eman ta zabal zazu

Universidad del País Vasco | Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Informatika Ingeniaritzako Gradua

Konputagailuen Ingeniaritza

Gradu Amaierako Lana

Argi-barreiaketaren kode zientifikoaren paralelizazioa

Egilea

Borja Leandro Barrantes

2020

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Informatika Ingeniaritzako Gradua

Konputagailuen Ingeniaritza

Gradu Amaierako Lana

Argi-barreiaketaren kode zientifikoaren paralelizazioa

Egilea

Borja Leandro Barrantes

Zuzendaria(k)

Olatz Arregui Uriarte / Agustin Arruabarrena Frutos

Laburpena

Proiektu honetan fisika arloko argi barreiaketaren inguruko kode bat paralelizatu da. Horretarako hainbat paralelizazio teknika eta irizpide erabili izan dira paralelizazio maila handiena lortzeko asmoarekin.

Proiektuan garatuko den aplikazioa simulatzaile bat da. Simulatzaile honen eginkizuna argiaren barreiaketa simulatzea da. Horretarako, ekuazio ugari ebazten dira, eta aurretik ezarritako hainbat irizpideen arabera, simulatzaileak hainbat kalkulu egiten ditu.

Simulatzaile hau konplexua da eta kalkulu astunak egiten ditu. Beraz, simulatzaileak denbora asko behar du kalkulu guztiak egiteko. Ondorioz, denbora asko behar denez, tamaina handiko simulazioak egitea ez da bideragarria kodea ez bada paralelizatzen.

Arazo honi konponbidea emateko, aplikazio honi konputazio paralaleloa aplikatzea erabaki da. Konputazio paraleloaren helburu nagusia, exekuzio-denborak jaistea da, eta arazo honi konponbidea emateko hainbat teknika eta tresna erabiltzen dira. Proiektuan erabiltzen diren teknikei dagokionez bi erabiltzen dira. Alde batetik, simulatzaileko kalkulu astunen paralelizazioa egingo da, honi "barne paralelizazioa"deituko zaio. Beste aldetik, simulatzaileko begizta nagusia paralelizatuko da, honi kanpo paralelizazioa"deituko zaio.

Palelizazioan erabiltzen diren tekniken artean, proiektu honetan erabiliko direnak bi dira: **OpenMP** eta **OpenMPI**. **OpenMP** memoria partekatuan oinarritutako paralelizazioko tresna bat da. **OpenMPI** aldiz, memoria banatuan oinarritzen da. Horrez gain, kontzeptu bezala **CUDA**-ko bertsio bat egitea proposatu da.

Azaldutako tresna eta teknika erabiliz, hainbat proba egingo dira, eta proba hauetako exekuzio denborekin konparaketak egingo dira ikusteko zein teknika eta zein tresna diren egokienak simulatzaile honi begira, exekuzio denbora gutxitzeko helburuarekin.

Gaien aurkibidea

Laburpena **i**

Gaien aurkibidea **iii**

Irudien aurkibidea **vii**

Taulen aurkibidea **ix**

1 Sarrera **1**

1.1 Motibazioa 1

2 Proiektuaren helburuen Dokumentua **3**

2.1 Proiektuaren deskribapena 3

2.2 Plangintza 4

2.2.1 Proiektuaren atalak 4

2.2.2 LDE diagrama 5

2.2.3 Denbora-estimazioak 5

2.3 Lan metodologia 6

2.4 Arriskuak eta neurriak 6

3 Argiaren barreiaketa **9**

4 Konputazio paraleloa **11**

4.1 Flynn-en sailkapena 12

4.1.1 SISD 12

4.1.2 SIMD 13

4.1.3 MIMD 13

4.2 Eraginkortasuna 15

4.2.1 Helburua 15

4.2.2 Arazoak 16

5 Hardwarearen analisia **19**

5.1 Zen 2 20

5.1.1 Koma higikorreko unitatea 22

5.1.2 Zenbaki osokoen unitatea 22

5.1.3 Cache memoria 22

5.1.4 Infinity Fabric 23

6 Aplikazioaren Paralelizazioa **25**

6.0.1 Simulatzailearen egitura 25

6.0.2 Esperimentazioa 30

6.1 OpenMP 31

6.1.1 Paralelizazioarekin hasi aurretik egin beharrekoa 35

6.1.2 Paralelizazioko irizpideak 36

6.1.3 Kodearen OpenMP-ko lehen bertsioa 37

6.1.4 Kodearen OpenMP-ko bigarren bertsioa 40

6.1.5 Proiektuaren OpenMP-ko hirugarren bertsioa 42

6.1.6 Bertsioen arteko konparaketak 45

6.2 OpenMPI 48

6.2.1 OpenMP eta OpenMPI-ren arteko konparaketa 49

6.3 CUDA 51

7 Ondorioak **53**

7.1 Proiektuaren ondorioak 53

7.2 Ondorio pertsonalak 54

7.3 Etorkizunerako lana 55

Eranskinak

Bibliografia **67**

Irudien aurkibidea

2.1 LDE diagrama 5

4.1 Flynnen sailkapena 12
4.2 *Single Instruction Single Data* arkitektura 13
4.3 *Single Instruction Multiple Data* arkitektura 13
4.4 *Multiple Instruction Multiple Data* arkitektura 14
4.5 Memoria partekatuko sistema 14
4.6 Memoria banatuko sistema 15
4.7 Amdahl-en legea 17

5.1 Ryzen 3900x nukleoen erloju-maiztasuna 20
5.2 Zen 2 txip baten eta nukleoen eskemak 21
5.3 Zen 2 arkitekturaren eskema 21
5.4 Zen 2 arkitekturako koma higikorreko unitatearen eskema 22
5.5 Zen 2 arkitekturako cachearen eskema 23

6.1 *material_list* fitxategiaren edukia 27
6.2 *simul_parameters* fitxategiaren edukia 28
6.3 *static* banaketa motaren eskema, chunk kopuru ezberdinekin 34
6.4 *dynamic* banaketa motaren eskema 34

6.5 *guided* banaketa motaren eskema 35
6.6 Azelerazio-faktorea hari kopuruekiko 39
6.7 Eraginkortasuna hari kopuruekiko 39
6.8 200 iterazioko grafikak 43
6.9 500 iterazioko grafikak 43
6.10 1000 iterazioko grafikak 43
6.11 Azelerazio-faktorea Hirugarren bertsioa (500 iterazio) 45
6.12 Eraginkortasuna Hirugarren bertsioa (500 iterazio) 46
6.13 Azelerazio-faktorea hiru bertsioak (200 iterazio) 47
6.14 Eraginkortasuna hiru bertsioak (200 iterazio) 47
6.15 Azelerazio-faktorea OpenMPI VS OpenMP (500 iterazio) 50
6.16 Eraginkortasuna OpenMPI VS OpenMP (500 iterazio) 51

Taulen aurkibidea

2.1 Proiektuaren denbora-estimazioa 6

5.1 Makinaren ezaugarriak 19

6.1 OpenMP-ko lehen bertsioaren 200 iterazioko exekuzio denboren taula . . 38

6.2 *schedule* motak 24 hariekin lortutako exekuzio denboren taula 41

6.3 OpenMP-ko bigarren bertsioaren exekuzio denboren taula 42

6.4 OpenMP-ko bigarren bertsioaren aelerazio-faktore eta eraginkortasuna-ren taula 42

6.5 OpenMP-ko Hirugarren bertsioaren 500 iterazioko exekuzio denboren tau-la 45

6.6 OpenMPI-ko 500 iterazioko exekuzio denboren taula 49

1. KAPITULUA

Sarrera

1.1 Motibazioa

Konputazioaren hasieratik, konputagailu-software guztia seriekoa zen. Kode guztiak, bata bestearen atzetik exekutatzen ziren aginduz idatzita zeuden. Agindu hauek guztiek konputagailuaren Prozesatzeko Unitate Zentralean (PUZ) ordenan exekutatzen ziren.

Denborak aurrera egin zuen ahala, arazoei konponbide berriak ematen saiatzen zen, eta arazo horietako bat denbora zen. Ordura arteko programak ez ziren oso astunak, garai hartako makinak ez zirelako gai programa oso astunak prozesatzeko, honek emango zuen denbora oso altua zelako. Beraz, arazo honi konponbide bat emateko paralelizazio-teknikak eta tresnak sortu ziren. Hauen helburu nagusia, hardwareko baliabideak guztiz aprobetxatzea da, eginkizun ezberdinak aldi berean egiten. Modu honetan, programen jatorrizko portaera mantenduz, denbora murrizten da.

Konputatzio paraleloan, seriekoan ez bezala, kodearen zati ezberdinak aldi berean exekutatzen dira. Horretarako, kodearen analisia egiten da aurretik eta ebatzi nahi den arazoa hainbat zati txikiagotan banatzen da. Ondoren, paralelizatutako kodea exekutatzen da; horretarako prestatutako prozesadore mota desberdin duten makinak daude: hainbat prozesadore dituen konputagailu bat, konputagailuez osatutako sare bat, eta hauen artean egin daitezkeen konbinaketak. Orokorrean, konputazio paraleloa zientzia eta ingeniaritza arloko simulazioak egiteko erabili izan da.

Gaur egun, teknologiaren aurrerapenei esker, edozein etxetako konputagailuetan nukleo

anitzeko prozesadoreak daude. Beraz, konputazio paraleloa edozeinen eskura dago eta beraz, edozein pertsonak erabiltzen dituen programak modu paraleloan erabili daitezke.

Ikerketa-arloari eta enpresa-munduari dagokionez, konputazio paraleloak abantaila izugarria ekarri baitu, simulazio handiak exekutatzeko gaitasuna ekarri du, hura erabiliz murrizten den exekuzio denboragatik. Simulazioa hainbat zati independentetan banatzen da, eta ondoren modu paraleloan exekutatzen da; horrela eginda, denbora asko aurrezten da. Enpresa-munduari dagokionez, haiek erabiltzen edo eskaintzen dituzten zerbitzuak aldi berean exekuta ditzakete, horretarako hainbat makinez osaturiko clusterrak dituztelako.

Proiektu honetan, konputazio paraleloa ikerketa-arloko kode batean erabili da, nukleo ugariko prozesadorea duen makina eskaintzen duen paralelizazio maila aprobetxatzeko. Horretarako, jatorrizko kodea aztertu da, eta konputazio paraleloko hainbat prozedura erabili dira proiektuaren helburua lortzeko.

2. KAPITULUA

Proiektuaren helburuen Dokumentua

Atal honetan, proiektu honen inguruko informazioa azalduko da: proiektuaren deskribapena, egindako planifikazioa, banatutako atazak eta bestelakoak.

2.1 Proiektuaren deskribapena

Proiektu honen helburua fisika arloko kode baten paralelizazioa da. Proiektua aurrera eramateko, EHUko Informatika Fakultatea eta DIPC zentroaren arteko kolaborazioa gauzatu da. Jatorrizko proiektua, Eris izenekoa, fisika arloko fenomeno baten simulazioa da: *Light Scattering* edo argiaren barreiaketa.

Simulatzaileak ondorengo funtzionamendua du. Sarrera-fitxategi batetik konfigurazio-parametroak irakurtzen ditu. Ondoren, hainbat kalkulu egingo dira anplitude (λ) izeneko parametroaren inguruan. Anplitudea, balio minimo (λ_{min}) batetik balio maximo (λ_{max}) batera arte igotzen joaten da kalkuluak egin ahala. Bukaeran, simulazioaren emaitzak irteera-fitxategi batean uzten dira.

Simulatzailean egiten diren kalkuluak konplexuak dira eta beraz, probak egiterako garaian arazo garrantzitsuenetako bat denbora da. Arazo hau dela eta, simulatzailearekin egindako proben tamaina asko murriztu behar da, ez delako bideragarria denbora handiko exekuzioak egitea, batzuetan emaitzak izateko ez dagoelako hainbeste denbora edota exekutatu beharreko makinetan exekuzio-denborako murriztapenak daudelako. Beraz, hori kontuan izanda, simulatzaileak kalkuluak egiteko behar duen denbora murrizteko,

konputazio paraleloko kontzeptu, teknika eta tresnak kontuan izango dira. Modu honetan, lortutako denbora-murriztapenarekin, simulatzailearekin datu-tamaina handiagoekin egin daitezke probak, denbora berean kalkulu gehiago egiten direlako. Bestalde, tamaina faktore nagusi bat ez bada, aldi berean simulazio bat baino gehiago exekuta daitezke datu-tamaina txikiagokoak.

Simulazioaren jatorrizko kodea **C++** lengoaian idatzita dago. Proiektuan zehar, hainbat aldaketa egin dira jatorrizko kodean, berezko funtzionalitatea mantenduz. Jatorrizko kodea paralelizatzeko, honen funtzionamendua ulertu behar da. Kodean hainbat klase erabiltzen dira eta haien esanahia aztertu behar da, programa nagusian klase horietan definitutako funtzioak erabiltzen direlako. Behin kodea aztertuta, konputazio paraleloko oinarriak eta teknikak aztertu behar ditugu, ondoren zein paralelizazio-tresnak erabiliko diren erabaki ahal izateko. Azkenik, behin kodea paralelizatu dugunean, jatorrizko serieko eta proiektuan zehar garatutako bertsioen emaitzak konparatuko ditugu hainbat irizpideren arabera, eta emaitza horiekin ondorio batera iritsiko gara.

2.2 Plangintza

2.2.1 Proiektuaren atalak

Proiektu honek hainbat atal ditu.

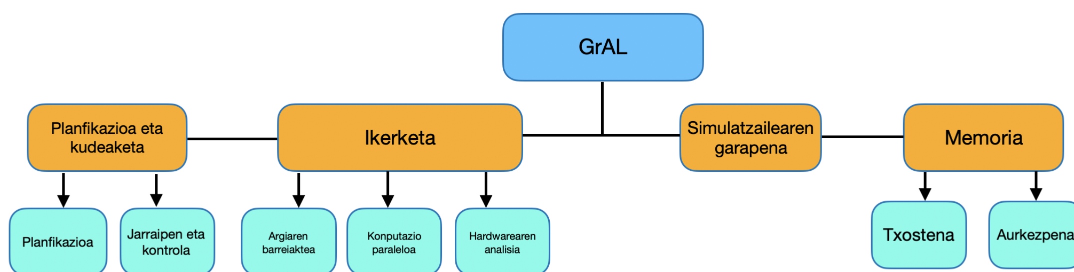
- Planifikazioa eta Kudeaketa
  - **Planfikazioa:** GrAL-ren garapenerako plangintza egingo da, eta bertan, atal bakoitzerako denbora-dedikazioa finkatzen da. Horrez gain, proiektuan zehar gerta daitezkeen arriskuak aztertuko dira eta haien eragina gutxitzeko hainbat neurri hartzea erabakiko da.
  - **Jarraipena eta kontrola:** Proiektuan zehar egindako aurrerapenak zuzendariekin eztabaidatuko dira bileretan. Beste aldetik, DIPC-ko zientzialariarekin proiektuaren nondik norakoak, emaitzak eta bestelako gehigarri eta hobekuntzei buruz eztabaidatuko dira bileren bitartez.
- Ikerketa

  - **Argiaren barreiaketa:** Nahiz eta proiektu hau informatika arlokoa izan, beharrezkoa da jatorrizko kodearen eginkizuna ulertzea. Horretarako, argiaren barreiaketaren oinarria jakitea komenigarria da.
  - **Konputazio paraleloa:** Konputazio paraleloko oinarria ezagutzea ezinbestekoa da proiektu hau aurrera eramateko. Beraz, honi buruz ikertuko da.
  - **Hardwarearen analisia:** Simulazioa exekutatzen duten ordenagailuen hardwareari buruz analisi sakona egingo da.

- Garapena
  - **Garatutako simulatzailea:** Jatorrizko simulatzailea hartuta, proiektuan zehar garatutako simulatzailearen bertsio berriak konputazio paraleloko hainbat ereduetan oinarritutako tresnak erabiliz.

- Memoria
  - **Txostena:** Proiektuari buruzko informazioa azaltzen duen dokumentua.
  - **Aurkezpena:** GrAL-ren defentsarako proiektua laburbiltzen duen aurkezpena

2.2.2 LDE diagrama



2.1 Irudia: LDE diagrama

2.2.3 Denbora-estimazioak

Proiektuaren atal bakoitza garatzeko aurreikusitako denbora ondorengo 2.1 taulan ikus daiteke.

Atalak	Denbora estimazioa
Planifikazioa	7 h
Jarraipena eta kontrola	10 h
Argiaren barreiaketa	10 h
Konputazio paraleloa	15 h
Hardwarearen analisia	12 h
Garapena	140 h
Txostena	75 h
Aurkezpena	15 h
Proiektua	**300 h**

2.1 Taula: Proiektuaren denbora-estimazioa

2.3 Lan metodologia

Proiektuaren garapen osoa etxetik egingo da, **Covid-19** pandemia dela eta. Horretarako, etxeko ordenagailua erabiliko da. Ordenagailu honek simulazioak exekutatzeko *hadware* egokia du; beraz, ez da arazorik izan simulazioak exekutatzeko. Simulatzailearen kodea **Atom** *IDE*-an idatzi da eta simulazioak **Kubuntu** sistema eragile batean exekutatu dira.

