3] $\hat{Q}$ ortogonal

```
def qr_blocked(A, MKLProc, overwrite_a=False):

  Q = genIdentity(MSIZE,BSIZE,MKLProc)

  if not overwrite_a:
    R = copyBlocked(A)
  else:
    R = A

  for i in range(MSIZE):

    actQ, R[i][i] = qr(R[i][i], MKLProc, transpose=True)

    for j in range(MSIZE):
      Q[j][i] = dot(Q[j][i], actQ, MKLProc, transposeB=True)

    for j in range(i+1,MSIZE):
      R[i][j] = dot(actQ,R[i][j], MKLProc)

    #Update values of the respective column
    for j in range(i+1,MSIZE):
      subQ = [[np.matrix(np.array([0])), np.matrix(np.array([0]))],
              [np.matrix(np.array([0])), np.matrix(np.array([0]))]]
      subQ[0][0],subQ[0][1],subQ[1][0],subQ[1][1],R[i][i],R[j][i] =
              littleQR(R[i][i],R[j][i],MKLProc,BSIZE,transpose=True)

      #Update values of the row for the value updated in the column
      for k in range(i+1,MSIZE):
        [[R[i][k]],[R[j][k]]] =
              multiplyBlocked(subQ, [[R[i][k]],[R[j][k]]], BSIZE,
  MKLProc)

      for k in range(MSIZE):
        [[Q[k][i], Q[k][j]]] =
              multiplyBlocked([[Q[k][i], Q[k][j]]], subQ,
                              BSIZE, MKLProc,transposeB=True)
  return Q,R
```

Figura 20 – Algoritme per obtenir la factorització QR per blocs

Com en el cas de la descomposició de *Cholesky*, s'han creat tasques amb el mateix nom de les funcions *NumPy* per a que el codi paral·lelitzat sigui el més semblant possible al codi seqüencial. Concretament, les següents crides generen tasques:

1. `qr`: calcula la factorització QR d'un bloc de mida NxN
2. `dot`: retorna el resultat de multiplicar dos blocs de mida NxN
3. `littleQR`: calcula la factorització QR d'un bloc de mida $2NxN$
4. `multiplyBlocked`: retorna el resultat de multiplicar una matriu de mida 2 blocs x 2 blocs i una altra de 2 blocs x 1 bloc

S'ha consultat la documentació oficial de *NumPy* per tal de conèixer les implementacions internes de les funcions utilitzades i poder construir la taula 9[4] que mostra les crides a cadascuna de les tasques creades així com la seva complexitat computacional.

Funció	Crides	Complexitat
`qr`	M	$2 \cdot N^3$
`dot`	$M^2 + \frac{M \cdot (M-1)}{2}$	$2 \cdot N^3$
`littleQR`	$\frac{M \cdot (M-1)}{2}$	$4 \cdot N^3$
`multiplyBlocked`	$\frac{M^2(M-1)}{2} + \frac{M \cdot (2M^2 - 3M + 1)}{6}$	$8 \cdot N^3$

Taula 9 – Crides fetes a la llibreria *NumPy* i la seva complexitat en la descomposició de *QR* per blocs

A partir de la taula 9, es pot construir l'equació 4.8 trobant l'expressió de la complexitat computacional de l'algoritme.

[4] M=mida de la matriu, N=mida de bloc

$$
\begin{aligned}
\text{Complexitat QR} &= 2MN^3 + 2\left(M^2 + \frac{M\cdot(M-1)}{2}\right)N^3 + 4\left(\frac{M\cdot(M-1)}{2}\right)N^3 \\
&+ 8\left(M^2\frac{(M-1)}{2} + \frac{M\cdot(2M^2-3M+1)}{6}\right)N^3 \\
&= 2MN^3 + 2M^2N^3 + M^2N^3 - MN^3 + 2M^2N^3 - 2MN^3 \\
&+ 8N^3\left(\frac{M^3}{2} - \frac{M^2}{2} + \frac{2M^3}{6} - \frac{3M^2}{6} + \frac{M}{6}\right) \\
&= 12\frac{MN^3}{6} + 12\frac{M^2N^3}{6} + 6\frac{M^2N^3}{6} - 6\frac{MN^3}{6} + 12\frac{M^2N^3}{6} - 12\frac{MN^3}{6} \\
&+ 24\frac{M^3N^3}{6} - 24\frac{M^2N^3}{6} + 16\frac{M^3N^3}{6} - 24\frac{M^2N^3}{6} + 8\frac{MN^3}{6} \\
&= \frac{20M^3N^3}{3} + 3M^2N^3 + \frac{MN^3}{3} \approx \frac{20N^3M^3}{3} = \frac{20}{3}(NM)^3
\end{aligned}
\tag{4.8}
$$

Com s'ha vist, en aquest cas el cost de l'algoritme augmenta notablement. Aquest fet es deu a dos factors combinats:

- No s'ha utilitzat la versió més òptima
  El càlcul de la descomposició QR mitjançant les reflexions de *Householder* té una complexitat de $2N^3M^3$ mentre que el mateix càlcul amb rotacions de *Givens* comporta $3N^3M^3$ operacions de coma flotant.

- No s'han gestionat les matrius iguals a zero no les identitats
  Si s'analitza l'algoritme en profunditat, es poden detectar dos aspectes molt remarcables:

  – A l'equació 4.3 els termes $\Gamma^{0,1}$ i $\Sigma^{1,1}$ no s'utilitzen en el càlcul. Per tant, es podria prescindir de la informació que contenen aplicant directament la transformació de *Householder* corresponent.