Covid-19 dela eta, zuzendariekin eta zientzialariarekin izandako bilerak **BB Collaborate** eta **Skype** plataformen bitartez egin dira. Bilerak, bi astean behin egin dira bai zuzendariekin bai eta zientzialariarekin ere. Horrez gain, posta elektronikoa ere erabili da komunikatzeko.

2.4 Arriskuak eta neurriak

Proiektu honen hasieratik, garapenean zehar ager daitezkeen arriskuen analisia egin da. Behin arriskuak identifikatzen eta aztertzen direnean, hauei aurre egiteko neurriak erabaki dira.

- **Proiektuaren garapenean gerta daitezkeen arriskuak:** Proiektuaren hasierako ikuspegia aldatzen joaten da proiektua garatzen den bitartean. Horrez gain, garapenean zehar arazoak gerta daitezke, batez ere kode ezezagun bat moldatu behar denean. Arazoak gertatzen direnean, proiektuaren norabidea galtzeko arriskua dago. Beraz, arazoak ez gertatzeko edo haien eragina gutxitzeko, proiektuan parte hartzen dutenen arteko komunikazioa izatea garrantzitsua da. Horretarako, jatorrizko kodea ulertzeko arazoak izan direnean, **DIPC**-ko zientzialariarekin bilerak adostu dira.

Bestetik, proiektuaren ikuspegia aldatzen bada, zuzendariekin bilera bat adostuko da hura eztabaidatzeko.

- **Aktiboak sor ditzaketen arriskuak:** Proiektuak garrantzi handiko hainbat aktibo ditu. Alde batetik, proiektua garatzen den ordenagailua dugu. Ordenagailu honetan, kodea idazteaz gain, simulazioak bertan exekutatuko dira ere. Beraz, bai *hardware* aldetik bai *software* aldetik, edozer gertatuz gero proiektuan eragin handia izango luke. Horregatik, hau ez gertatzeko, ordenagailuaren osagaien monitorizazioa egingo da, hauek modu egokian funtzionatzen ari diren konprobatzeko, batez ere *PUZ*a. Honek, simulazioaren exekuzio garaian temperatura altua izaten du eta beraz, honen monitorizazioa beharrezkoa da. *Software* aldetik, sistema eragilea eguneratuta izatea komeni da, *BIOS*aren bertsioarekin batera. Horrez gain, Simulaziorako beharrezkoak diren paketeak instalatuta eta eguneratuta izatea beharrezkoa da, segurtasuna eta eraginkortasunaren ikuspuntutik.

  Proiektuko kodea, memoria eta bestelako fitxategiak babesteko hainbat neurri hartu dira. Alde batetik, informazio guztia *ext4* fitxategi sistemak eskaintzen duen zifratua erabili da ordenagailuaren biltegiratze unitatean. Modu honetan, unitatea lapurtzen badute, edukia babestuta egongo litzateke. Beste aldetik, proiektuarekin zerikusia duten fitxategi guztien kopiak egin dira beste biltegiratze unitate eta *cloud* motako zerbitzuetan.

3. KAPITULUA

Argiaren barreiaketa

Argiaren portaeran, teoria korpuskular edo ondularrek parte hartu dute Newtonen garaitik egon den eztabaida handian. Argumentu hau portaera korpuskularraren alde egiten duen arren, argiaren difrakzio eta interferentzien gainean egindako esperimentu ugariek dudan jarri zuten teoria hau. Argiaren portaera duala zuen, hau da, batzutan partikula baten portaera zuen, eta beste batzutan uhin batena. XX. mendearen hasieran mekanika kuantikoaren helduerak, eztabaida honi amaiera eman zion, egiaztatuz, argiaren portaera duala zela, hau da, argia uhin baten eta partikula baten portaera duela. Mekanika kuantikoak, objektuak deskribatzen dituenak, hala nola; molekulak, atomoak, elektroiak, supereroaleak, super-fluxuak, etab., Plancken eskalan, historian gehien egiaztatu den teoria izan da, gaur egun arte kontraesanik aurkitu ez zaiona.

Partikulak, Schrödinger ekuazioaren bidez deskribatzen dira, zein bere emaitzak uhin-funtzio bezala ezagutzen diren. Funtzio hauek, ondular izaera dutenak, modu probabilistikoan interpretatzen dira, partikula bat puntu batean aurkitzeko probabilitatea adieraziz. Materiaren portaera ondularra ez da mundu makroskopikoan ikusten, haien uhin-luzerak infinitesimalak direlako.

Argia eta materian arteko elkarrekintza hainbat ikuspuntutatik ekin daitekeen fenomeno konplexua da. Oinarrizko teoria elektrodinamika kuantikoa den arren, kasu askotan posiblea da errepresentazio sinplifikatuak erabiltzea non portaera ondular klasikoago bat kontsideratzen den, Maxwellen ekuazioen bidez deskriba daitekeena.

Proiektu hau kokatzen den lan eremuetako bat, partikulen bidezko argi uhinen barreiaketa da. Modu honetan, garraio homogeneo baten bidez argi-uhinak mugitzen direnean,

argiaren zati bat xurgatua izaten da, eta beste zati bat difraktatua izaten da inguruan dauden partikulengatik. Partikula hauek argiren bitartez kitzikatzen dira, eta uhinen bigarren mailako elikadura-iturri bihurtzen dira. Partikularen gaineko eremu magnetikoa, eremu magnetiko erasotzaile eta barreiaketa fenomenoaren ondorioz lortutako eremuen baturen bitartez definitu daiteke.

Argiaren eta egitura atomiko ezberdinen arteko elkarrekintzen analisia, konplexutasun handiko fenomenoa dena, garrantzi handiko lan-eremua da nanofotonikaren eremuan, zein teknika numerikoekin edo erdi-analitikoekin ekintzen den, hala nola, dipolo diskretuen bidezko hurbilketak, zein objektibo jarraitu bat errepresentatzen duen zenbaki finituko polarizagarri puntuetan, zeinek eremu magnetikoen bidez elkarrekintzen duten. Tresna ahalmentsua da, gai dena partikula irregular eta tamaina diferenteetan argiaren xurgapena eta difrakzioa kalkulatzen, Maxwellen ekuazioak ebatziz.

Jakina, materia eta argiaren arteko elkarrekintzen emaitzak, argiaren propietateen zein elkarrekintzen duen materiaren (forma, tamaina, erantzun dielektrikoa...) menpe daude. Horrela, hainbat jokaleku kontuan hartu ditzakegu, hala nola, partikula txiki esferiko anisotropokoek edo dipolo erresonanteek igorritako argiaren edota argi erasotzailearen xurgapenaren elkarrekintzaren efektuaren ondorioz lortzen den polarizazioa.

Proiektu honetan, partikula diferenteekin elkarrekintzen duten argiaren barreiaketa simulatzen duen herramienta. aten paralelizazioarekin ekin da. Programaren jatorrizko serie bertsio **Nuno de Sousa** fisikariak diseinatu du, **DIPC**-ko nanofotonika ikerketa baten barruan.

4. KAPITULUA

Konputazio paraleloa

Konputazio paraleloa, problema bat zati txiki eta independentetan exekutatzeko prozesuari deritzo. Prozesu honetan, jatorrizko problematik lortutako zati txiki eta independenteak aldi berean exekutatzen dira hainbat prozesadoretan. Konputazio paraleloaren helburu nagusia eskura dagoen konputazio-ahalmena handitzea da. Modu honetan, aplikazioak azkarrago exekutatzeko eta lehen bideragarriak ez ziren aplikazioak exekutatzeko aukera eskaintzen du konputazio paraleloaren erabilerak.

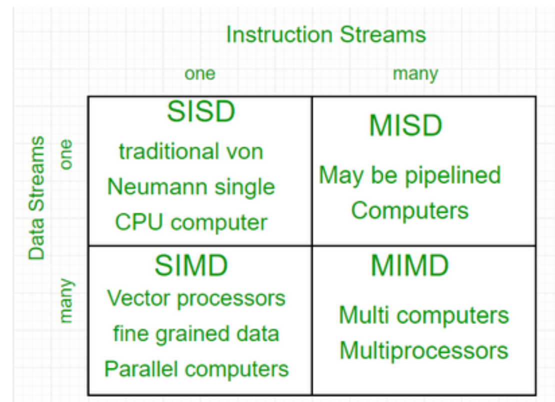
Konputazio paraleloan, paralelismo mota ugari dago, eta garrantzitsuenak ondorengoak dira.

- **Hari mailako paralelismoa:** Paralelismo mota honetan, hariak garrantzia handia hartzen du. Beraz, mota honetan *Multithreading* arkitekturak erabiltzen dira. Arkitektura hauetan, baliabideak harien artean partekatzen dira. Horrez gain, planifikatzaile batek erabakitzen du uneoro zein harik erabiliko dituen baliabideak. Arkitektura hauetan, orokorra da *Simultaneous multithreading* (SMT) erabiltzea.
- **Datu mailako paralelismoa:** Paralelismo mota honetan, eragiketa bera exekutatzen da datu kopuru handi baten gainean. Beraz, mota hauetako arkitekturak *(Single Instruction Multiple Data)* **SIMD** arkitekturak dira. Arkitektura mota hauen barruan, bi prozesadore mota bereizten dira: Bektore-prozesadoreak eta Prozesadore grafikoak **(GPU**).
- **Prozesu mailako paralelismoa:** Paralelismo mota honetan, *multicore* sistemak erabiltzen dira. Prozesu mailako paralelismoan, *Multiple Instruction Multiple Da-*

ta **(MIMD)** arkitekturak erabiltzen dira. Prozesadoreak exekuziorako nukleo asko ditu, bakoitzak berezko baliabideekin.

4.1 Flynn-en sailkapena

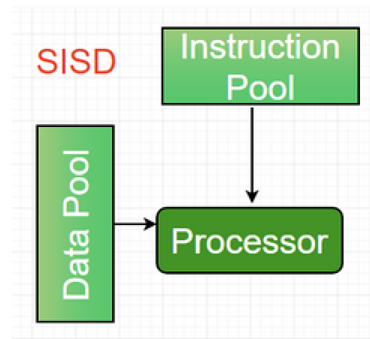
Aurreko atalean azaldu bezala, konputazio paraleloan paralelismo mota ugari daude, eta paralelismo mota bakoitzean konputazioko arkitektura mota zehatz bat erabiltzen da. Konputazioko arkitekturen sailkapenari, Flynn-en sailkapena deritzo. Sailkapen honetan, aldi berean prozesatu daitezkeen agindu eta datu-fluxuen arabera sailkatzen dira sistemak. Modu honetan beraz, Flynn-en sailkapenaren arabera 4 arkitektura mota definitu daitezke, nahiz eta hiru bakarrik erabiltzen diren. Flynnen sailkapenaren eskema 4.1 irudian ikus daiteke.



4.1 Irudia: Flynnen sailkapena

4.1.1 SISD

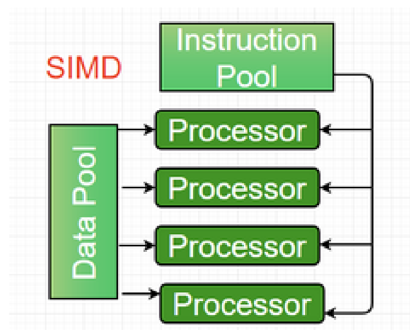
Single Instruction Single Data arkitekturan, ez dago paralelismorik. Arkitektura mota honetako prozesadoreak serieko prozesadoreak dira. Honek esan nahi du, prozesadore bakar batek agindu bakar bat datu-fluxu bakarrarekin exekutatzen duela. Prozesadore horrek, aginduak modu konkurrentean exekuta ditzake. SISD arkitektura eta *von Neumann*-en arkitekturaren berdinak dira. Arkitektura honen eredua 4.2 irudian ikus daiteke.



4.2 Irudia: *Single Instruction Single Data* arkitektura

4.1.2 SIMD

Single Instruction Multiple Data arkitekturan, sistemak agindu bera exekutatzen du dituen prozesadore guztietan, baina prozesadore bakoitzak datu-fluxu ezberdinak erabiltzen ditu aginduaren exekuzioan. SIMD arkitekturan oinarritutako makinak, oso egokiak dira konputazio zientifikoan, normalean arlo honetan egiten diren kalkuluetan matrize eta bektore ugari erabiltzen direlako. Modu honetan, matrizeak eta bektoreak azpi zatitan banatzen dira eta zati horiek prozesadore diferenteetan kalkulatzen dira. SIMD arkitekturan oinarritutako sistema garrantzitsuenak, *Graphical Procesing Unit* (**GPU**) eta bektore prozesadoreak dira. SIMD arkitekturaren eredua 4.3 irudian ikus daiteke.



4.3 Irudia: *Single Instruction Multiple Data* arkitektura

4.1.3 MIMD

Multiple Instruction Multiple Data arkitekturan, sistemak agindu ezberdinak exekutatzen ditu hainbat prozesadoretan. Prozesadore bakoitzak bere datu fluxuaren gainean egiten du lan. Beraz, MIMD motako sistemak edozein aplikazio motetarako balio dute. SIMD eta

MISD arkitekturetan ez bezala, MIMD arkitekturan oinarritutako sistemak modu asinkronoan funtzionatzen dute. MIMD arkitekturaren eredua 4.4 irudian ikus daiteke.

4.4 Irudia: *Multiple Instruction Multiple Data* arkitektura

MIMD makinak, memoria arkitektura ezberdinak erabil dezakete. Memoria arkitektura ezberdinek, abantailak ekar ditzakete aplikazio mota batzuentzako eta desabantailak beste batzuentzako. Sailkapen honen arabera, hiru memoria arkitektura bereizten dira: **Memoria partekatua**, **Memoria pribatua** eta **Arkitektura hibridoa**.

Memoria partekatua

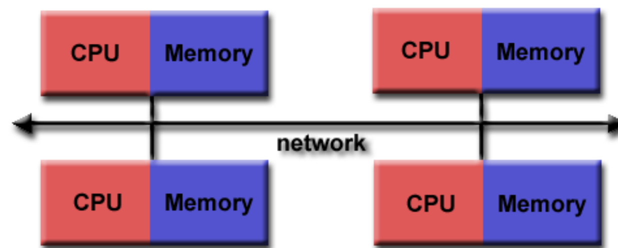
Memoria partekatuan oinarrituriko sistemetan, prozesadore guztiek memoria global bat atzitzen dute. Prozesadoreen arteko komunikazioa memoria partekatuaren bidez egiten da, hau da, prozesadore batek memorian egindako aldaketa gainontzeko guztiek ikus dezakete. Horretarako, arkitektura mota hauetako sistemek zelatari edo *snoopy* bat erabiltzen dute memoriaren koherentzia mantentzeko, ezinbestekoa baita arkitektura hau funtzionatu ahal izateko. Memoria-koherentzia mantentzeko hainbat protokolo erabiltzen dira. Memoria partekatuaren eredua 4.5 irudian ikus daiteke.

4.5 Irudia: Memoria partekatuko sistema

Memoria pribatua

Memoria banatuko sistemetan, ez da memoria global bat erabiltzen: prozesadore bakoitzak bere memoria pribatua du. Memoria banatuetako sistemak komunikazio-sare bat era-

biltzen du prozesadoreak lotzeko. Prozesadore bakoitzak bere memoria duenez, ezin du beste prozesadore baten memoria atzitu; horregatik, memoria banatuko sistemetan memoria globalaren kontzeptua desagertzen da, *cache* koherentziarekin batera. Arrazoi berdinengatik, prozesadore bakoitzak modu independentean erabil dezake bere memoria, eta egindako aldaketak ez du inongo eraginik beste prozesadoreetako memorietan. Prozesadore batek beste prozesadore baten memorian dagoen datu bat eskuratu nahi duenean, memoria horren jabe den prozesadoreari eskatu behar dio, eta datua mezu-trukearen bidez bidaltzen da. Datua noiz eta nola bidaltzen den programatzailearen ardura da. Memoria banatuko sistema baten eskema 4.6 irudian ikus daiteke.



4.6 Irudia: Memoria banatuko sistema

4.2 Eraginkortasuna

Paralelismoaren kontzeptua azaltzerakoan, eraginkortasuna atal nagusienetako bat da, azken finean paralelismoko helburuetako bat exekuzio-denbora jaistea da. Beraz, eraginkortasunaren helburua garbi izan behar da. Horrez gain, paralelismo egoki bat lortzen saiatzerakoan arazoak gerta daitezke, eta ezinbestekoa da arazoak ulertzea, gainditu ahal izateko.

4.2.1 Helburua

Aurreko ataletan azaldu bezala, paralelismoaren helburu nagusia exekuzio denborak murriztea da. Beraz, programen exekuzio-denborak jaisten badira, hiru gauza egin daitezke. Alde batetik, programa bera exekutatu daiteke eta paralelismoari esker, hura denbora gutxiagoan exekutatuko da. Modu honetan, emaitzak azkarrago lortzen dira eta horiekin beste gauzak egiteko denbora geratzen da. Beste aldetik, programa handiagoak denbora berean exekutatzeko ahalmentzen du paralelismoaren erabilerak. Modu honetan, lehen

bideragarriak ez ziren programak exekutatu daitezke. Azkenik, lortzen den denbora murriztapenari esker, aldi berean programa ugari exekutatu daitezke denbora berdintsuan.

4.2.2 Arazoak

Programa baten paralelizazio-prozesua zaila izan daiteke, prozesuan hainbat arazo gerta daitezkeelako. Arazo hauek oso ohikoak dira paralelismoan eta beraz, hauek jakitea oso komenigarria da kodea paralelizatzerakoan.

Serieko frakzioak

Programa bat paralelizatzerako garaian, agertzen den arazo nagusietako bat serieko frakzioak dira. Programa guztietan daude paralelizatu ezin diren zatiak; arrazoi ugariengatik izan daiteke: kodearen zatien arteko dependentziak, hardware mailako baliabide partekatuak ... Modu honetan, nahiz eta prozesadore kopurua asko igo, gerta liteke exekuzio denbora paraleloa ez igotzea espero den bezala. P prozesadore lehiatzerakoan lortzen den **azelerazio-faktorea** 4.1 ekuazioan definitzen da, non **Ts** seriko exekuzio-denbora eta **Tp** P prozesadore paraleloan erabiliz lortutako denbora diren.

$$\mathbf{af} = \frac{Ts}{Tp} \tag{4.1}$$

Modu honetan, azelerazio-faktore onena lortzeko, linealki hasi behar du P kopurua ere linealki hasten den heinean. Mundu errealean ordea, hura lortzea zaila da, paralelizazioko prozesuan gainkarga berriak sortzen direlako.

Azelerazio-faktorea neurtzeaz gain, eraginkortasuna neurtzea ere beharrezkoa da, P prozesadoreak erabiltzean galtzen den eraginkortasuna ikusteko. **Eraginkortasuna** 4.2 ekuazioan definitzen da.

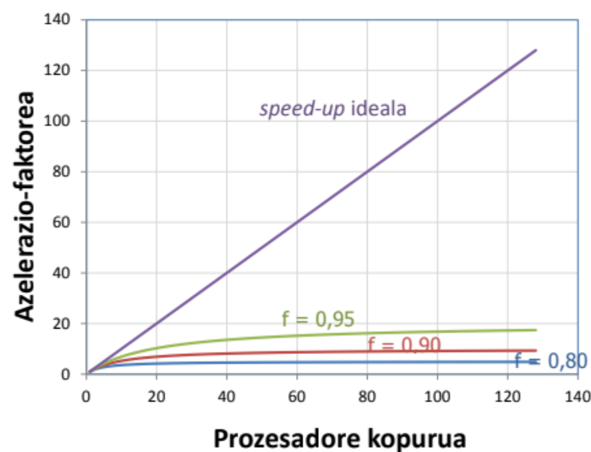
$$\mathbf{erag} = \frac{af}{P} \tag{4.2}$$

Kasu ideala lortzeko, azelerazio-faktorea P izan behar du, hau da, P aldiz prozesadore gehiago jarrita programa P aldiz azkarrago exekutatzen da eta ondorioz, eraginkortasuna %100 balioko du, ez delako eraginkortasunik galtzen. Ordea, lehen azaldu bezala, mundu errealean hau ez da gertatzen paralelismoko gainkargak direla eta. Hau kontuan izanda, bi lege daude arazo hau bi ikuspuntu diferenteekin azaltzen dutenak:

- **Amdahl-en legea:** Esan bezala, programa bat paralelizatzerakoan zati bat seriean exekutatu behar da ezinbestez hainbat arrazoirengatik. Beraz, paralelizatu daitekeen zatia **f**-rekin errepresentatzen da, eta paralelizatu ezin den zatia $1-f$. Modu honetan, Amdahl-en legearen arabera honela adierazten da azelerazio-faktorea:

$$\mathbf{af} = \frac{P}{f+(1-f)*P} \tag{4.3}$$

Beraz, prozesadore kopurua oso handia denean, azelerazio faktoreak $1/(1-f)$ balioko du. Honek esan nahi du, azelerazio-faktorea bakarrik paralelizatu daitekeen zatiaren menpe dagoela. Beraz, nahiz eta prozesadore gehiago erabili, azelerazio-faktorea ez da igoko puntu batetik aurrera. Azaldutakoaren grafikoa, 4.7 irudian ikus daiteke.



4.7 Irudia: Amdahl-en legea

- **Gustafson-en legea:** Paralelismoa ez da kasu guztietan erabiltzen programa bera azkarrago exekutatzeko; beste batzuetan tamaina handiagoko programak exekutatzeko asmoarekin erabiltzen da. Modu honetan, jakinda paralelizatzeko garaian paralelizatu ezin den zati bat dagoela, programa handitzean paraleliza daitekeen zatia handitzen da baino paralelizatu ezin dena ez da handitzen. Hau kontuan izanda, Gustafson-en legearen arabera azelerazio-faktorea 4.4 ekuazioan azaltzen da.

$$\mathbf{af} = (1-f)+f*P \tag{4.4}$$

Aurreko bi ataletan ikusi dugunez, paralelismoko bi ikuspuntu ikusi ditugu: programa bat azkarrago exekutatzea eta problema bat handitzea. Modu honetan, eskalagarritasunaren

kontzeptua agertzen da. Eskalagarritasuna, sistemak prozesadore kopurua hazten denean eraginkortasuna mantentzeko neurria da. Bi eskalagarritasun mota daude aurreko atalean azaldutako bi kasuekin zerikusia dutenak.

- **Eskalagarritasun sendoa:** P prozesadore kopurua, aldatzean eraginkortasuna nola aldatzen den neurtzen du. Programa tamaina berarekin, prozesadore gehiago erabiltzerakoan, prozesadore bakoitzak lan-karga gutxiago izango du.
- **Eskalagarritasun ahula:** P prozesadore kopurua eta programaren tamaina aldatzean eraginkortasuna nola aldatzen den neurtzen du. Programaren tamaina handitzean, P kopurua ere hazten bada, prozesadore bakoitzak, lan-karga bera izango dute.

Sinkronizazioa eta komunikazioa

Programa bat paralelizatzerako garaian baliteke harien artean lehiaketak gertatzea. Hau ez gertatzeko garrantzitsua da harien arteko sinkronizazio egokia erabiltzea programa modu egokian funtzionatzeko. Horretarako, sinkronizazioa zehatza eta egokia izan behar da, hau da, sinkronizazio-mekanismoak toki egokian eta modu egokian erabili behar dira trafiko asko ez sortzeko eta programa ahalik eta azkarren exekutatzeko. Sinkronizazio-mekanismo ugari daude, baino gehien erabiltzen direnak sekzio kritikoak dira.

Memoria

Memoria erabiltzerakoan, datuen kontsistentzia mantendu behar da. Bi hari edo gehiagoren artean aldagai partekatuak dituztenean, memoriarekin egiten diren atazak kontsitenteki egin behar dira atomikotasuna bermatzeko. Horretarako, bi kontsistentzia-eredu daude: sekuentziala eta malgua. Eredu sekuentzialak murriztapen asko ezartzen du eta eragin handia izan dezake eraginkortasunean. Eredu malguek aldiz, memoria-aginduen arteko orden-erlazio batzuk ez betetzea ahalbidetzen dute.

5. KAPITULUA

Hardwarearen analisia

Programa bat exekutatzeko garaian erabilitako lengoaia, liburutegi, tresnak eta sistema eragileaz gain, erabiltzen den makinak ere garrantzia handia du lortzen den emaitzetan. Makina horrek erabiltzen duen prozesadorea eta memoria dira *hardware*aren aldetik emaitzetan gehien eragiten duten faktoreak.

Proiektu honetan, probetarako erabili den makinaren ezaugarriak 5.1 taulan ikus daitezke.

PUZ	AMD Ryzen 3900x
Memoria	Corsair Vengeance Pro 2x16GB DDR4 3200Mhz CL16 (*Dual Channel*)
GPU	nvidia RTX 2080 Ti
Plaka nagusia	Gigabyte Aorus Ultra X570

5.1 Taula: Makinaren ezaugarriak

Proiektuan erabilitako prozesadorea, 2019-an kaleratutako AMD-ko **Zen 2** arkitekturako **Ryzen 3900x** prozesadorea da. Prozesadore honek 12 *core* ditu baina *Simultaneus MultiThreading* (SMT) teknologiari esker 24 nukleo logiko ditu. Prozesadoreak, 64 biteko x86 arkitektura du eta 7 nm FinFET litografia du. Prozesadorearen nukleo guztiak 3,8 GHz-ko abiaduran funtzionatzen dute, baino AMD-ren *Precision Boost* teknologiari esker, momentuko lan-kargaren arabera nukleoen erloju-maiztasuna automatikoki aldatzen joaten da, 5.1 irudian ikus daitekeen bezala.

Memoriari dagokionez, *Double Data Rate 4* (DDR4) SDRAM memoria erabiltzen da. DDR memoria motak, ziklo bakoitzean idazketa bat eta irakurketa bat egin ditzake. Beste aldetik, memoria 3200 MHz-ko abiaduran funtzionatzen du (1600 MHz-ko abiadura era-

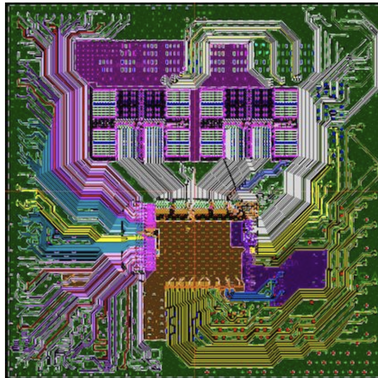
Sensor	Value	Min	Max
Clocks			
Core #0	4266 MHz	3413 MHz	4491 MHz
Core #1	4291 MHz	3011 MHz	4516 MHz
Core #2	4266 MHz	3373 MHz	4341 MHz
Core #3	3593 MHz	3373 MHz	4541 MHz
Core #4	3593 MHz	3027 MHz	4466 MHz
Core #5	3593 MHz	3027 MHz	4291 MHz
Core #6	4291 MHz	2874 MHz	4291 MHz
Core #7	4291 MHz	2874 MHz	4291 MHz
Core #8	3593 MHz	2874 MHz	4291 MHz
Core #9	4291 MHz	3373 MHz	4341 MHz
Core #10	3593 MHz	3373 MHz	4291 MHz
Core #11	4266 MHz	3433 MHz	4291 MHz

5.1 Irudia: Ryzen 3900x nukleoen erloju-maiztasuna

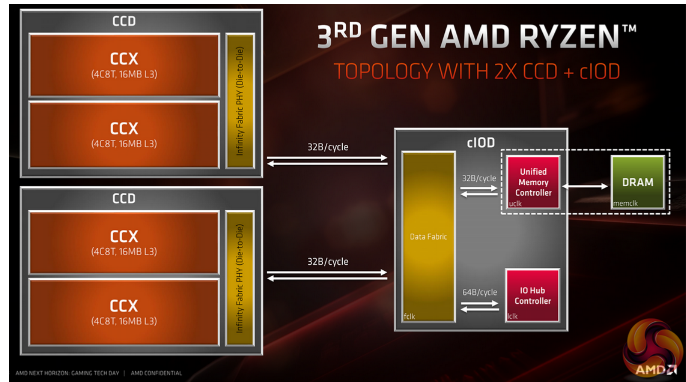
ginkorra). Memoriaren *Column Address Strobe or Signal* (CAS) latentzia 16 ziklokoa da; horrek esan nahi du memoriak 16 erloju ziklo behar dituela datu bat eskuratzeko.

5.1 Zen 2

Zen 2 arkitektura AMD enpresak garatutako prozesadore arkitektura da. **Zen 2**, **Zen** eta **Zen +**-ren ondorengoa da. **Zen 2**, 2019 urtean garatu zen. Arkitektura honek, 8 **core**ko 2 *chiplet* eta *input output controller* unitate bat ditu. *Chiplet* bakitzak 4 nukleoz osaturiko 2 *Core complex* (CCX) ditu. CCX bakoitzak L3 cache bat dauka. IO kontrolagailuak, *chiplet* barneko nukleoen arteko komunikazioak izan ezik beste komunikazio guztiak kontrolatzen ditu, *Peripheral Component Interconnect express* (PCIe 4.0), memoria kanalen eta *chiplet*en arteko komunikazioak hain zuzen ere. *Chiplet*en arteko komunikazioak AMD-ren *Infinty Fabric* loturen bidez egiten dira. *Chiplet*ak 7 nm FinFET prozesuan eginda daude; IO kontrolagailua aldiz, 12 nm-ko *Global Foundries* prozesuan eginda daude. Zen 2 arkitekturaren eskema 5.2a, 5.2b eta 5.3 irudietan ikus daiteke.

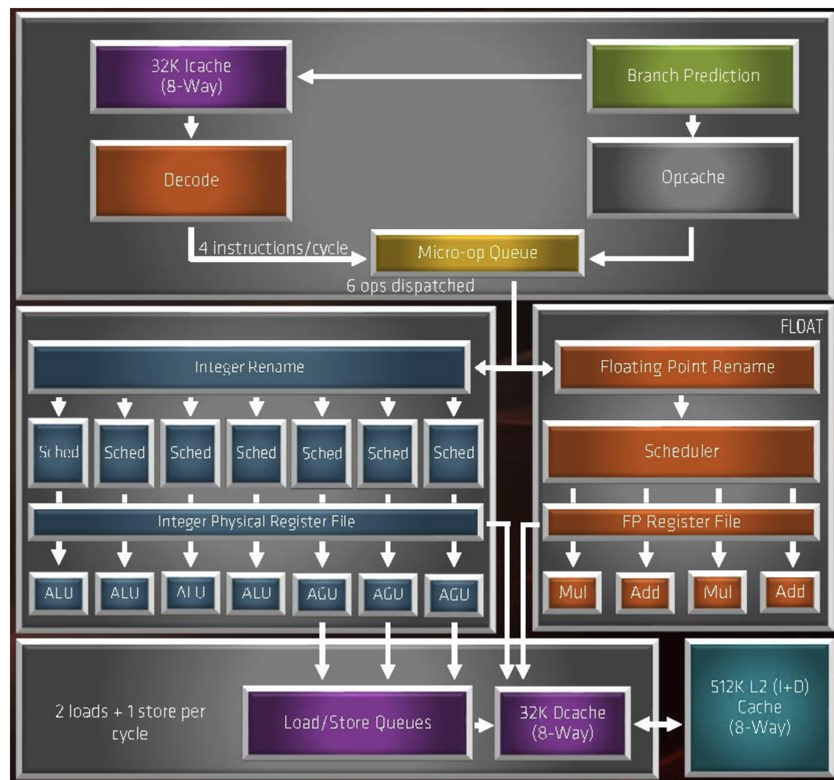


(a) Zen 2 txip baten eskema



(b) Zen 2 arkitekturaren eskema

5.2 Irudia: Zen 2 txip baten eta nukleoen eskemak

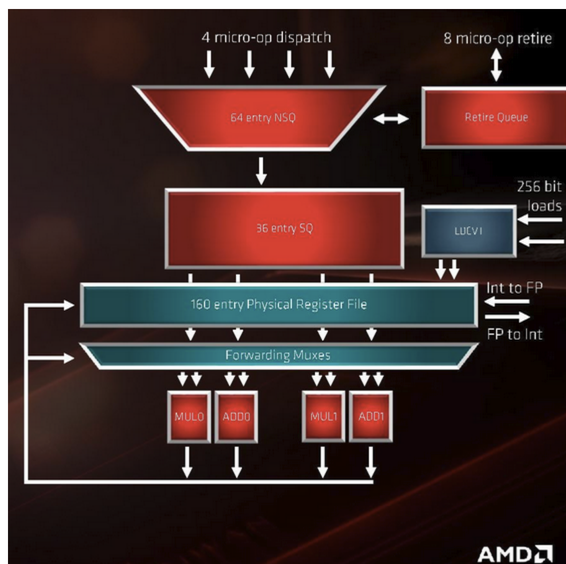


5.3 Irudia: Zen 2 arkitekturaren eskema

5.1.1 Koma higikorreko unitatea

Zen 2-ko koma higikorreko unitateak 256 biteko datuak prozesatzen ditu. Honen abantaila handiena da AVX2 motako kalkuluak ziklo bakarrean egiten direla.

Koma higikorreko unitatearen eskema 5.4 irudian ikus daiteke.