  – A l'equació 4.4 es mostren els primers dos components de la matriu resultat Q. Com es pot veure, la majoria dels blocs són iguals a [0] o a la identitat. Sempre que un bloc prengui un dels dos valors, es podria obtenir el resultat sense realitzar cap càlcul.

  Tenint en compte els dos punts anteriors, s'aconsegueix que la complexitat de l'algoritme caigui fins als $2N^3M^3$.

Una execució d'aquest algoritme utilitzant una mida de matriu igual a 4 genera el graf de dependències presentat a la figura 21. Les tasques blaves es corresponen amb les inicialitzacions de les matrius, les blanques amb `qr`, les vermelles amb `dot`, les roses amb `littleQR` i les marrons amb `multiplySingleBlock`.

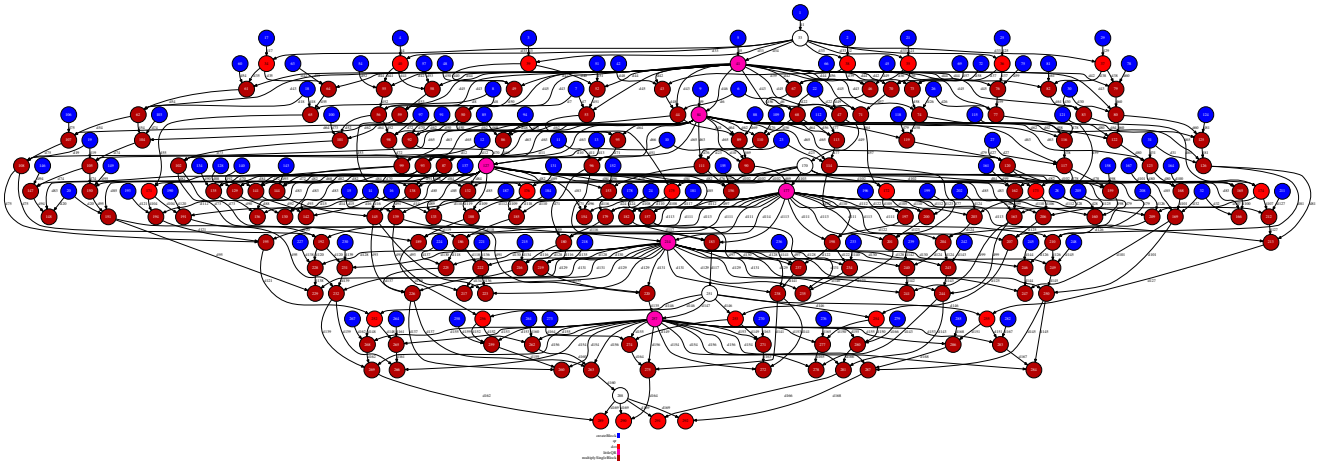

Figura 21 – Graf de dependències corresponent a la factorització QR

Aquest graf mostra que tot i que s'ha de pagar un alt preu pel que a nombre de càlculs es refereix, el grau de paral·lelisme assolit és molt bo.

La figura 22 mostra la traça obtinguda amb una mica de matriu igual a 4. L'anàlisi de la mateixa mostra el que ja es podia intuir en el graf de dependències: l'algoritme té un bon grau de paral·lelisme. Com es pot veure, es confirma el grau de paral·lelisme esperat observant del graf de dependències.

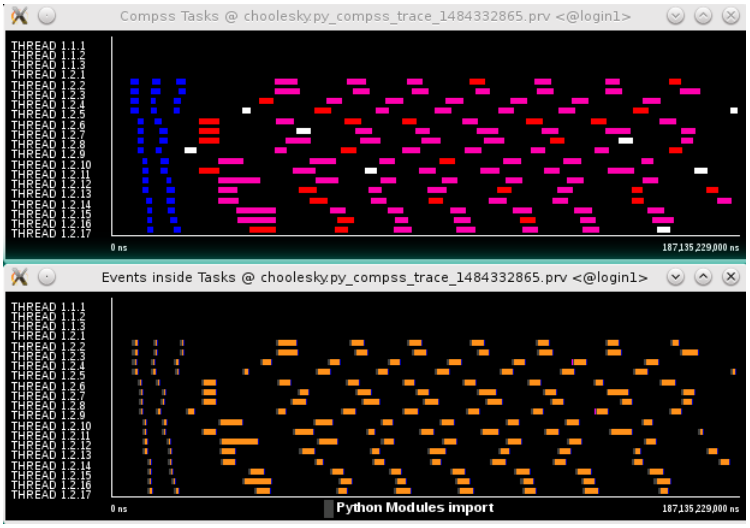

Figura 22 – Vista *Compss Tasks* per a la factorització QR amb mida de matriu 4

4.3 Llibreria matemàtica distribuïda

Com s'ha pogut comprovar, la implementació d'algoritmes per a l'execució d'operacions d'àlgebra lineal no es trivial. Aquest fet provoca que la implementació d'una llibreria distribuïda requereixi un quantitat de treball molt alta, ja que caldria implementar un nombre mínim d'operacions per a fer-la interessant.
Tenint aquest fet en compte es va realitzar un estudi d'alternatives. Es va arribar a la conclusió que la millor opció era aprofitar les llibreries ja disponibles. Concretament, es va pensar en aprofitar les llibreries que donin un millor rendiment, escollint *NumPy* per poder aprofitar les seves crides a *MKL*.
La solució adoptada, mostrada a l'apèndix B, intercepta totes les crides a un determinat mòdul i tots els seus submòduls, podem modificar les crides com es desitgi. D'aquesta manera, es permet cridar les funcions distribuïdes en el cas que aquestes estiguin implementades. En cas contrari, es crida les funcions normals de *NumPy*. Aquest comportament s'aconsegueix simplement canviant l'`import numpy as np` per `import matlib as np`.
L'existència d'aquesta nova implementació fa possible afegir els algoritmes existents de forma gradual garantint a l'usuari que tindrà accés a la llibreria de *NumPy* complerta.