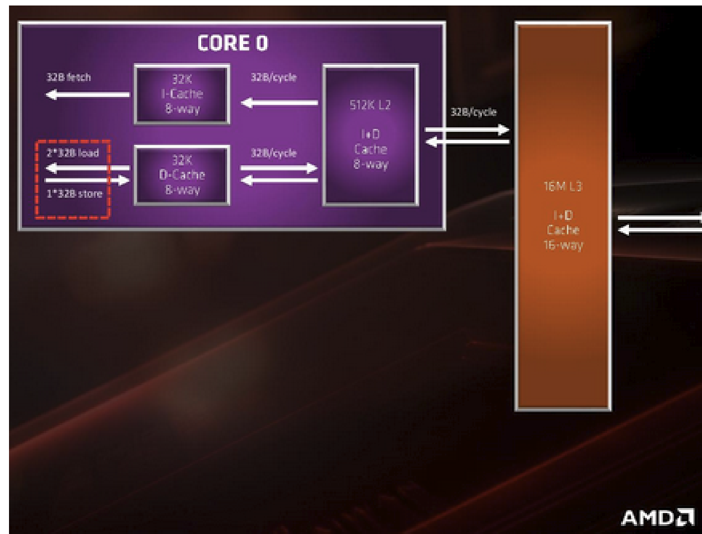
5.4 Irudia: Zen 2 arkitekturako koma higikorreko unitatearen eskema

5.1.2 Zenbaki osokoen unitatea

Osoko unitateen planifikatzaileak 6 mikro-agindu aldi berean onartu ditzake. Agindu hauek 224 sarrera dituen berrordenatze-buffer batean sartzen dira. Osoko unitateak 7 exekuzio-portu ditu: 4 unitate aritmetiko-logiko eta helbideak sortzeko 3 unitate.

5.1.3 Cache memoria

Zen 2 arkitekturako L1 cacheak banatuta daude, 32 KB datuetarako eta 32 KB aginduetarako, nukleo bakoitzerako, gainera, 8 blokeko elkargarritasuna du. L2 cachea, 8 blokeko elkargarria da eta bateratura da, 512 KB nukleoko. L3 cacheak, 16 MB-ko 4 cache ditu. L1 cachearen latentzia 4 ziklokoa da, L2 cachearen latentzia, 12 ziklokoa eta L3-koa 40 ziklokoa da. Hona hemen cache memoriaren eskema bat 5.5 irudian.



5.5 Irudia: Zen 2 arkitekturako cachearen eskema

5.1.4 Infinity Fabric

Infinty Fabric AMD-ren konexio busa da; busak, chiplet ezberdinen eta IO kontrolagailuen arteko loturak osatzen ditu. Zen 2-ko Infinity Fabric PCIe 4.0 bertsioa onartzen du; horrez gain, busaren zabalera 512 bitekoa da. Infity Fabric bertsio honen erloju-maiztasuna, RAM memorien erloju-maiztasun bera izatea ahalbidetzen du 1 : 1 moduarekin; horrez gain, 2 : 1 moduan, Infity Fabric-aren erloju-maiztasuna RAM memoriak duenaren erdia izaten da. Beraz, RAM memoriaren erloju-maiztasuna 3.800 MHz tik beherakoa denean 1 : 1 modua erabiltzea gomendatzen da, baina memoriaren erloju-maiztasuna oso altua denean 2 : 1 modua erabiltzea gomendatzen da Infinity Fabric-aren latentziak ez igotzeko hainbeste.

6. KAPITULUA

Aplikazioaren Paralelizazioa

Proiektu honetan, argiaren barraiatzea simulatzen duen eta **DIPC**-ko ikerlariek erabiltzen duten programaren bertsio paraleloa sortu da. Proiektuaren garapenak fase ugari izango ditu eta lehenengo fasean jatorrizko kodearen analisia egitea da.

Argiaren barreiaketaren simulatzaileak, argia partikula diferentetan nola barreiatzen den kalkulatzen du. Horretarako, simulazioa hasi aurretik simulazioaren ezaugarriak aukeratzen dira, hala nola, partikulen tamaina eta material mota. Beste aldetik, zein kalkulu egin nahi diren eta kalkulu horiek zein modutan egin nahi diren ere erabaki ahal dira. Azkenik, partikulen propietateak ere aldatu daitezke; adibidez, partikulek xurgatzen duten argi-kopurua aldatu daiteke. Modu honetan, egindako aldaketa guztien arabera simulatzaileak balio ezberdinak kalkulatuko ditu eta beraz, egoera ezberdinetan argia nola barreiatzen den simula daiteke.

6.0.1 Simulatzailearen egitura

Simulatzailearen egitura konplexua den arren, labur azaltzeko, simulatzaileak klase nagusi bat du eta bertan simulatzailearen prozedura guztiak egiten dira. Horrez gain, simulatzaileak klase ugari ditu, bakoitzak eginkizun zehatz eta diferenteak ditu. Programaren portaera ondorengoa da. Lehenik, parametroen fitxategian simulaziorako parametroak eta zein kalkulu mota egin nahi diren idazten dira (partikulen tamaina eta materiala, iterazio kopurua, λ_{min}, λ_{max}, ...). Ondoren, simulatzailea exekutatzen da; exekuzioaren hasierako zatian adierazitako parametroak kargatzen dira. Behin datuak kargatuta, exekuzioan

zehar erabiliko diren aldagaiak hasieratzen dira. Aldagaiak hasieratu ondoren, iterazioen exekuzioa hasten da; iterazio bakoitzean λ zehatz batekin egiten dira kalkuluak. Kalkuluen prozesuaren egitura konplexua da, baino azaletik azaltzeko, matrize ugari hasieratzen dira programaren hainbat klasetan eta matrize horiekin hainbat kalkulu aljebraiko egiten dira. Hasierako kalkulu horiek egin ondoren, erabakitako kalkulu moten arabera kalkulu ezberdinak egingo dira. Azkenik, iterazio bakoitzean lortutako emaitzak-irteera fitxategietan inprimatzen dira.

Jatorrizko kodea **C++** lengoaian idatzita dgo. Simulatzaileak hainbat klase ditu bertako fisikako atalen eginkizunak betetzen dituzten funtzioak eta parametroekin. Informatikaren ikuspuntutik, klaseak aztertu behar dira exekuzio fluxua nolakoa den jakiteko, baina behin aztertzen denean, kodearen zati garrantzitsuenetan sakonduko da. Hau guztia esanda, kodearen klase garrantzitsuenak ondorengoak dira: **io**, **materials**, **particle**, **parameters**, **load_equations** eta **main** klase bera.

Simulatzailearen klaseak

Klase hauek ondorengo eginkizunak dituzte simulatzailean.

- **io:** Klase honetan, simulazioak erabiliko dituen parametroen karga egiten da. Horretarako **charger** izeneko beste klasetik, fitxategiak irakurtzeko funtzioari deitzen dio. Ondoren, fitxategiko parametroak hainbat aldagaitan gordetzen ditu. Azkenik, **main** klasetik parametroen balioak aldagaietan esleitzeko funtzioak eskaintzen ditu.

- **materials:** Klase honetan simulazioan erabiliko dien partikulen materialak kargatzen dira. Horretarako, *material_list* izeneko fitxategian simulatzailerako dauden material diferenteak agertzen dira. Fitxategiaren lehenengo zutabean, materialaren izena jartzen du (Si, SIMO, AU, PMMA, ...) eta bigarrengoan, material horri dagokion propietateen fitxategiaren izena. 6.1 irudian *material_list* fitxategiaren edukia ikus daiteke.

- **particle:** Klase honetan, simulatzailean erabiliko diren partikulen ezaugarriak gordetzen dira, eta parametro bezala erabiltzen da simulazioaren kalkuluak egiteko.

- **parameters:** Klase honetan, simulazioko parametro nagusiak gordetzen dira. Bertan, λ_{min} eta λ_{max} balioak kargatzen dira, **io** klaseko objektutik. Horrez gain, si-

```
1   SiMO  dielectric_eps/SiMO.dat
2   silicon  dielectric_eps/Si.dat
3   Si  dielectric_eps/Si.dat
4   Au_ext  dielectric_eps/Au_ext.dat
5   TiO2  dielectric_eps/TiO2.dat
6   Au  dielectric_eps/Au_ext.dat
7   Si_20C dielectric_eps/Si_20C.dat
8   PMMA  dielectric_eps/PMMA.dat
9   NaYF4 dielectric_eps/NaYF4.dat
10  melamine dielectric_eps/melamine.dat
11  Res dielectric_eps/Res.dat
12
```

6.1 Irudia: *material_list* fitxategiaren edukia

mulazioan zehar egingo den iterazio kopuruaren balioa eta λ-ren uhin-zenbakia gordetzen dira ere.

- **load_equations:** Klase honetan, simulazioko kalkulo garrantzitsuenetakoak egiten dira. Bertan, matrizeen gainean ibiltzen diren funtzioak daude eta matrize horien karga eta kalkuluak beharrezkoak dira bukaerako **solver**-eko kalkuluak burutzeko.

- **main:** Klase hau simulazioaren atal nagusia da: bertan objektu guztiak sortu eta iterazio guztiak exekutatzen dira. Klase honetan, exekuzio denbora eta emaitzak inprimatu dira.

Exekuzio-fluxua

Simulatzailearen eginkizunean pixka bat sakontzeko asmoz, simulazioaren exekuzio-fluxua egin da. Fluxu honetan, lehen azaletik azaldutako eginkizuna modu zehatzago batean azaltzen da.

1. Simulazioaren parametroak *simul_parameters* izeneko fitxategian idazten dira. Bertan, hasierako *lambdaren* aldagaiaren balioa, aldagaiaren amaierako balioa, exekuzioaren iterazio kopurua, partikula kopurua, partikulen materiala eta erabiliko den kalkulu mota aukeratzen da. Parametro hauen balioa gordetzen duen fitxategiaren edukia 6.2 irudian ikus daiteke.

2. Simulazioko parametroen kargak egiten dira *io* klaseko **values** izeneko objektuan. Ondoren, exekuzioan zehar erabiliko diren aldagaiei balio horiek esleitzen zaizkie.

```
1   1000  #initial lambda
2   2000  #final lambda
3   10000 #number of steps
4   1 #epsilon_0
5   1 #epsilon_medium
6   0 #0 for transport, 1 for emission
7   0   #ux incident wave direction (k_x)
8   0 #uy incident wave direction (k_y)
9   1 #uz incident wave direction (k_z)
10  1 #electric field Amplitude
11  0 #x position of the source
12  0 #y position of the source
13  0 #z position of the source
14  1189  #number of particles
15  structure_files/structure_1189_Si.dat #path of the structure file.
16  results/output_test_  #output first word for output
17  position_fields.txt #input positions to evaluate the field
18  1 #0 deactivate, 1 activate sections
19  0 #0 deactivate, 1 activate particle fields
20  0 #0 deactivate, 1 activate polarizations
21  0 #0 deactivate, 1 activate projections
22  0 #0 deactivate, 1 activate forces
23  Direct  #method used: Direct or BiCStab
24  120 #number of iterations (only for BiCStab)
25  1e-9  #tolerance (only for BiCStab)
```

6.2 Irudia: *simul_parameters* fitxategiaren edukia

3. Simulazioko λ_{min}, λ_{max} eta iterazio kopurua jakinda, iterazioen banaketa egiten da prozesadore kopuruaren arabera.

4. Simulazioaren zati independenteak hasten dira; prozesadore bakoitzeko **Particle**, **Parameters** eta **Material** objektuak sortzen dira.

5. Prozesadore bakoitzak, kalkulatu behar dituen λ kopuruak bektore batean gordetzen ditu. Batzuk λ gehiago kalkulatuko dituzte, lan-banaketako estrategiaren arabera.

6. Simulazioaren begizta nagusia hasten da, eta bertan prozesadore bakoitzak uneko iterazioari dagokion λ balioa gordetzen duen bektoretik lortzen dute.

7. Prozesadore bakoitzak iterazio horren kalkulurako erabiliko diren matrizeak eta bestelako aldagaiak hasieratzen ditu.

8. Simulazioko partikulen materiala aztertzen da eta hiru kasu daude: *Meyer gold* eta *Resonant* motako materialentzat tratamendu berezia aplikatzen da; bestela, kasu orokorrari dagokion tratamendua egiten da.

9. *Transport* fasea hasten da, eta fase honetan *simul_parameters* fitxategian jarritako kalkulu motaren arabera kalkulu eta analisi ezberdinak egiten dira. Kalkulu hauek egiteko, **Parameters** eta **load_equations** klaseko funtzioak erabiltzen dira.

10. *Emission* fasea hasten da, eta fase honetan berriro ere *simul_parameters* fitxategian aukeratutako kalkulu motaren arabera, kalkuluak egingo dira edo ez. Kalkuluak egiten diren kasuetan, **Parameters** eta **load_equations** klaseko funtzioak erabiltzen dira.

11. *Solver* fasea hasten da, exekuzioko zati garrantzitsuenetako bat da. Fasean, *simul_parameters* fitxategian aukeratutako "solver" motaren arabera metodo ezberdinak erabiliko ditu emaitzak kalkulatzeko. Metodoen artean **Dircet solver** eta **BiCStab** metodoak daude.

12. Emaitzak kalkulatzen direnean, irteera-fitxategietan idazten dira. Fitxategi horiek **results** izeneko direktorioaren barnean sortzen eta idazten dira. *simul_parameters* fitxategian jarritako parametroen arabera fitxategi ezberdinak sortuko dira egindako kalkuluen arabera.

13. Iterazio bakoitzaren bukaeran teorema optikoaren egiaztapena egiten da. Horretarako, aurretik kalkulatutako emaitzak erabiltzen dira eta teorema optikoaren egiaztapena irteera estandarretik inprimatzen du.

14. Iterazio guztiak kalkulatzen direnean, *0 hariak* simulazio osoaren exekuzio-denbora kalkulatzen du. Ondoren, irteera estandarretik eta *execution_times* izeneko fitxategian exekuzio-denborak inprimatzen ditu. Azkenik, simulazioa amaitzen da.

Kode honek liburutegi batzuk erabiltzen ditu simulazioko kalkuluak egiteko. Liburutegi garrantzitsuenak **Armadillo** eta **intel Math Kernel Library (MKL)** dira. Bi liburutegi hauek ondoren azalduko diren esperimentazio guztietan erabili izan dira.

Armadillo

Armadillo, aljebra linealeko liburutegi bat da. Liburutegi hau, batez ere **C++** lengoaiarekin erabiltzeko diseinatuta dago. Armadilloren helburua, programetako matrizeen edo aljebra lineala erabiltzen duten aldagaien erabileraren erraztasuna eta abiadura hobetzea

da. Horretarako, Armadillok sintaxi propioa du liburutegiko funtzioak erabiltzeko eta aldagaiak erazagutzeko. Horrez gain, Armadillok zenbaki osoen, koma higikorren eta konplexuen erabilera onartzen ditu. Armadillo hainbat arloetako kalkuluetarako erabili daiteke: *machine learning*, patroien erazagupena, konputagailu bidezko ikusmena, seinaleen prozesaketa, bio-informatika, estatistika eta finantza eta kalkulu zientifikoetarako adibidez. Armadillok eskaintzen duen liburutegia erabili ahal izateko, **LAPACK** izeneko tresna behar du, edo tresna honen errendimendu altuko ordezkoak, **Intel MKL** eta **OpenBLAS**. Armadillo liburutegia, konputazio paraleloko **OpenMP** eta **OpenMPI**-rekin guztiz integratzen da. Armadillok lizentzia permisiboa duenez, kode irekiko nahiz pribatuko softwarean erabili daiteke.

MKL

Intel MKL edo *Intel Math Kernel Library*, liburutegi bilduma bat da Intelek sortua. Liburutegi bilduma honetan; **LAPACK**, **BLAS**, **FFT**, **PARDISO** eta beste liburutegi asko daude. Gainera, liburutegi hauek optimizatu dira eraginkorrago funtzionatzeko, batez ere Intel etxeko hardwarean.

6.0.2 Esperimentazioa

Proiektu honetan, esperimentu ugari egin dira simulatzailearen exekuzio-denbora hobetzeko. Horretarako, konputazio paraleloko teknikak aplikatu dira eta lortutako emaitzak aztertu dira ondorio batera iritsiz. Egindako esperimentoen artean, sailkapen orokor bat eginez, esan dezakegu egindako esperimentazioak 3 ereduetan banatzen direla. Alde batetik **OpenMP** tresna erabiliz, hau da, memoria partekatuko ereduko tresna erabiliz. **OpenMP** inguruan egin dira esperimentazio gehienak, jatorrizko ideia **OpenMP** tresna erabiltzea baitzen. Beraz, tresna honen inguruan 3 bertsio egin dira proiektuaren garapenean. Bertsio bakoitza aurrekoaren hobekuntza bezala garatu da, eta guztiek simulatzailearen jatorrizko portaera mantentzen dute. Eredu honen azalpenaren bukaeran, bertsio guztien emaitzak azalduko dira eta horren ostean, hiru bertsioen arteko konparaketa egingo da.

Erabiliko den bigarrengo eredua **OpenMPI** da. Eredu hau memoria pribatuan oinarritzen da eta beraz, **OpenMP**-kin konparatuta, modu ezberdinean programatu behar da. **OpenMPI**-rekin bakarrik bertsio bat garatu da eta bertsio hori oso antzekoa da **OpenMP**-ko hirugarrengo bertsioarekin, egitura berdintsua baitute. Beraz, **OpenMPI**-ren eta **OpenMP**-ko hirugarrengo bertsioak konparatu dira, ondorio batera iritsiz.

Azkenik, **CUDA**-ren eredua geratzen da. **CUDA** erabiltzeko *GPU* bat erabili behar da eta printzipioz, **OpenMP** edo **OpenMPI**-rekin batera funtziona dezake. Zoritxarrez, jatorrizko proiektuan erabiltzen diren aldagai eta funtzioek sortzen duten bateragarritasun arazoengatik ezin izan da **CUDA**-ko bertsiorik garatu. Hala ere, **CUDA**-ren erabilera etorkizunean onuragarria izan daitekeenez, hura aipatzea merezi duela erabaki da.