Capítol 5

Implementació dels nous planificadors

Índex

Durant l'execució dels algoritmes matemàtics més complexos, es van detectar certes anomalies en la planificació de les tasques. Tenint aquest fet en compte, es va procedir a l'estudi de la implementació dels planificadors.

5.1 Introducció

En aquesta fase, es va procedir a la comprensió, estudi i posterior modificació del codi actual.
Una vegada entès el funcionament dels mateixos, es va detectar una tipologia clara de planificador. Concretament, aquest document estudia els casos en el que sol es tenen en compte les tasques llestes per executar. Es a dir, ja sigui aquelles lliures de dependències com aquelles l'execució dels predecessors de les quals ja ha acabat.
Tenint en compte que aquest conjunt de planificadors té una mida notable, es va introduir una classe abstracta que servís d'interfície per a les posteriors implementacions. Aquesta classe hereta del planificador original, conservant el màxim nombre possible de funcions ja implementades.
Per a fer més fàcil la interpretació de les traces, s'han executat tots els exemples amb `ComputingUnits=1`.

Finalment, cal tenir en compte el comportament esperat del planificador actual per entendre el procediment seguit a continuació. Primer de tot, es planifiquen les tasques disponibles per a emplenar tots els workers. A partir d'aquí, cada vegada que un recurs queda lliure s'hi intenta col·locar la tasca que capaç d'executar-se en l'espai alliberat que utilitza el màxim nombre de dades presents en aquell worker. Després de detectar certs problemes amb el seu comportament, es va decidir implementar planificadors el més bàsics però robustos alhora per anar complicant la seva política gradualment.
Com es pot veure, l'últim planificador presentat té un comportament que, tot i que més simple que l'exposat al paràgraf anterior, aconsegueix un comportament similar.

5.2 FIFO

En aquesta implementació, quan un determinat worker alliberant recursos per a una nova execució, es planifica la tasca amb l'identificador més baix. L'identificador de les tasques és un enter que s'assigna de forma estrictament creixent a mesura que les tasques s'enregistren.
Aquest planificador ha estat dissenyat per assegurar que un graf de dependències s'explorarà amb la màxima amplitud possible.
Com es pot apreciar a la figura 23, el resultat obtingut és l'esperat. El gradient de colors és perfecte, cosa que indica que la planificació s'ha realitzat en l'ordre esperat. A més a més, es comprova que el grau de paral·lelisme és bo. Es poden apreciar, periòdicament, breus lapses de temps. Aquest fet es desprèn del graf de dependències presentat a la figura 18. Les tasques de tipus `potrf` (en color blanc) creen punts de sincronisme.

5.3 LIFO

En aquest cas, la planificació es realitza triant les tasques amb un identificador el més alt possible.
Considerant el mecanisme d'assignació dels identificadors, aquest planificador tendirà a l'exploració del graf de dependències en profunditat.
Com es pot apreciar a la figura 24, en aquest cas el resultat obtingut també és l'esperat. El gradient de colors es troba invertit en la mesura del possible, es a dir, sense sobrepassar els punts de sincronisme creats per les tasques de tipus `potrf`. Aquest fet demostra que el planificador funciona tenint en compte el criteri introduir però assegurant que cap dependència es viola en cap moment.

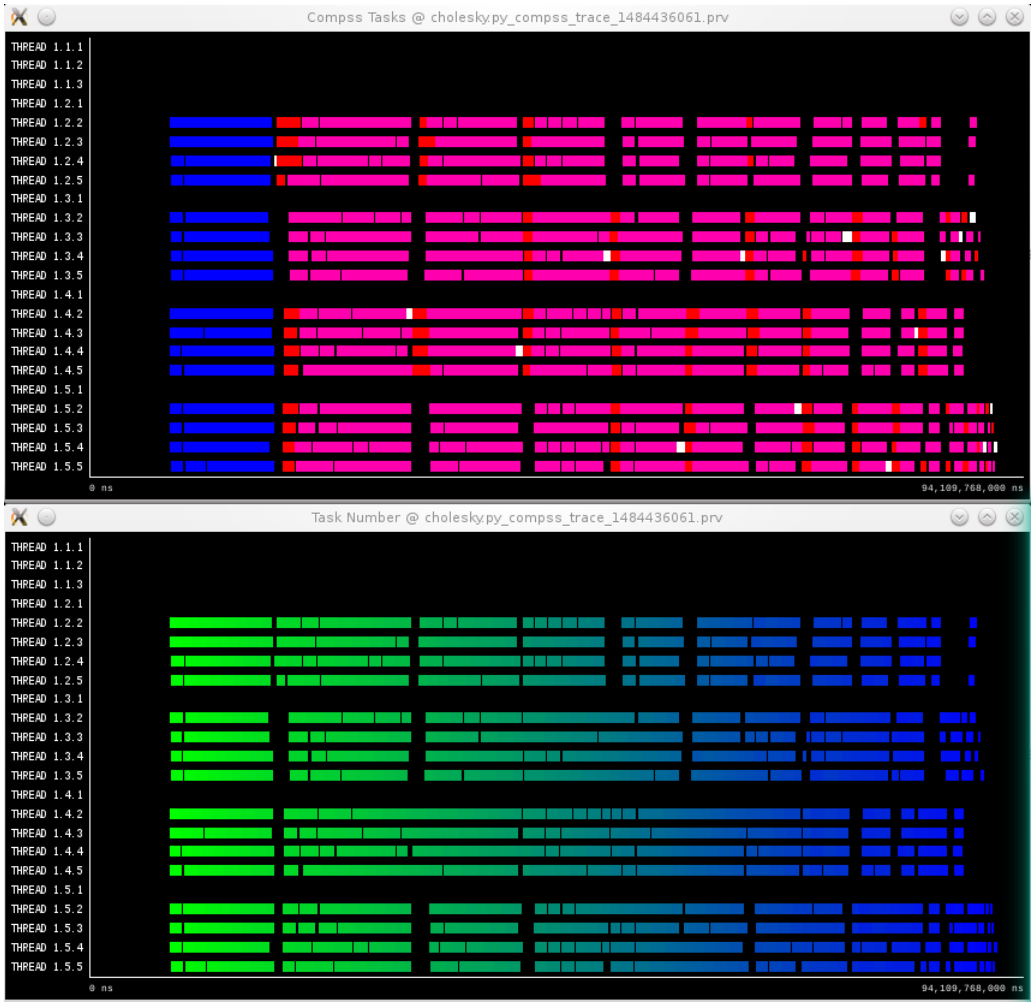

Figura 23 – Vistes *Compss Tasks* i *Task Number* per a la factorització de Cholesky amb mida de matriu 16 amb planificació FIFO

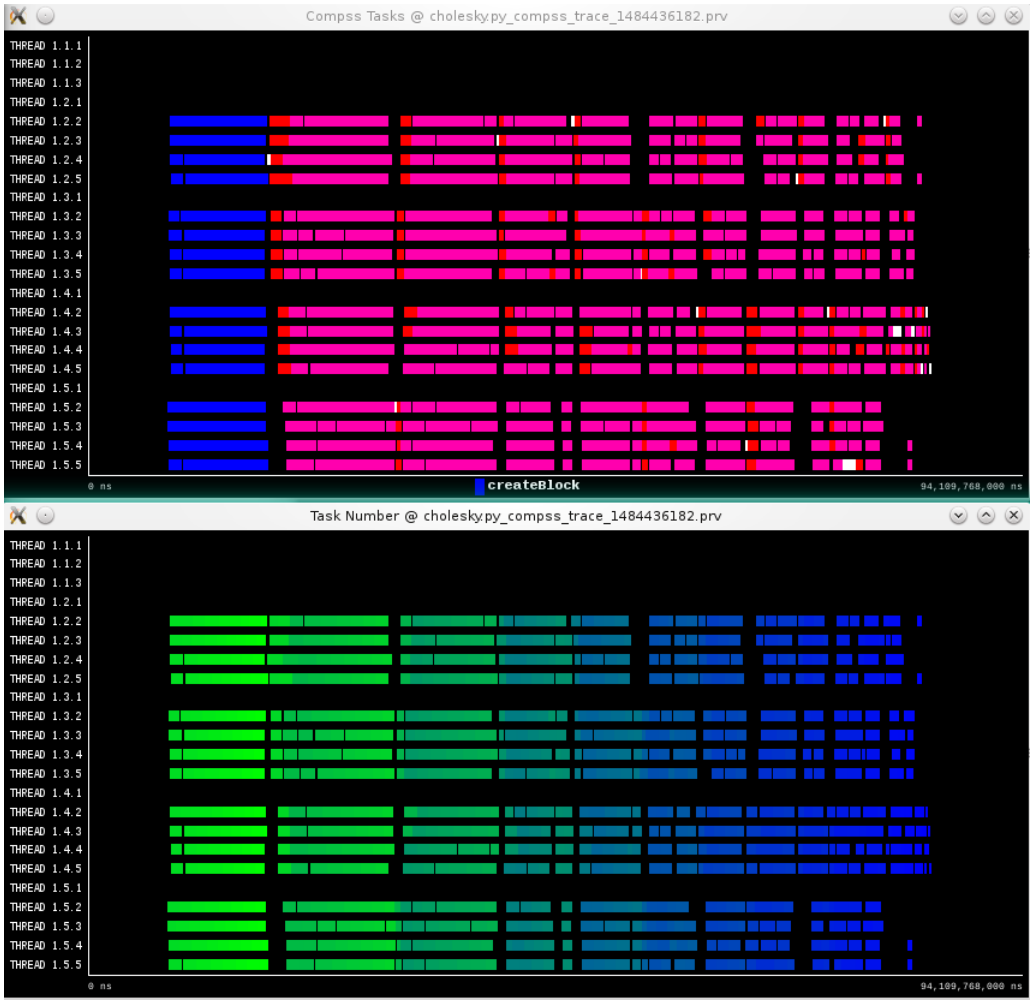

Figura 24 – Vistes *Compss Tasks* i *Task Number* per a la factorització de Cholesky amb mida de matriu 16 amb planificació LIFO