6.1 OpenMP

Proiektuaren hasierako helburua jatorrizko simulatzailearen paralelizazioa egitea izan da. Paralelizazio hau **OpenMP** tresnarekin egitea pentsatu zen.

OpenMP aplikazioen programazio interfaze (API) bat da. API honek multiprozesuan eta memoria partekatuan oinarritzen da. **OpenMP** *C*, *C++* eta *Fortan* lengoaietan funtzionatzeko diseinatuta dago eta hainbat plataforma eta arkitektura desberdinetan funtzionatzeko prest dago, *Unix*, *GNU/Linux* eta *Windows* plataformetan adibidez. Hau lortzeko, API honek konpilatzailearen direktibak edo sasi-aginduak, liburutegiko errutinak eta ingurune-aldagaiak erabiltzen ditu exekuzioaren portaera adierazteko.

OpenMP-ko hainbat funtzio eta direktiba nagusiak azalduko dira ondoren:

Aldagaien izaera

OpenMP erabiltzen dugunean, kontuan izan behar dugun gauza nagusietako bat erabiltzen diren aldagaien izaera jakitea da. **OpenMP** memoria partekatuan oinarrituta dago eta beraz, erabiltzen diren aldagai guztiak partekatuak dira. Honek esan nahi du, aldagai hori edozein harik irakurri eta idatzi dezakeela. Beraz, aldagaian egindako aldaketak erabiltzen duten hari guztiei eragingo die. Bestalde, gure asmoa hari bakoitzak aldagai baten kopia pribatua izatea bada, kodean aldagaia pribatua izatea nahi dugula adierazi behar dugu *private* klausularekin. Modu honetan, aldagaien gainean egindako aldaketak egin duen hariko aldagaian bakarrik eragina izango du. Aipatutako bi izaeraz gain, *reduction* izeneko izaera dago. Izaera honekin, prozesadore bakoitzak aldagai horren kopia bat du, eta adierazitakoaren eragiketa motaren arabera, kalkulu mota bat egingo da prozesadore baten aldagaiaren gainean; kalkulu horretan, prozesadore guztiek parte hartuko dute.

Identifikadoreen funtzioak

Mota hauetako direktibak hariak identifikatzeko erabiltzen dira. Bi dira nagusiak: zein den hari bakoitza eta guztira zenbat hari diren. Exekuzioan parte hartzen duten hari bakoitzaren identifikadorea jakiteko, *omp_get_thread_num()* funtzioa dugu. Beraz, pentsa dezagun 12 hariko exekuzioa dugula; kodearen zati paraleloan funtzioari deitzen zaionean hariek 0-tik 11-ra arteko balioak izango dute. Hariak identifikatzeko, aipatutako funtzioa erabiltzen da exekuzio-atalak banatzeko edo zati zehatz bat hari batek soilik exekutatzea nahi dugunean, batez ere.

Exekuzioan parte hartzen duten harien kopurua jakiteko *omp_get_max_threads()* funtzioa dugu. Funtzio hau erabiltzen da, batik bat, dauden harien kopuruaren araberako banaketak egin behar direnean; adibidez, iterazio kopuru zehatz bat, dauden hari kopuruen arabera banatu behar denean.

Sasi-aginduak

OpenMP-k sasi-agindu ugari ditu eta hauek exekuzioren portaera zehazteko balio dute. Hemen daude **OpenMP**-ko direktiba erabilienak.

- **parallel:** Sasi-agindu honek kode baten zati paraleloaren hasiera adierazten du. *C/C++* lengoaietan, sasi-aginduaren sintaxia ondorengoa da: #pragma omp parallel.

- **section:** Sasi-agindu honek kodearen zati bat paraleloan exekutatu datekeela adierazten du; adierazitako zati bakoitzarentzat hari bat erabiltzen da. Sasi-agindu hau erabilgarria da hainbat zati independente aldi berean exekutatzeko banatu nahi ditugunean. *C/C++* lengoaietan, sasi-aginduaren sintaxia ondorengoa da: #pragma omp section

- **critical:** Sasi-agindu honek kodearen zati bat sekzio kritiko bezala adierazten du. Beraz, aldi berean soilik hari bat sar daiteke sekzio kritiko horretan; gainontzeko hariak zain geratuko dira barruan dagoen haria sekzio kritikotik irten arte. Beste aldetik, **atomic** sasi-agindua dugu. Sasi-agindu hau, **critical** sasi-aginduaren antzekoa da, azken finean horren kasu partikularra baita. **atomic** sasi-aginduaren erabilera *RMW* motako aginduetarako erabiltzen da. *C/C++* lengoaietan, bi sasi-aginduen sintaxia ondorengoa da: #pragma omp critical eta #pragma omp atomic

- **barrier:** Sasi-agindu hau sinkronizazioko sasi-agindu bat da, *barrier* erabiltzeak kodearen zati batean hesi bat jartzea suposatzen du. Modu honetan, hariak hesira iristen direnean bertan geratzen dira hari guztiak hesira iristen diren arte. Direktiba hau asko erabiltzen da harien arteko sinkronizazioa behar denean. *C/C++* lengoaietan, sasi-agінduaren sintaxia ondorengoa da: #pragma omp barrier

- **for:** Sasi-agindu honek begizta bateko iterazioak banatzen ditu direktiba horretara iristen diren harien artean. *C/C++* lengoaietan, direktiba honen formatua ondorengoa da: *#pragma omp for [klausulak]*. Direktiba honetan hainbat klausula erabili daitezke funtzionalitate ezberdinetarako. Alde batetik, *private* klausula dugu. Klausula honetan for begiztan erabiliko diren aldagaiak pribatuak izango direla deklaratzen da. Modu honetan, hari bakoitzak aldagai horren kopia pribatua izango du eta, beraz, begiztaren barnean aldagai horretan egiten diren aldaketak ez dira hari guztietarako berdinak izango, bakoitzak bere aldagaia duelako. Beste aukera *shared* klausula erabiltzea da, baina kontuan izanda **OpenMP** memoria partekatuan oinarrituta dagoela, defektuz aldagai guztiak partekatzen dira hari guztien artean *private* klausula erabiltzen ez den bitartean.

  Beste klausuletako bat *schedule* klausula dugu. Klausula honek begiztaren iterazioen banaketa mota zehazten du. Mota bakoitzak bere abantaila eta desabantaila ditu paralelizatutako begizten portaeraren arabera. Mota nagusiak ondorengoak dira.

  - Static : *Schedule* mota honetan, begiztaren iterazioen banaketa modu estatikoan egiten da. Horretarako, begiztaren exekuzioa *chunk* izeneko zatitan banatzen da eta zati hauek harietan banatzen dira modu zirkular batean. *Chunk* tamaina aukera daiteke *schedule* klausularen barnean, honen sintaxia *schedule(static, chunk-size)* da. *Chunk-size* tamaina aldatzerakoan, banaketen tamainak aldatzen dira; honen adibidea 6.3 irudian ikus daiteke. Banaketa estatikoaren erabilera egokiena begizta baten iterazioen lan-karga oso antzekoa denean da.

  - Dynamic : *Schedule* mota honetan, begiztaren iterazioen banaketa dinamikoki egiten da. Horretarako, iterazioak tamaina desberdineko *chunk* edo zatitan banatzen dira. Ondoren, hari bakoitzak esleitu zaizkion zatia exekutatu duenean zati berri bat esleitzen zaio, amaitzen diren arte. Banaketa mota honetan ere *chunk* tamaina aukera daiteke modu honetan : *schedule(dynamic, chunk-size)*. Adibide bat 6.4 irudian ikus daiteke. Banaketa dinamikoaren erabilera egokie-

```
schedule(static):
****************
                ****************
                                ****************
                                                ****************
```

```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

6.3 Irudia: *static* banaketa motaren eskema, chunk kopuru ezberdinekin

na begizta baten iterazioen arteko lan-karga desorekatua denean da. Lan-karga orekatua denean, banaketa dinamikoa erabiltzea denbora zigorra ekartzen du banaketak momentuan egiten direlako.

```
schedule(dynamic):
*   ** **  * * *  *      *  *    **   *  *  * *       *  *   *
  *       *     *    * *     * *   *    *        * *   *    *
 *       *    *     * *   *   *     *  *       *  *  *  *  *   *
   *  *     *    * *    *  *    *    *    ** *  *   *     *   *
```

```
schedule(dynamic, 4):
            ****                    ****                    ****
****            ****    ****            ****        ****
    ****            ****    ****            ****        ****
        ****                    ****            ****
```

6.4 Irudia: *dynamic* banaketa motaren eskema

- Guided : *Schedule* mota honetan, begiztaren iterazioen banaketa modu dinamikoaren antzera egiten da. Horretarako, iterazioak tamaina desberdineko *chunk* edo zatitan banatzen dira. Ondoren, hari bakoitzak esleitu zaizkion zatia exekutatu duenean zati berri bat esleitzen zaio, amaitzen diren arte. Modu honen eta modu dinamikoaren ezberdintasuna *chunk*-en tamaina da. *Guided* moduan, *chunk* tamaina jaisten joaten da hariak *chunk*-ak exekutatzen dituzten heinean. Hauen tamaina proportzionala da geratzen diren iterazioak zati dauden hari kopuruekiko. Banaketa mota honetan *chunk* tamaina aukera daiteke modu honetan : *schedule(guided, chunk-size)*. Hona hemen 6.5 irudian *schedule* mota honen adibide bat.
- Auto : *Schedule* mota honetan, konpilatzaileari zein *schedule* mota erabiliko den erabakitzeko ardura ematen zaio.
- Runtime: *Schedule* mota honetan banaketa **OMP_SCHEDULE** ingurune-aldagaiaren

```
schedule(guided):
                            *********                        *
                ************                     *******  ***
                                     *******                   *
****************                             *****       **    *
```