5.4 FIFO amb localitat de dades

Aquest planificador suposa un híbrid entre els dos anteriors.
En un moment de tria lliure, es triaran sempre les tasques amb un identificador més baix. D'aquesta manera, l'exploració del graf es realitzarà en amplitud. No obstant, quan una determinada tasca acaba, es comprova si allibera les dependències d'alguna altra. En cas afirmatiu, enlloc de planificar la tasca amb un identificador més baix, es planifica alguna de les tasques alliberades en el mateix recurs que acaba de quedar lliure. D'aquesta manera, s'obté una exploració per amplitud que, quan pot, aprofita la localitat de les dades d'una tasca que ha acabat en aquell mateix moment.
A la figura 25 es mostren les vistes corresponents a la traça obtinguda en l'execució realitzada per mostrar el comportament d'aquest planificador. Com es pot observar, el resultat obtingut està entre les dues implementacions anteriors. Les inversions en el gradient no són tan clares com en el cas *LIFO* però la seva existència es pot apreciar clarament. Aquestes es corresponen amb els casos en els que es decideix planificar les tasques pel criteri de localitat de dades.

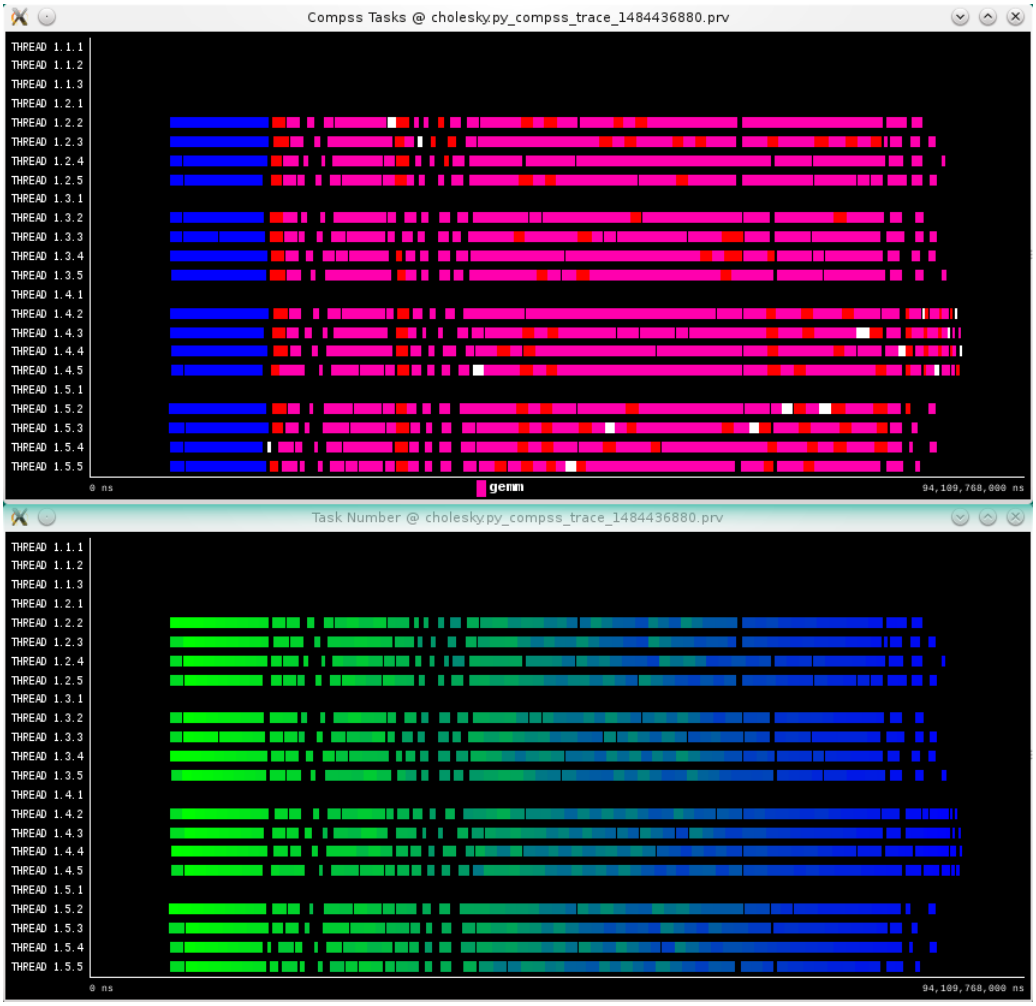

Figura 25 – Vistes *Compss Tasks* i *Task Number* per a la factorització de Cholesky amb mida de matriu 16 amb planificació FIFO amb localitat de dades

Capítol 6

Conclusions

En aquest projecte s'ha abordat una temàtica dintre del domini de la recerca relacionada amb un tema d'actualitat. En aquesta secció es resumeixen els resultats obtinguts, els aspectes a millorar i els possibles treballs futurs derivats del treball realitzat.

6.1 Resultats obtinguts

Els resultats obtinguts més destacables són els següents:

1. Augment del rendiment de la multiplicació de matrius en un factor superior a 40 (de 4.75 GFlops/s a 203 GFlops/s en còmputs amb un sol worker)

2. Implementació de dos nous algoritmes d'àlgebra lineal per a realitzar el càlcul de forma distribuïda

3. Implementació d'una llibreria matemàtica per a incloure-hi els algoritmes distribuïts de forma gradual

4. Introducció de 3 noves polítiques de planificació a *COMPSs*

6.2 Aspectes a millorar

Els aspectes a millorar més importants són els següents:

1. Integració de forma genèrica de l'afinitat dels threads per socket considerant qualsevol arquitectura Com s'ha explicat a l'apartat corresponent, la implementació actual només funciona a *MareNostrum* i caldria modificar-la per executar-la en un entorn diferent

2. Revisió del codi dels planificadors La nova implementació dels planificadors es va fer pensant en els exemples presentats en aquest document. Caldrà revisar aquesta implementació per fer-la més general i fer possible la introducció de polítiques més complexes.

6.3 Treball futur

Com a treball futur, es consideren els següents punts:

1. Prova en profunditat del rendiment de les factoritzacions de *Cholesky* i *QR*

2. Implementació del contingut de la llibreria matemàtica

3. Millora de les classes abstractes del planificador per facilitar la implementació de polítiques més complicades

4. Implementació d'un planificador per a minimitzar el nombre de transferències

5. Millora de la multiplicació per blocs per evitar multiplicar els blocs que siguin igual a zero o la identitat

Apèndix A

Paraver

Paraver[28] es una eina de visualització i anàlisi basada en traces que per a l'estudi de les execucions en sistemes distribuïts. S'ha cregut apropiat la inclusió en aquest document d'un apèndix que n'expliqui el funcionament bàsic donada la seva importància en l'anàlisi de l'execució de la multiplicació de matrius.
Aquest programa funciona a base de vistes. Cada vista proporciona una determinada informació relacionada amb l'execució del programa.

A.1 Descripció general de les vistes

A la figura 26 es mostra una vista general del programa *Paraver*. Concretament, a la part esquerra (figura 27) es mostra un índex dels threads instrumentats pel programa al llarg d'una determinada execució. El segon índex fa referencia al node instrumentat. El tercer índex indica la unitat de còmput que realitza una determinada tasca. A la part central (figura 28) es mostra la informació corresponent a la vista seleccionada.
En aquest exemple concret, l'execució disposa de cinc nodes: 1 master i 4 workers. Cada worker disposa de 4 *threads* de còmput i un de gestió. L'escala de colors present al master i als *threads* de gestió de cada worker fan referència a informacions relacionades amb la planificació i transferència de dades. Les variacions mostrades als *threads* de còmput dels workers mostren el temps d'execució de les tasques. Concretament, en aquest cas es mostra la traça obtinguda en l'execució d'una multiplicació entre dos matrius de 4 blocs x 4 blocs.

Figura 26 – Vista general d'una vista del programa Paraver

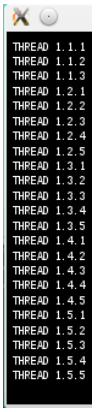

Figura 27 – Conjunt de threads instrumentats en una determinada traça

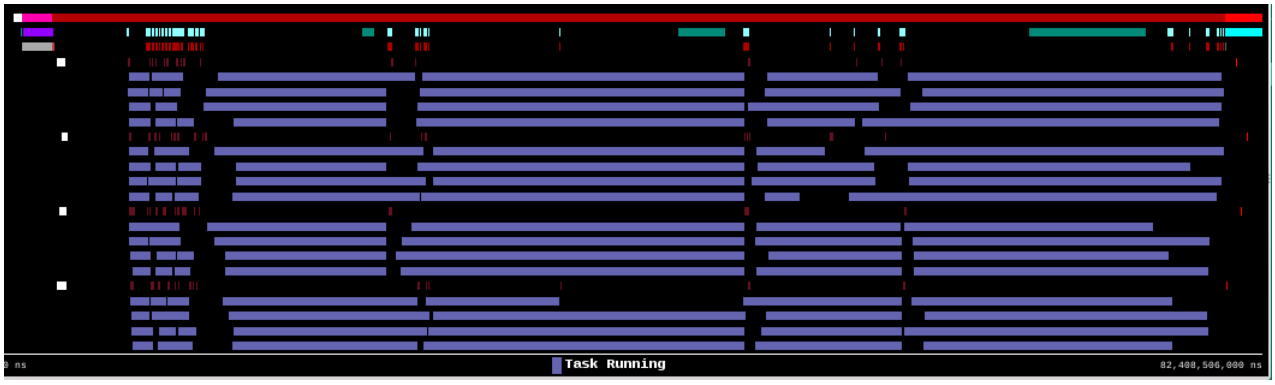

Figura 28 – Informació lligada a una determinada vista

A.2 Vistes utilitzades

A continuació es fa una breu descripció de les vistes utilitzades en el projecte. S'utilitzarà la traça presentada a la secció anterior. Concretament, la vista que s'ha presentat és *Compss Runtime*. Es considera que ja s'ha descrit suficientment.

Compss Tasks

L'objectiu d'aquesta tasca es mostrar l'execució de les diferents tasques. Cada tipologia de tasca és d'un color diferent. Concretament, a la vista presentada a la figura 29 les inicialitzacions dels blocs es mostren en blau mentre que les tasques de multiplicació es mostren en blanc.

Task Number

L'objectiu d'aquesta vista es veure l'ordre d'execució de les tasques. Les tasques que han estat registrades abans es mostren en color verd clar. Les últimes tasques en ser introduïdes en el sistema es mostren en blau fosc. Les tasques intermitges es mostren amb colors degradats entre les dues tonalitats esmentades. A la figura 30 es mostra un exemple d'aquesta vista. Cal remarcar que, si es desitja, es pot marcar l'inici i el final de les tasques amb una bandera verda. Si dues tasques consecutives es troben molt juntes, es podria pensar que una determinada línia fa referència a una sola traça quan, en realitat, n'engloba l'execució de més d'una.