```
schedule(guided, 2):
                ************                     ****     **
                                     *******         ***    **
                            *********
****************                             *****       **    **
```

6.5 Irudia: *guided* banaketa motaren eskema

eskura uzten du. Mota hau asko erabiltzen da programazio garaian zein banaketa mota erabiliko den ez dakienean, erabakia exekuzio garairako utziz.

6.1.1 Paralelizazioarekin hasi aurretik egin beharrekoa

Kodea paralelizatu aurretik jatorrizko kodean aldaketa eta analisia egin behar dira honen portaera jakiteko. Hau kontuan izanda, paralelizazioa hasi aurretik kodearen garbiketa egin da kodearen klase nagusia txukuntzeko asmoz. Beste aldetik, kodearen exekuzioari *profiling* bat egin zaio. Azkenik, kodearen analisi sakon bat egin da zatien arteko dependentziak aztertuz.

Kodea garbitu

Jatorrizko kodearen garbiketa sinple bat egiteko asmoarekin **io** izeneko klasea sortu zen. Klase hori, aurreko ataletan azaldu bezala, simulazioko parametroen kargarako diseinatutako klasea da. Klase hau sortzeko arrazoia kodearen garbiketa izan zen; klasea sortu baino lehen karga guztiak **main** klasean egiten ziren. Karga hauek klase nagusian egitean kodea gehiegi luzatzen zen. Hori izan da arrazoia **io** klasea sortzeko. Dena den, **io** klasearen sorkuntzak ez du izan exekuzio-abiaduran inongo eragin bereizgarririk.

Profiling

Paralelizazioa hasi aurretik, *Profiling* moduko bat egin zen kodean, jakiteko zein zen simulatzailearen exekuzio-fluxua, exekuzio-fluxuaren zatiek exekuzio osoaren zenbat denbora hartzen zuten, eta horietatik zeintzuk paralelizatu zitezkeen. Analisi hori egin ondo-

ren ikusi zen exekuzioaren denbora gehien hartzen zuten zatiak **load_equations**, **solver** eta **sections** klaseetako funtzioak zirela.

Dependentziak aztertu

Behin kodea garbitu denean eta profiling-a aplikatu zaionean, kodea aztertu behar da exekuzio zatien dependentziak bilatzeko. Kodea aztertzerakoan, *section* sasi-aginduaren erabileraren bideragarritasuna aztertu da. Modu honetan, hariek iterazio baten atal ezberdinak egingo dituzte aldi berean, eta horrela iterazioen exekuzio-denbora jaitsiko da. Kodearen exekuzio-fluxua aztertzerakoan, ikusten da dependentzia ugari daudela iterazioen barruan. Beraz, *section* klausulak ezin dira erabili, atazak aldi berean ezin direlako exekutatu, haien arteko dependentziak direla eta. Dependentziak aurkitzea ez da erraza izan, funtzio deietan *armadillo*-ko sintaxia duten funtzioak erabiltzen direlako eta askotan zaila izaten da hauen eginkizuna zein den jakitea, gutxiago ere simulatzaile konplexu batean. Hau ikusita, ezin izango dira kalkuluak aldi berean egin bata besteak sortutako emaitzaren beharra duelako; beraz, kodea paralelizatzeko beste tokitik hasi beharko da.

6.1.2 Paralelizazioko irizpideak

Oinarri lerroa

Paralelizatzerako garaian, konparaketak egiteko eta hobekuntzak ikus ahal izateko oinarri-lerro bat ezarri behar da. Kasu honetan, oinarri-lerroa simulatzailearen serieko bertsioa izango da. Modu honetan, proiektuan zehar egindako bertsio guztietan, oinarri-lerroa serie bertsioa izango da.

Konparaketetako erabilitako neurriak

Aurreko atalean, simulatzailearen serieko bertsioa oinarri lerroa izango dela azaldu da. Hala ere, konparaketak egiteko zein neurri kontuan hartuko diren erabaki behar da. Gure kasuan, konparaketak egiteko, exekuziorako erabili diren **nukleo kopurua**, **azelerazio-faktorea** eta **eraginkortasuna** hartuko dira kontuan. Modu honetan, bertsioen arteko konparaketak garbiak izango dira.

Lortutako exekuzio denborak-hiru saiakeren arteko minimoak dira; probak 200, 500 eta

1000 iterazioekin egingo dira. Proba guztiak makina berean egingo dira eta baldintza berdinetan.

6.1.3 Kodearen OpenMP-ko lehen bertsioa

Kodearen OpenMP-ko lehenengo bertsioa *Ryzen 3900x* prozesadorea duen ordenagailuan exekutatu da. Ordenagailuak *Kubuntu 20.04* sistema eragilea du. Bertan *Armadillo 9.900* eta *MKL* softwareen azkeneko bertsioak erabili dira. Kodearen idazketarako *Atom* programa erabili da.

Kodearen bertsio honetan, aurretik azaldu bezala iterazio barruko kalkuluak dependentzia dutenez, kalkuluak aldi berean egin beharrean, kalkuluetako barneko funtzioak paralelizatu dira. Horrela, nahiz eta kalkuluak aldi berean ezin diren egin, kalkuluak egiteko behar den denbora gutxitzen da. Modu honetan, egindako profilingari esker; denbora gehien behar duten kalkuluak identifikatu eta paralelizatu dira metodo ezberdinekin.

Load_equations zatian, kalkuluetarako matrizeak hasieratzen dira; ondoren, *armadillo*ko sintaxiko hainbat aljebra linealeko kalkulu egiten dira 4 dimentsioko begiztetan. Beraz, OpenMP-ko *#pragma omp parallel for* sasi-agindua erabili da kanpoko begizta harietan banatzeko. Modu honetan, matrizeetako kalkuluak modu paraleloan egiten dira, eta aldagai guztiak partekatuta daudenez, ez da arazorik gertatzen matrizeen kalkuluak egitean.

Solver zatian, *armadillo*ren bitartez matrizeak sortzen dira eta matrize horiekin hainbat fasetako kalkuluak egiten dira. Kalkuluaren lehengo fasea 4 dimentsioko begizta bat da, beraz, OpenMP-ko *#pragma omp parallel for* sasi-agindua erabili da kanpoko begizta harietan banatzeko. Kalkuluko bigarrengo eta hirugarrengo faseetan ere, 4 dimentsioko begiztak paralelizatu dira OpenMP-ko *#pragma omp parallel for* sasi-agindua erabiliz.

Sections zatian, *double* aldagai ugariren kalkulua egiten da hainbat begizten artean. Lehengo kalkuluan, **abs_secx**, **abs_secy** eta **abs_secz** aldagaien kalkuluak egiten dira 4 dimentsioko begitza batean. Kalkulu horietan aipatutako aldagaien gainean batuketak egiten direnez, OpenMP-ko *#pragma omp parallel for reduction(+:abs_secx,abs_secy,abs_secz)* sasi-agindua erabili da. *reduction* klausula erabiltzean hari bakoitzak bere kalkulua egiten du eta begizta amaitzerakoan batura guztia egiten da. Modu honetan denbora gutxiago behar da kalkulu hori egiteko. Bigarrengo kalkuluan, **scat_secx**, **scat_secy** eta **scat_secz** aldagaien kalkulua egiten da eta aurrekoan bezala *reduction* klausula erabili da kalkuluak egiteko denbora guxtiago behar izateko. Hirugarren kalkuluan, aurreko bien antze-

ra **ext_secx**, **ext_secy** eta **ext_secz** aldagaiak kalkulatzen dira eta honetan ere *reduction* klausula erabili da.

Bertsio honetan egindako aldaketekin, simulatzailearen jatorrizko bertsioarekin konparatuta, emaitza eta exekuzio denbora hobeagoak ematen ditu paralelismoaren gehikuntzarengatik. Paralelismoko bertsio hau simulatzailearen barneko ataletan egiten da. Modu honetan, probak egin dira **200** iterazioekin eta *Ryzen 3900x* prozesadoreko nukleo kopuru ezberdinak erabiltzen. Probak egiterakoan *SMT* teknologia desgaitu da; beraz, 12 hari erabiliko dira gehienez; horrez gain, AMD-ren *Percision Boost* teknologia desgaitu da nukleoen erloju-maiztasunen abiadurak berdinak izateko; guztiek 3,8 GHz-ko abidaura mantentzen dute. Probetan **Azelerazio-faktorea** eta **Eraginkortasuna** kalkulatu dira erabili diren hari kopuru ezberdinetarako. **Azelerazio-faktorea** eta **Eraginkortasuna** 4.1 eta 4.2 ekuazioetan ikus daitezke, hain zuzen ere. Emaitzak 6.1 taulan eta 6.6, 6.7 irudietan ikus daiteke.

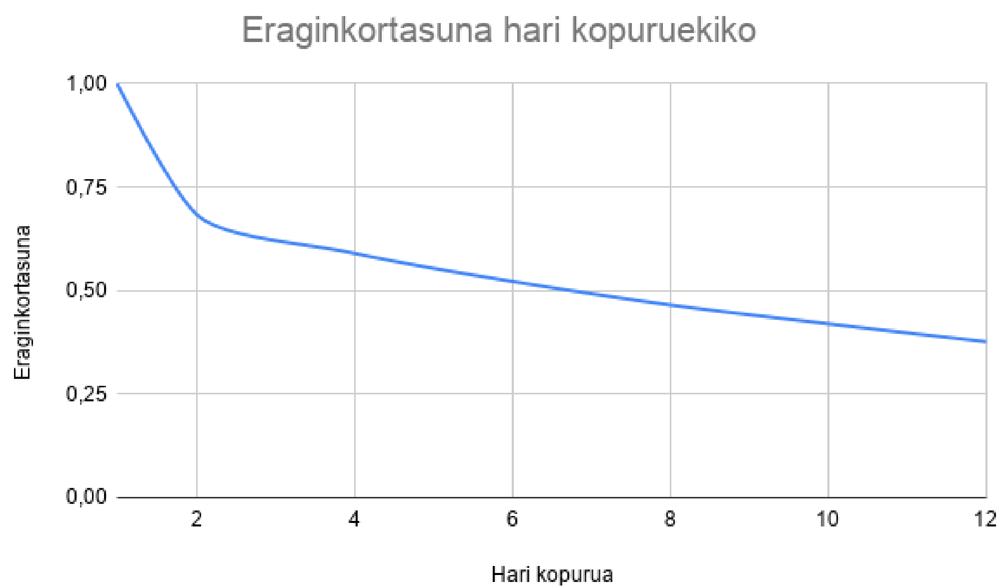
Hari kopurua	Exekuzio denbora (s)	Azelerazio-faktorea	Eraginkortasuna
1	890	1	1
2	650	1,36	0,68
4	377	2,36	0,59
6	284	3,13	0,52
8	239	3,72	0,47
10	212	4,20	0,42
12	197	4,52	0,38

6.1 Taula: OpenMP-ko lehen bertsioaren 200 iterazioko exekuzio denboren taula

6.6 irudian ikus dezakegunez, egindako paralelizazioarekin hari kopurua igotzen den bitartean azelerazio-faktorea ere igotzen joaten da. Beraz, honek esan nahi du programa nahiko eskalagarria dela, nahiz eta amdhalen idealetik urrun egon.

6.7 irudian ikus dezakegunez, eraginkortasun maila jaisten joaten da prozesadore gehiago erabiltzen dugun heinean. Hala ere, beherakada handiena serieko bertsioa eta 2 harien artean dago, honen arrazoi nagusia paralelismoko gainkarga da. Kodea paralelizatzerakoan, gainkarga bat sortzen da, eta kasu honetan 2 hari bakarrik erabiltzen ditugunez, sortutako gainkarga erabilitako nukleoa baino handiagoa da eta beraz, exekuzio-denbora motelago du. 2 haritik aurrera, eraginkortasunaren malda jaisten da; beraz, esan dezakegu 2 haritik aurrera eraginkortasun gutxiago galtzen dela.

6.6 Irudia: Azelerazio-faktorea hari kopuruekiko



6.7 Irudia: Eraginkortasuna hari kopuruekiko

6.1.4 Kodearen OpenMP-ko bigarren bertsioa

Proiektuaren hurrengo pausoa paralelizazio maila altuago bat lortzea da. Paralelizazio maila altua lortzeko hainbat proposamen aztertu dira. Proposamenen artean, hoberena, simulatzailearen begizta nagusia paralelizatzea izan da.

Kodearen bertsio honetarako, kodea berriz aztertu da eta kodea azkartzeko aukera bat ikusi da simulatzailearen begizta nagusian. Begizta nagusian, iterazioak banatzen dira eta beraz, begizta hau paralelizatu ahal bada, iterazioen exekuzio osoa paraleliza daiteke. Modu honetan, hari bakoitzak iterazio oso bat exekutatzen du, kalkulu zehatz batzuk bakarrik exekutatu beharrean. Aukera hau bideragarria dela ikusterakoan, kanpoko begiztaren paralalelizazioaren prozesua hasi da. Prozesua hasterakoan iterazioen analisia egin da eta iterazioak elkarrengandik guztiz independenteak direla aztertu da. Begitza nagusia paralelizatzeko, OpenMP-ko *#pragma omp parallel for* sasi-agindua erabili da iterazioak harien artean banatzeko eta bakoitzak bere iterazioak exekutatzeko. Honela, kodearen zati handiagoa paralelizazten da eta beraz, exekuzio-denbora jaitsiko dela aurreikusten da.

Begizta nagusiaren paralelizazioa egiterakoan arazo bat gertatu da. Begizta nagusian **double** motako aldagai bat erabiltzen da indize moduan. Indize honek, exekuzioaren iterazioa zehazten du eta honen balioa λ aldagaian godetzen da, hau da, iterazioa uneko λ balioa da eta λ honen balioa aurretik konfiguratutako λ_{min} baliotik hasten da eta λ_{max} baliora iristen da, azkenengo honen kalkulua egin gabe. Arazoa OpenMP-ko for sasi-aginduarekin gertatzen da, honek ez duelako onartzen **double** motako aldagaiak erabiltzea. Arazo hau konpontzeko kodean aldaketa batzuk egin dira. **Integer** motako aldagaia sortu da λ-ren balioa gordeko duena. Aldagai berri hau indize bezala erabiliko da begizta nagusian OpenMP-ko *for* sasi-agindua arazoa ez emateko. Hala ere, iterazioko kalkuluetarako **double** motako aldagai bat behar da λ-rekin zerikusia duten kalkulu guztietarako. Hau kontuan izanda, erabaki da iterazioetako hasieran **double** motako aldagaiari indizearen balioa esleitzea. Modu honetan, nahiz eta zenbaki osoa izan, aldagaia **double** formatua izango du eta, beraz, iterazioko kalkuluetan ez du arazorik emango. Hau egitean ordea, simulatzailean muga bat jartzen zuen, λ-ren balio guztiak osoak izango direlako eta beraz, ezin izango dira zenbaki arrazionalak erabili kalkuluetako.

Behin kanpoko begitza paralelizatu denean, proba ugari egin dira *schedule* mota hoberena lortzeko. Kasu honetan garrantzi handia du *schedule* klausula egokia erabiltzeak, begiztaren iterazioen lan-karga aurreko bertsioarekin konparatuta askoz handiagoa delako. Beraz, kasu honetan iterazioen banaketa nola egiten den kontuan izan behar da denbora aurrezteko. Hau kontuan izanda, probak egin dira *schedule* mota guztiekin 12 hari erabilita eta 200