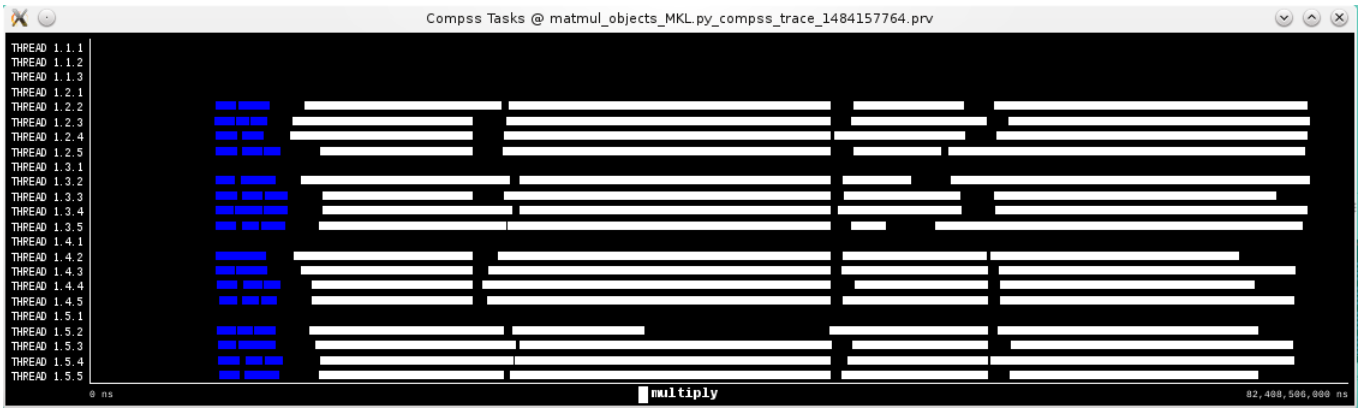

Figura 29 – Exemple de vista Compss Tasks

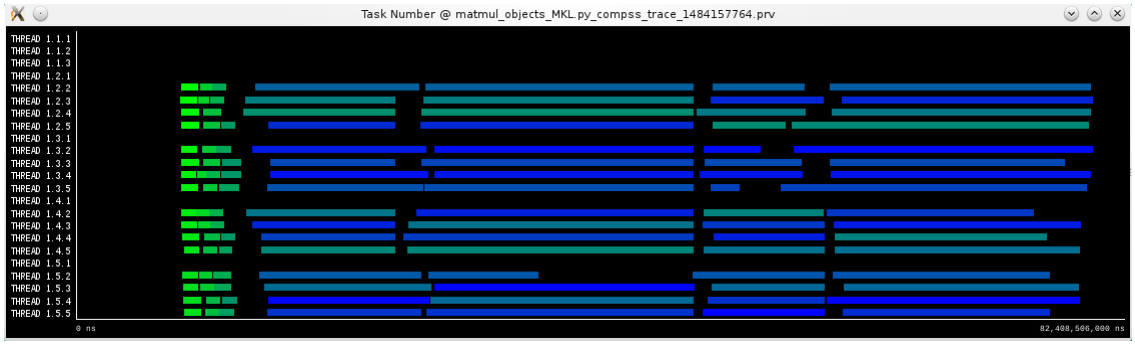

Figura 30 – Exemple de vista Compss Number

Events Inside Tasks

L'objectiu d'aquesta vista es comprovar en que s'ha invertit el temps d'execució dintre d'una tasca. Aquesta vista és pròpia de *PyCOMPSs* i a la figura 31 es mostra la vista general amb l'execució que ha servit de referència durant tot l'apèndix.
El més important en aquest cas és adonar-se que en taronja es mostra el temps d'execució del codi de la tasca. El colors a l'esquerra i a la dreta dels fragments taronges indiquen el temps que s'ha invertit en aixecar l'intèrpret de *Python*, la transferència de dades i altres operacions realitzades per *COMPSs*. A la figura 32 es mostren de costat dues tasques, una d'inicialització i una de multiplicació. Com es pot veure, l'overhead de les tasques d'inicialització es desproporcionat respecte al temps de còmput. En canvi, a les tasques de multiplicació, el temps de còmput es molt més gran que el temps invertit en realitzar operacions auxiliars.