eta 500 iterazioko exekuzioekin. Lortutako exekuzio denborak 6.2 taulan ikus daitezke.

Schedule mota	exekuzio denbora (s) 200 it.	exekuzio denbora (s) 500 it.
static	116	281
dynamic	115	288
auto	117	282
guided	112	280
runtime	113	284

6.2 Taula: *schedule* motak 24 hariekin lortutako exekuzio denboren taula

Bi tauletan ikus dezakegunez, *schedule* mota guztietatik *guided* da emaitza hoberenak ematen dituena. 200 iterazioko exekuzioan, *guided* motak denbora hoberena lortu du. Atzetik, *dynamic*, *auto* eta *runtime*. Azkenik, *static* mota dugu. Kasu honetan, iterazio gutxi daudenez, *schedule* mota ezberdinen arteko exekuzio-denborak ez dira oso handiak, dagoen diferentzia handiena 2 segundoko da. Hala ere, ikusten da *guided* kasuan emaitza onena lortzen dela. Oro har, 200 iterazioko exekuzioaren kasuan iterazio gutxi dira *schedule* motak exekuzio denboran garrantzi asko izateko.

500 iterazioko exekuzioen kasuan, *schedule* moten arteko diferentzia nabarmenagoa da 200 iterazioko kasuarekin konparatuta; hala ere, orokorrean *schedule*n arteko diferentzia ez da oso nabarmena. 500 iterazioko kasuan ere *guided schedule* motak exekuzio denbora azkarrena lortzen du. Honen atzetik, *static* eta *runtime* ditugu . Azkenik, *dynamic* eta *auto schedule* motak izan dute exekuzio-denbora motelenak. 500 iterazioko kasuan, 200 iteraziokoan bezala *guided* da emaitza hoberenak ematen dituen *schedule* mota. 200 iterazioko kasuan, lortzen den exekuzio-denbora tartea ez da hain handia *schedule* mota ezberdinak kontuan izateko. 500 iteraziokoan ere, nahiz eta diferentzia-tartea handiagoa den; 8 segundoko gehienez, ez da oso handia erabilitako iterazio kopuru eta harietarako. Beraz, honekin ondoriozta dezakegu, nahiz eta diferentzia asko ez izan, simulatzaile honen kasurako, *guided schedule* mota dela egokiena eta honen eragina nabarmenagoa dela iterazio kopurua altuak direnean. Honen arrazoia, *guided*-en portaera da, *guided* banatu beharreko iterazioen lan-kargen arabera, hariei *chunk* tamaina diferenteko *chunk*-ak bidatzen zaizkie. Kontuan izanda iterazio guztiek lan karga berdina dutela, *chunk* tamaina kalkulatzea erraza da eta beraz banaketa orekatua lortzen da.

Behin erabakita zein *schedule* mota erabiliko den, proba gehiago egin dira aurreko bertsioan bezala hari ugariekin eta iterazio ezberdinekin exekuzioak egiten. Modu honetan, bigarrengo bertsio honetan aurrekoarekin konparatuta exekuzio denbora hobeagoak ematen dituen aztertuko da. Horretarako, probak 200, 500 eta 1000 iterazioko exekuzioak izango dira. Exekuzio denborak 6.3 taulan ikus daitezke; azelerazio-faktorea eta eragin-

kortasuna aldiz, 6.4 taulan ikus daitezke. Gainera, tauletako edukia 6.8, 6.9 eta 6.10 irudietan ikus daiteke.

Hari kopurua	Exek. denbora (s) 200 it.	Exek. denbora (s) 500 it.	Exek. denbora (s) 1000 it.
1	890	2165	4370
2	612	1506	3057
4	304	755	1533
6	208	512	1031
8	162	394	786
10	130	322	653
12	119	280	556

6.3 Taula: OpenMP-ko bigarren bertsioaren exekuzio denboren taula

Hari kop.	Af 200 it.	Af 500 it.	Af 1000 it.	Erag. 200 it.	Erag. 500 it.	Erag. 1000 it.
1	1	1	1	1	1	1
2	1,45	1,43	1,43	0,73	0,72	0,71
4	2,93	2,87	2,85	0,73	0,72	0,71
6	4,28	4,22	4,24	0,72	0,71	0,71
8	5,49	5,49	5,56	0,69	0,69	0,69
10	6,85	6,72	6,69	0,69	0,67	0,67
12	7,48	7,73	7,86	0,62	0,64	0,65

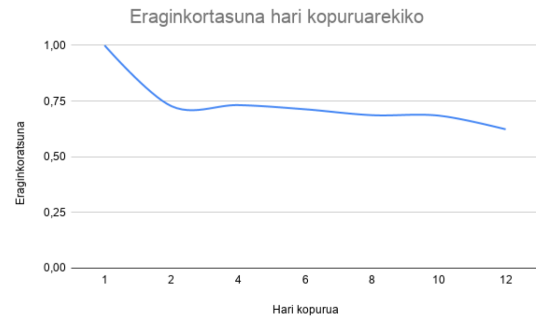
6.4 Taula: OpenMP-ko bigarren bertsioaren aelerazio-faktore eta eraginkortasunaren taula

Emaitza guztiak aztertzerakoan, proiektuaren bertsio honetan exekuzioak azkarragoak izan direla ikus dezakegu. Hiru iterazio kopuru ezberdinetan, azelerazio-faktorearen kurbak antzekoak dira. Beraz, esan genezake simulatzailearen konputazio lan-karga linealki eskalagarria dela iterazio kopuruarekiko. Bigarrengo bertsio honetan, azelerazio-faktore maximoa 12 hariekin lortzen da. Eraginkortasunaren aldetik, 200, 500 eta 100 iterazioko exekuzioak antzeko eraginkortasun-kurba dute, seriko bertsiotik 2 harira eraginkortasuna dezente jaisten da, honen arrazoia iterazio osoa hari bakar batek egitean esleitzen zaion lan karga da. Bestalde, 4 haritik 12 harira eraginkortasun-kurbaren malda ez da oso altua. Beraz, hari gutxi erabiltzen direnean eraginkortasuna jaisten da harirekiko esleitzen den lan-karga handia delako eta ondorioz, eraginkortasuna mantentzeko 2 hari baino gehiago erabili behar dira lan-kargaren banaketa orekatzeko.

6.1.5 Proiektuaren OpenMP-ko hirugarren bertsioa

Proiektuaren fase honetan, bigarrengo bertsioan aldaketak egiterakoan jarritako muga konpondu zen, hau da, bigarrengo bertsioan begizta nagusia paralelizatzerakoan *double*

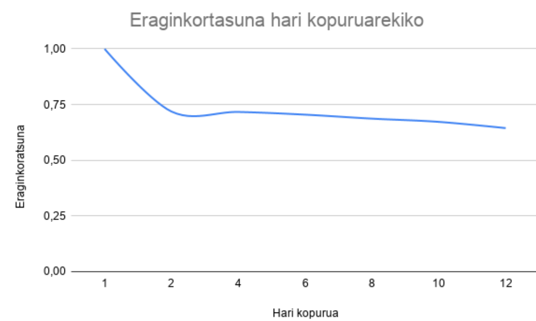
(a) Azelerazio-faktorea hari kopuruarekiko



(b) Eraginkortasuna hari kopuruarekiko

6.8 Irudia: 200 iterazioko grafikak

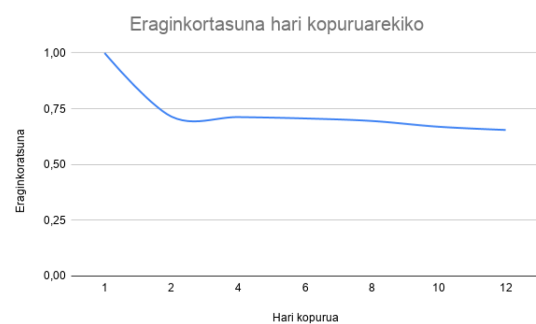
(a) Azelerazio-faktorea hari kopuruarekiko



(b) Eraginkortasuna hari kopuruarekiko

6.9 Irudia: 500 iterazioko grafikak

(a) Azelerazio-faktorea hari kopuruarekiko



(b) Eraginkortasuna hari kopuruarekiko

6.10 Irudia: 1000 iterazioko grafikak

indizerik ezin zenez erabili, *integer* indizea erabili zen. Modu honetan, indizeen inkrementuak osokoak izango ziren eta beraz, kalkulatu beharreko λ tartea eta iterazio kopuruaren zatiketak osokoa izan behar zuen.

Hau guztia kontuan hartuta, bertsio honetan osoko inkrementuen muga ezabatu zen kodean hainbat aldaketa egiten, baina aldi berean begizta nagusiko paralelizazioa mantentzen. Egindako aldaketen artean hauek dira garrantzitsuenak.

1. Lehenik, aldagai bat sortzen da eta bertan iterazio eta hari kopuruen zatiduraren balio esleitzen zaio.

2. Kodearen zati paraleloa hasten da; horretarako, *#pragma omp parallel private (lambda)* jartzen da adieraziz hari bakoitzak bere λ izango duela.

3. **Parameters** objektuaren hasieraketa egiten da. Horrez gain, *Integer* motako aldagai bat sortzen da eta bertan hari bakoitzak erabiliko duten bektoreen luzeraren balio gordetzen da.

4. Zatiketan geratutako hondarra kontuan izanda, hari bakoitzak kalkulatuko duen lehenengo λ eta bektorearen luzera kalkulatzen dira. Hondarra geratzen bada, iterazioak harietara modu orekatuan banatzen dira.

5. **Double** motako bektorea sortzen da, aurretik kalkulatutako luzerakoa. Bektorearen lehenengo balioa hari horrek kalkulatu behar duen lehenengo λ balioa izango da. Hurrengo balioak begizta batean kalkulatzen dira, bertan bektorearen aurreko posizioko aldagaiari inkrementua gehituz lortzen da.

6. Begizta nagusia hasten da; bertsio honetan, *Integer* motako aldagaia erabiltzen da eta begiztatik irteten da bektoreko luzeraren baliora iristerakoan.

7. Begiztaren hasieran uneko λ-ren balioa bektoretik lortzen du, begizta nagusiko indizea erabiliz.

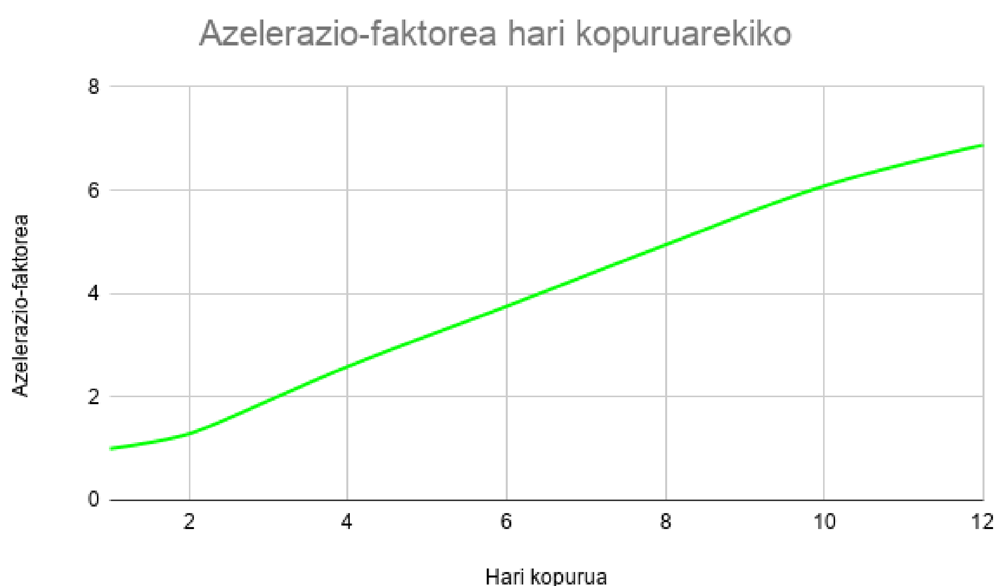
8. Gainontzeko funtzionamenduak berdin jarraitzen du.

Aldaketa guztiak egiterakoan, simulatzaileak osokoak ez diren inkrementuak onartzen ditu eta beraz λ kasu gehiago kalkula daitezke. Hau egitean ordea, aurreko bertsioko *schedule* mota galtzen da, orain banaketa eskuz egin behar delako. Beraz banaketa estatikoa egin behar da. Hala ere, kontuan izanda aurretik badakigula iterazioen lan-karga oso antzekoa dela, banaketa estatiko bat egiteak ez du asko kaltetzen exekuzio denbora.

Bertsio honetako exekuzio-probak 500 iteraziorekin egin dira. Hirugarrengo bertsioaren exekuzio denborak 6.5 taulan eta 6.11, 6.12 irudietan ikus daiteke.

Hari kopurua	Exekuzio denbora (s)	Azelerazio-faktorea	Eraginkortasuna
1	1953	1	1
2	1519	1,29	0,64
4	754	2,6	0,65
6	520	3,76	0,63
8	395	4,94	0,62
10	321	6,08	0,61
12	284	6,88	0,57

6.5 Taula: OpenMP-ko Hirugarren bertsioaren 500 iterazioko exekuzio denboren taula

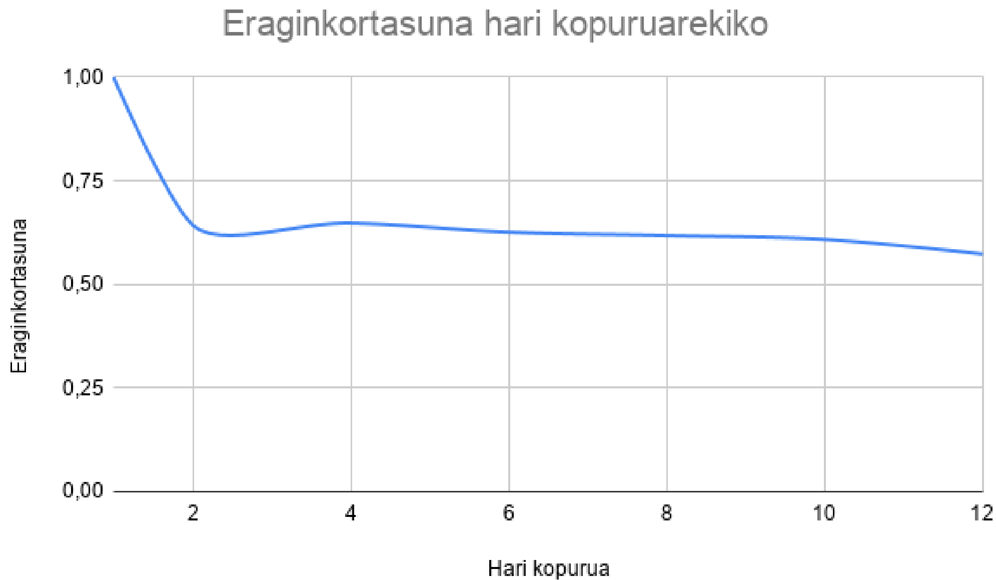


6.11 Irudia: Azelerazio-faktorea Hirugarren bertsioa (500 iterazio)

6.1.6 Bertsioen arteko konparaketak

Aurreko ataletan, hiru bertsioetan egindako hobekuntzak azaldu dira; orain hiru bertsioak konparatuko dira exekuzio-denboretan zein eragin izan duten ikusteko. Konparaketa ondo ulertzeko, labur azalduko da hiru bertsioetan egindako hobekuntzak.

Lehen bertsioan, jatorrizko kodea hartu da eta aurretik egindako profilingari esker kodean denbora gehien erabiltzen zuten funtzioak bilatu ziren eta hauek paralelizatu ziren.



6.12 Irudia: Eraginkortasuna Hirugarren bertsioa (500 iterazio)

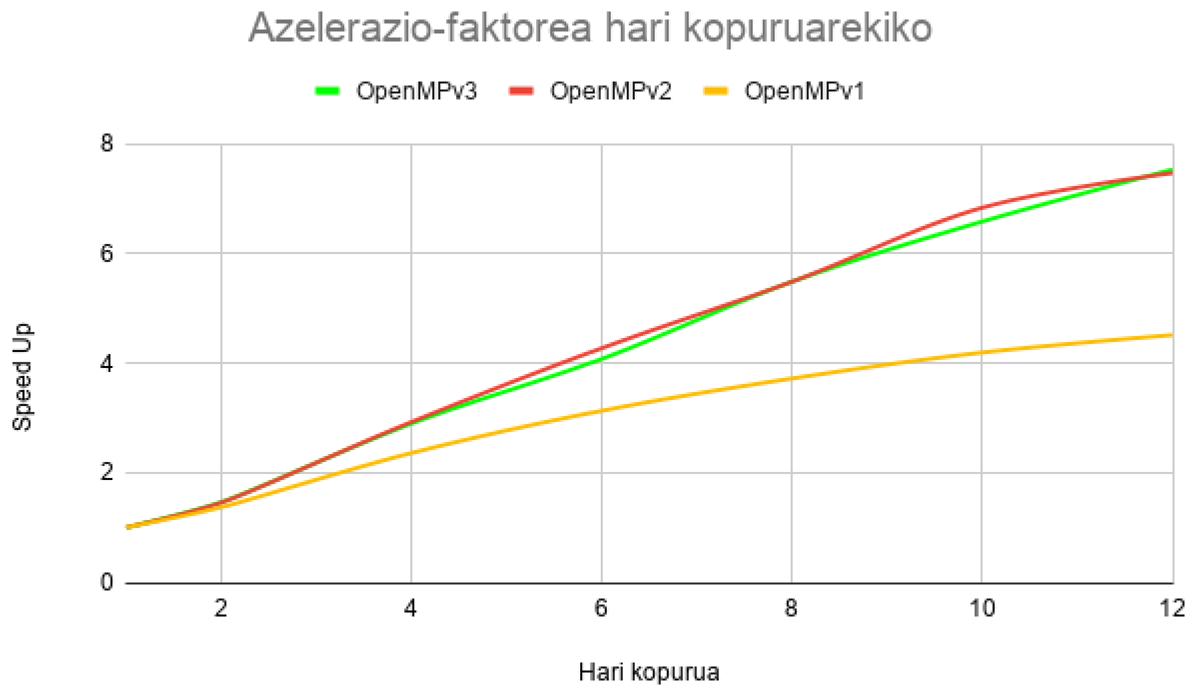
Bigarren bertsioan, hariei lan karga gehiago jartzeko asmoarekin simulatzailearen begizta nagusia paralelizatu zen, *guided schedule* mota erabiliz. Modu honetan, hari bakoitza iterazio osoaren kalkulua egingo zuen, kalkulu baten zati bat egin beharrean. Hau egitean ordea, **OpenMP**-ren bateragarritasun arazoengatik; indizea aldatu zen **double** izatetik **integer** izatera.

Hirugarren bertsioan, aurreko bertsioko arazoa konpondu zen. Horretarako, eskuz egindako banaketa bat egin zen hari bakoitzak jakiteko zein iterazio kalkulatu behar dituen. Modu honetan, **OpenMP**-ren bateragarritasun arazoa desagertu zen *parallel for* sasi-agindua ez zelako erabiltzen.

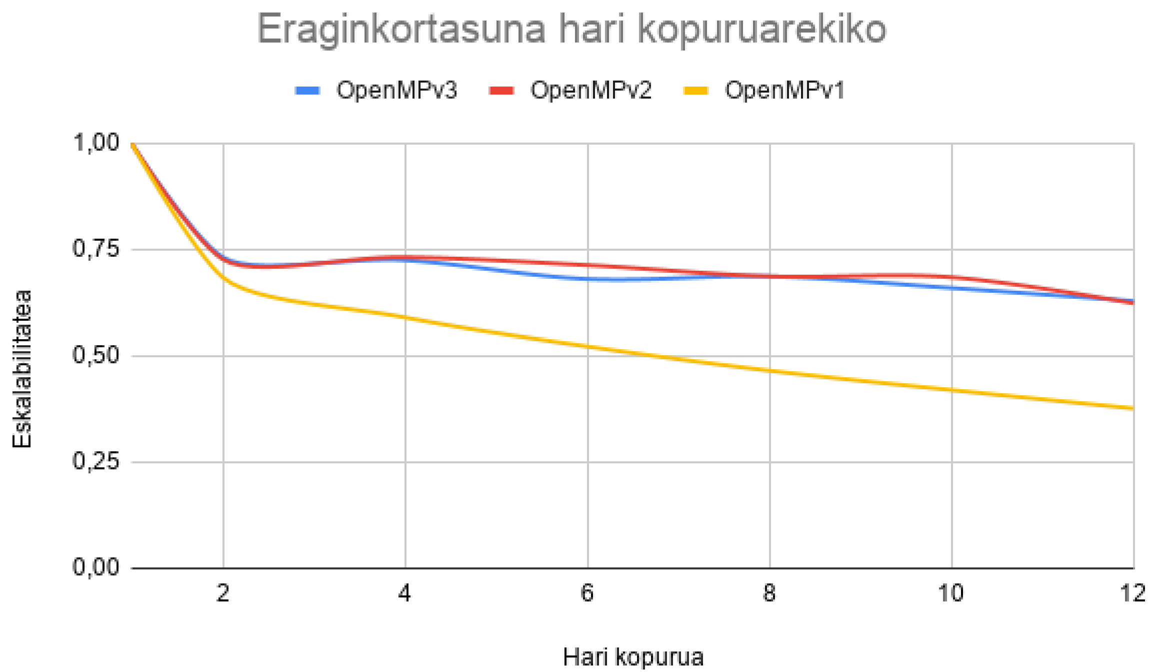
Behin hiru bertsioak laburtuta, haien arteko konparaketa egingo da. Horretarako, Hiru bertsioetako 200 iterazioko exekuzio denborak erabiliko dira konparaketa egiteko. Konparaketa 6.13 eta 6.14 irudietan ikus daiteke.

6.13 eta 6.14 irudietan ikus daitekeen moduan, hiru bertsioek portaera oso berdintsua dute. Hala ere, nabarmena da bigarren eta hirugarren bertsioetan lortutako hobekuntzak exekuzio denboretan lehenengoarekin konparatuta. Beraz, ideia ona izan zen begizta nagusia paralelizatzea, azken finean, hau egitean kodearen zati gehiago paralelizatzen zelako.

Bestalde, esan genezake bigarren eta hirugarrengo bertsioak oso parekoak direla. Bigarrengo bertsioa da azelerazio-faktore handienak ematen dituena. Hala ere, kontuan izanda bigarrengo bertsioak duen desabantaila esan genezake hiru bertsioetatik onena hiruga-



6.13 Irudia: Azelerazio-faktorea hiru bertsioak (200 iterazio)



6.14 Irudia: Eraginkortasuna hiru bertsioak (200 iterazio)

rrengoa dela, begizta nagusia paralelizatzen delako eta gainera **double** motako iterazioak kalkulatu daitezkeelako.

6.2 OpenMPI

Behin **OpenMP**-rekin ikusitako aldaketa posible guztiak eginda, kodea **OpenMPI**-ra pasatzea erabaki da. Horretarako, oinarri bezala **OpenMP**-ko hirugarrengo bertsioa hartu da, eta kontzeptu bera mantendu da **OpenMPI**-ko bertsiorako. Hala ere, kontuan izan behar da **OpenMPI**-n aparteko aldaketak egin behar direla kodea funtzionatzeko.

OpenMPI OpenMP bezala, paralelismorako aplikazioen programazio interfaze (API) bat da. **OpenMPI** ordea, memoria pribatuan oinarritzen da. **OpenMPI** *C*, *C++* eta *Fortan* lengoaietan funtzionatzeko diseinatuta dago eta hainbat plataforma eta arkitektura desberdinetan funtzionatzeko prest dago, *Unix*, *GNU/Linux* eta *Windows* plataformetan adibidez.