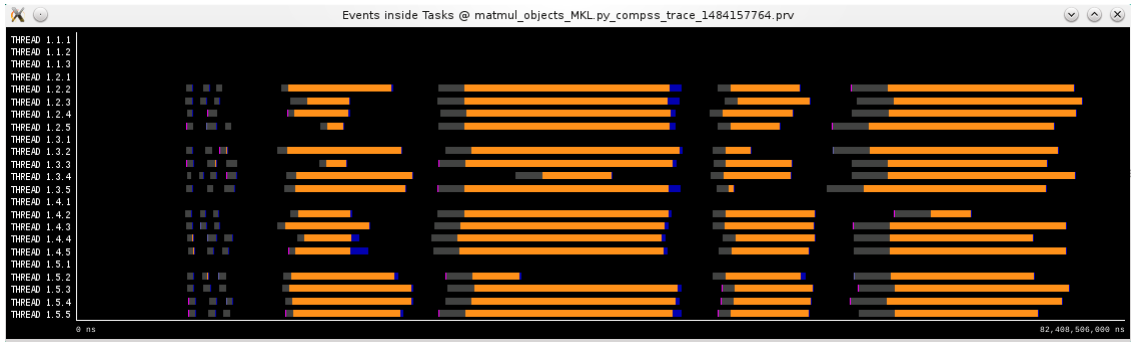

Figura 31 – Exemple de vista Events Inside Tasks

Figura 32 – Amplicació de la vista Events Inside Tasks

Apèndix B

Implementació de matlib

La figura 33 mostra la implementació del mòdul *matlib*.

```
import sys
import numpy

class Wrapper(object):
  def __init__(self, wrapped_module, end_module):
    self.wrapped_module = wrapped_module
    self.end_module = end_module

  def __getattribute__(self, item):
    return object.__getattribute__(self,item)

  def __getattr__(self, attr):
    try:
      orig_attr = self.wrapped_module.__getattribute__(attr)
    except (AttributeError,TypeError) as e:
      print("The following function or module doesn't exist %s" % e)
    else:
      if callable(orig_attr):
        def hooked(*args, **kwargs):
          self.pre(*args, **kwargs)
          result = orig_attr(*args, **kwargs)
          self.post()
          return result
        return hooked
      else:
        try:
          from types import ModuleType
          import sys
          if type(orig_attr) is ModuleType:
            orig = orig_attr.__name__
            if self.wrapped_module.__name__ in orig:
              dest = orig_attr.__name__.replace(self.wrapped_module.__name__,
    self.end_module)
              subModule = sys.modules[orig_attr.__name__]
              wrap = type(self)(subModule, dest)
              sys.modules[dest] = wrap
              return wrap
          return orig_attr
        except (AttributeError,TypeError) as e:
          print("%s error creating wrapper to module %s" % (e, orig_attr.
    __name__))
          return self.handleError()
    return getattr(self,"handleError")

  def pre(self,*args,**kwargs):
    print(">> pre")

  def post(self):
    print("<< post")

  def handleError(self):
    print("Error handler")

sys.modules[__name__] = Wrapper(numpy, __name__)
```

Figura 33 – Mòdul principal de la llibreria matlib

Bibliografia

[1] O. Shacham K. Olukotun L.Hammond C.Batten M. Horowitz, F. Labonte. 35 years of microprocessor trend data, 2010.

[2] Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[3] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[4] Tiobe index. http://www.tiobe.com/tiobe-index/. Lloc web visitat el 27-09-2016.

[5] Rohit Chandra. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[6] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[7] Apache Hadoop. Pàgina oficial del projecte. http://hadoop.apache.org/.

[8] Apache Hadoop. Pàgina amb les releases publicades. http://hadoop.apache.org/releases.html.

[9] Apache Spark. Pàgina oficial del projecte. http://spark.apache.org/.

[10] Barcelona Supercomputing Center. Grup de workflows and distributed computing. https://www.bsc.es/discover-bsc/organisation/scientific-structure/workflows-and-distributed-computing/team-people.

[11] OmpSs. Pàgina oficial del projecte. https://pm.bsc.es/ompss.

[12] Intel Corporation. *Math Kernel Library documentation*, 2016. MKL version 11.3.

[13] OpenMP. Pàgina oficial del projecte. http://www.openmp.org/.

[14] IHS Markit. Global semiconductor market. http://press.ihs.com/press-release/technology/global-semiconductor-market-slumps-2015-ihs-says.

[15] NumPy. Pàgina oficial del projecte. http://www.numpy.org/.

[16] Intel Corporation. Numpy/scipy with intel mkl and intel compilers. https://software.intel.com/en-us/articles/numpyscipy-with-intel-mkl.

[17] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[18] Howard Eves. *Elementary Matrix Theory*. Dover Publication, 1980.

[19] Intel Corporation. Especificacions de la família intel xeon e5-2600. http://www.intel.com/content/dam/support/us/en/documents/processors/xeon/sb/xeon_E5-2600.pdf.

[20] Marc Monguió. Sous mitjans d'un enginyer informàtic. http://blog.jobsbcn.com/index.php/2015/10/01/guia-salarial-para-programadores-barcelona-2015/?lang=es.

[21] Amazon USA. Preu dell latitude e7440 i7. https://www.amazon.com/Dell-Latitude-E7440-Ultrabook-Professional/dp/B00MDN3770.

[22] Intel Developer Zone. Need help to interpret result of mkl hpl mp linpack. https://software.intel.com/en-us/forums/intel-math-kernel-library/topic/605789.

[23] Documentació oficial del compilador C++ d'Intel. Thread affinity interface (linux* and windows*). https://software.intel.com/en-us/node/522691#KMP_AFFINITY_ENVIRONMENT_VARIABLE.

[24] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel*. "O'Reilly Media, Inc.", 2005.

[25] Jack J. Dongarra, Lain S. Duff, Danny C. Sorensen, and Henk A. Van der Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[26] Miguel Grau Sánchez and Miquel Noguera Batlle. Càlcul numèric. 2001.

[27] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert A. Van De Geijn, and Field G. Van Zee. *Scheduling of QR factorization algorithms on SMP and multi-core architectures*. 2008.

[28] Paraver. Pàgina oficial del projecte. https://www.bsc.es/discover-bsc/organisation/scientific-structure/performance-tools.