Kodearen **OpenMPI**-ko bertsioa egiterakoan, aldaketa batzuk egin dira kodearen funtzionalitatea mantenduz. Aldaketa horien artean, identifikadoreen funtzioak daude. **OpenMP**-ko eginkizun bera dute, hau da, exekuzioan parte hartzen duen hari bakoitzaren identifikadorea eta exekuzioa parte hartzen duten hari kopurua. Horretarako, kodearen zati paraleloa hasterakoan ondorengo sasi-aginduak erabili dira: *MPI_Init (&argc, &argv)* zati **OpenMPI**-ko zatiaren hasiera adierazteko; ondoren, *MPI_Comm_rank (MPI_COMM_WORLD, &thread)* sasi-aginduarekin hari bakoitzaren identifikadorea lortzen da *MPI_COMM_WORLD* komunikatzaile globalarekin; eta azkenik, *MPI_Comm_size (MPI_COMM_WORLD, &threads)* sasi-agindua erabili da exekuzio osoan parte hartu duten harien kopurua jakiteko. Azkenik, iterazio guztiak amaitzen direnean; hesi bat jartzen da exekuzio denbora kalkulatu aurretik, hari guztiak amaitu dutela bermatzeko. erabilitako hesiaren sintaxia ondoregoa da: *MPI_Barrier (MPI_COMM_WORLD)*.

OpenMPI memoria pribatuan oinarrituta dagoenez, kodean erabilitako aldagaiak pribatuak dira, hau da, hari bakoitzak erabiliko diren aldagaien kopia pribatua dituzte. Honek esan nahi du, harien arteko komunikazioa gauzatu nahi bada **OpenMPI**-k eskaintzen dituen mezu sasi-aginduak erabili behar dira. Gure kasuan ordea, ez dago inongo komunikaziorik harien artean, bakoitzak hasieratik bukaeraraino iterazio osoa exekutatzen dutelako. Iterazioen banaketa **OpenMP**-ko hirugarren bertsioan bezala hari guztiek egiten dute, hau da, hari bakoitzak kalkulatzen du zein λ-tik zein λ-ra egingo dituen kalkuluak. Modu ho-

netan, harien arteko komunikaziorik ez dagoenez eta kalkuluetako erabiliko diren aldagai guztiak pribatuak direnez, ez da zaila izan kodea **OpenMP**-tik **OpenMPI**-ra pasatzea.

OpenMPI-ko bertsio bat egiteko arrazoiak ugari izan dira. Alde batetik, simulatzailea paralelismoko beste eredu mota batean nola funtzionatzen zuen izan da. Modu honetan, makina eta simulatze bera erabilita, bi ereduetatik gehien zein moldatzen den ikusiko da. Beste aldetik, simulatzeailearen funtzionamenduak kodea **OpenMPI**-ra modu sinple batean pasatzea ahalbidetzen du, batez ere, aldagai guztiak pribatuak izatea eta harien arteko komunikaziorik ez egotea asko errazten du kodearen eraldaketa.

Hau guztia esanda, hemen dira **OpenMPI** erabilita lortutako emaitzak. Emaitzetarako neurriak, **OpenMP**-rekin erabilitako berdinak dira, eta hauek ondorengoak dira: exekuzio denbora, azelerazio-faktorea eta eraginkortasuna. Bestalde, probetarako 500 iterazio erabili dira; horrez gain, *SMT* teknologia desgaitu da eta nukleo guztiak 3,8 GHZ-ko maiztasunera jarri dira proba hauetarako, eta berriro ere, hiru proba egin dira kasu guztietarako eta hiru probetatik lortutako exekuzio denbora minimoa hartu da. Beraz, hona hemen probetan lortutako emaitzak 6.6.

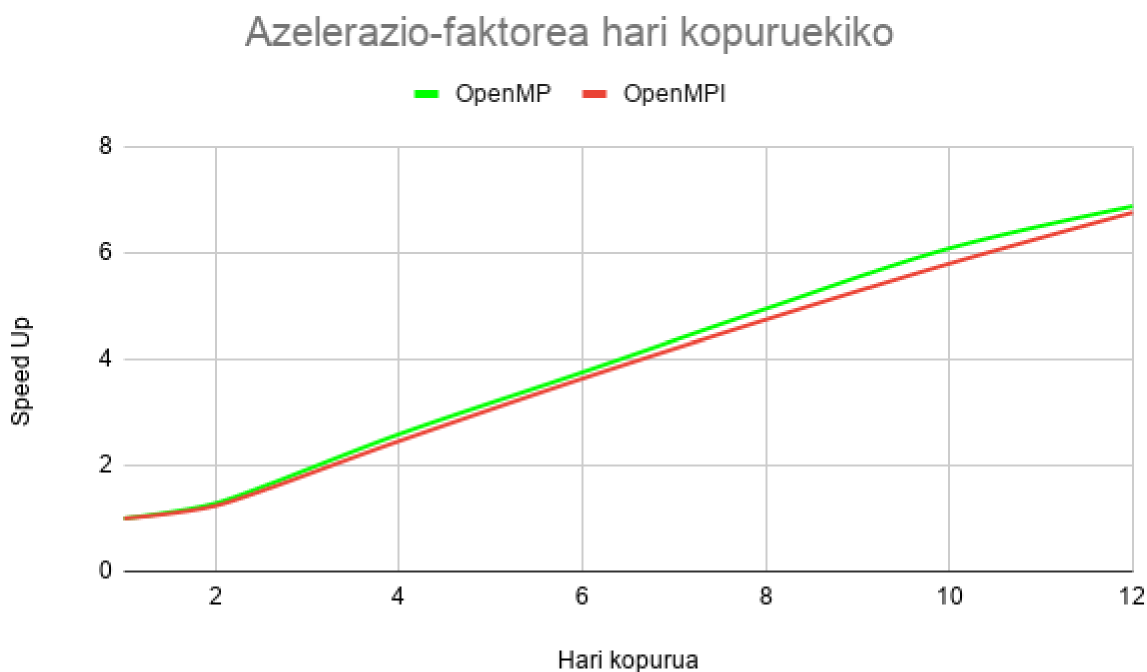
Hari kopurua	Exekuzio denbora (s)	Azelerazio-faktorea	Eraginkortasuna
1	1953	1	1
2	1578	1,24	0,62
4	794	2,46	0,61
6	537	3,64	0,61
8	412	4,74	0,59
10	337	5,8	0,58
12	289	6,76	0,56

6.6 Taula: OpenMPI-ko 500 iterazioko exekuzio denboren taula

6.2.1 OpenMP eta OpenMPI-ren arteko konparaketa

Aurreko bi ataletan, **OpenMP** eta **OpenMPI** ereduak azaldu dira. Bakoitzaren ezaugarriak azaldu dira; ondoren, eredu bakoitzean erabilitako sasi-agindu eta aldaketak azaldu dira. Azkenik, egindako hobekuntzetan lortutako emaitzak aztertu dira; **OpenMP**-ren kasuan, hiru bertsioen konparaketa egin da eta hirugarrengo bertsioa hiru bertsioetatik hoberena dela ondorioztatu da. Ondoren, **OpenMPI**-ko bertsioa azaldu da, bertsio hau **OpenMP**-ko hirugarrengo bertsioan oinarrituta dago eta nahiz eta eginkizun era duen *API* ezberdina erabiltzen du. Beraz, behin emaitza guztiak aztertuta, **OpenMP** eta **OpenMPI**-ko bertsioen konparaketa egitea falta da. Horretarako, aurreko ataletako azelerazio-

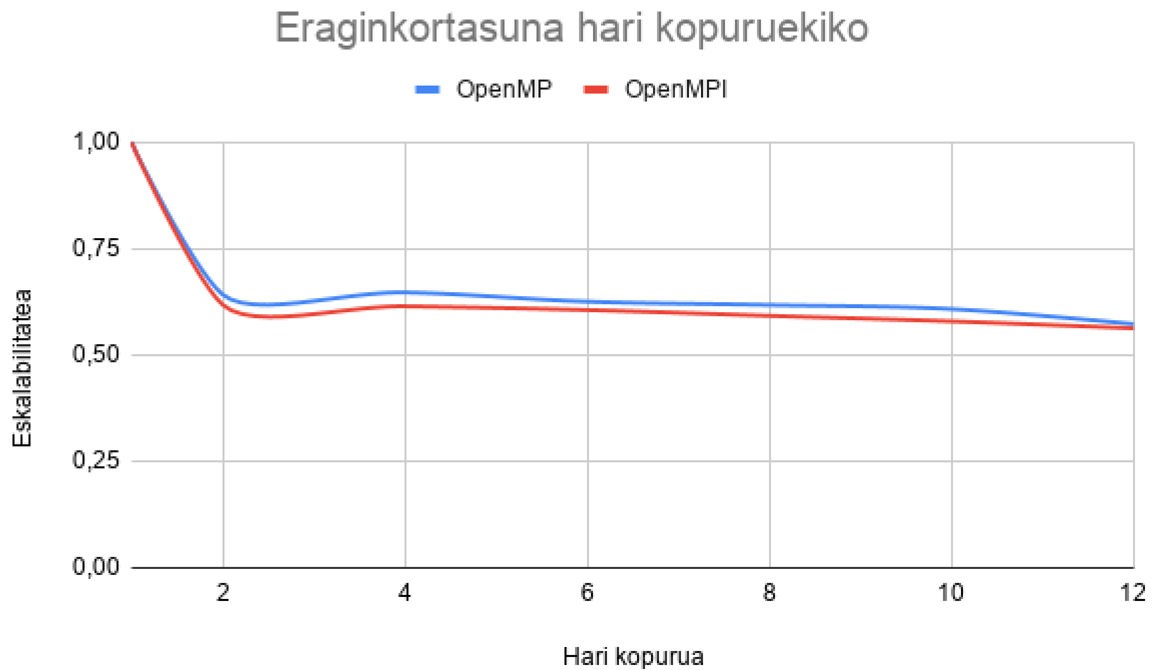
faktorearen eta eraginkortasunaren grafikak hartu dira eta grafika berri batean jarri dira konparaketarako. Konparaketan, 500 iterazioko exekuzioari dagozkion emaitzak hartu dira kontuan. Hau guztia esanda, hona hemen 6.15 eta 6.16 grafiketan, **OpenMP** eta **OpenMPI**-rekin lortutako emaitzenarteko konparaketa.



6.15 Irudia: Azelerazio-faktorea OpenMPI VS OpenMP (500 iterazio)

6.15 eta 6.16 irudietan ikus dezakegunez, **OpenMPI** erabilita **OpenMP**-ren emaitza oso antzekoak lortzen dira. Bi bertsioek portaera berdintsua dute, azelerazio-faktorea 4 harira arte ez da igotzen aurreko ataletan azaldu bezala, hari bakoitzak lan-karga asko duelako. Ondorioz, bi kasuetan, eraginkortasuna asko jaisten da 4 harietara iritsi arte. 4 haritik gora, azelerazio-faktorea linealki igotzen da eta beraz, eraginkortasuna mantentzen da.

Konparaketaren ondorio bezala esan dezakegu bi ereduak erabilita portaera berdintsuak ikus daitezkeela. Honen arrazoi nagusia, paralelizatzeko modua izan da, hau da, simulatzailearen iterazioak paralelizatu dira hainbat harien artean eta beraz, harien artean ez dago inongo komunikaziorik. Gainera, hari bakoitzak erabilitako aldagai guztiak pribatuak dira, eta harien arteko sinkronizazio mekanismo bakarra iterazio guztiak amaitzerakoan aplikatzen da. Modu honetan, **OpenMP**-ko bertsioak exekuzio-denbora baxuagoak lortzeko arrazoi nagusia probetarako erabili den makinaren arkitektura da. Probetan erabili den makina, nukleo anitzeko *CPU* bakarra izan da eta beraz, memoria partekatuan



6.16 Irudia: Eraginkortasuna OpenMPI VS OpenMP (500 iterazio)

oinarrituta dago. Beraz, zentzuzkoa da **OpenMP**-rekin emaitza hobeagoak izatea.

6.3 CUDA

Proiektuaren hasieratik, simulatzailea hiru ereduak erabilita aztertu nahi zen. Horretarako, hasieran **OpenMP**-ko bertsioaren garapena hasi zen. **OpenMP**-ko bertsioen garapena amaitzerakoan, aurreko ataletan azaldu bezala hiru bertsio sortu ziren, bakoitza aurrekoaren hobekuntza bezala proposatuta. Ondoren, behin **OpenMP**-rekin ezer gehiago lortu ezin zela ondorioztatzean, **OpenMP**-ko hirugarren bertsioa **OpenMPI**-ra pasa zen. Behin **OpenMPI**-ko bertsioa amaitzerakoan, **CUDA** eta **OpenMP** ereduak erabiltzen duen bertsio bat garatzea ere erabaki zen.

CUDA *(Compute Unified Device Architecture)* **Nvidia** etxeko konputazio paraleloko plataforma bat da. Plataforma honen bidez, *GPU*-aren bidez algoritmoak exekuta daitezke. Horretarako, *Nvidia*k konpilatzaile berezi bat sortu du *nvcc*, C eta C++-eko *g++* konpilatzailean oinarrituta. Horrez gain, *Nvidiak* aparteko tresnak sortu ditu programatzaileek *GPU*-ekin programatzeko aukera gehiago izateko. Nahiz eta **CUDA** C eta C++ lengoaie-

tako eginda dagoen, *wrapper*-en bidez beste hiztkuntzetan ere erabili daiteke; *Python*, *Fortran* eta *Java* adibidez.

Proiektuaren **CUDA**-ren bertsioaren garapena hasterakoan, aurretik espero ez ziren arazo ugari agertu ziren. Hasierako arazo nagusienetakoa **CUDA** eta **Armadillo**-ren bateragarritasun arazoak izan ziren. Armadillok, funtzionatzeko aljebra linealeko funtzioak exekutatzeko liburutegiak dituen plataforma bat behar du, **MKL** eta **OpenBLAS** adibidez. **OpenMP** eta **OpenMPI**-ren kasuan, intelen **MKL** liburutegi bilduma erabili da horretarako. **CUDA**-rekin ordea, ezinezkoa da **MKL** erabiltzea, hau *CPU*-rekin bakarrik erabili daitekeelako.

Hala ere, **Nvidiak cuBLAS** izeneko liburutegi bilduma garatu zuen; bilduma hau, **BLAS** liburutegi bildumaren inplementazio bat da. **cuBLAS** erabiltzerakoan, *GPU*-ak aljebra linealeko funtzioak erabili ditzake. Hala eta guztiz ere, Armadillok ez du onartzen oraindik **cuBLAS** eta beraz ezin da Armadillo erabili **CUDA**-rekin batera.

Aipatutako arazo horretaz aparte, jatorrizko simulatzailean erabiltzen ziren funtzio batzuk arazoak ematen zituen **CUDA**-ren konpilatzailearekin. Arazo gehien ematen klasea, C++ lengoiako *String* klasea zen. **CUDA**-k ez du onartzen datu mota hori eta beraz konpilatzaileak errorea ematen du hura erabiltzen denean. Hala eta guztiz ere, ezin da *String* datu mota kendu simulatzailetik; hark eskaintzen dituen funtzio asko erabiltzen direlako simulatzailean, eta hori guztia aldatzea funtzionalitate bera mantenduz oso zaila da eta gainera, denbora asko eskatzen du.

Dena den, *GPU*-aren erabilera oso aproposa ikusten zen, batez ere, simulatzailean matrize askoren gaineko kalkulu ugari egiten direlako eta *GPU*-ak kalkulu hauek egiten paregabeak direlako.

7. KAPITULUA

Ondorioak

Atal honetan proiektuaren ondorioak eta proiektuaren garapenak eragindako ondorio pertsonalak azalduko dira.

7.1 Proiektuaren ondorioak

Oro har, proiektuaren hasieran ezarritako helburuak bete dira, nahiz eta helburu batzuk ezin izan diren bete eta beste batzuk burutzea zaila izan diren. Proiektuaren helburu nagusia bete da, hau da, simulatzailea paralelizatu da modu eraginkor batean. Horretarako, konputazio paraleloko kontzeptuak bereganatu dira; gainera, paralelizazioko eredu ezberdinak erabili dira ikusteko zein egokitzen den gehiago simulatzailearen portaerarekin. Horrez gain, proiektuaren garapenean zehar bertsio ugari sortu dira bakoitza aurrekoaren hobekuntza bezala; eta orokorrean, guztietan lortutako emaitzak onak izan dira eta jatorrizko serieko bertsioarekin alderatuta hobekuntza nabarmenak lortu dira erabilitako bi ereduetarako, **OpenMP** eta **OpenMPI** hain zuzen ere.

Orokorrean, simulatzailea konplexua den arren, egindako analisiari esker honen portaera zein den jakin da, eta horren arabera egin dira bertsioak. Egia esan, simulatzailea parametro batzuen barnean erabili da, hau da, ingurune itxi batean erabili eta aztertu da. Simulatzaileak, hainbat konfigurazio parametroen arabera funtzionatzen du, eta gure proba guztietarako ingurune itxi horretan mantendu gara; honen adibide nagusia simulatzailean erabilitako partikulen tamaina da, hau aldatzean datuen tamainak aldatzen direlako, eta agian portaera aldatuko litzateke. Hau ordea, zaila da aztertzen, partikulen tamaina al-

datzeak memorian izugarrizko eragina duelako eta beraz, probetako erabili den makinak ezin du simulatzailea exekutatu.

Bestalde, nahiz eta **OpenMP** eta **OpenMPI** ereduekin lortutako bertsioen emaitzak onak diren, **CUDA**-ko bertsioa ezin izan da egin. **CUDA**k duen ahalmena kalkulu hauek egiteko haundia da baino hainbat arrazoirengatik ez da posiblea izan.

Azkenik, aipatu lortutako emaitzak benetan hobekuntza handia dakarrela jatorrizko bertsioarekin, batez ere azkeneko bertsioekin, azelerazio-faktorea linealki hasten delako eta beraz, ahalmen gehiagoko makinetan simulazio handiagoak exekuta daitezke baliabide gehiago erabili daitezkeelako.

7.2 Ondorio pertsonalak

Pertsonalki, proiektu hau garatzea oso baliagarria eta lagungarria izan da niretzat. Alde batetik, gaia oso interesgarria da, ikasturteetan ikasitako kontzeptuak "mundu errealeko"simulatzaile batean aplikatzeko aukera benetan interesgarria da. Bestalde, simulatzailea aztertzea eta arazoari konponbidea ematea duen zailtasuna gainditzea, erronka handia izan da niretzat.

Proiektuaren hasieratik, simulatzailea exekutatzeko beharrezkoak ziren modulu eta tresnak instalatzea arazoak eman dizkit. Ondoren, simulatzailea ulertzea zaila da eta are gehiago atzetik dagoen fisika arloko kontzeptuak ulertzea. Horrez gain, simulatzaileak dituen parametro eta klase ugarien eginkizuna ulertzea konplexua da, tartean *Armadilloko* sasi-aginduak erabiltzen direlako. *Armadillo* ez nuen ezagutzen proiektuaren garapenaren aurretik eta egia esan, funtzio batzuen eginkizuna jakitea zaila da, batez ere lehendik ez bada erabili. Makinari dagokionez, *Covid-19* pandemia dela eta etxeko ordenagailua erabili da exekuzio-probetarako eta ahalmentsua denez, egindako probak gogobetekoak izan dira. Bestalde, **DIPC**-ko super-konputagailuan probak egin ezin izatea pena bat izan da, konparaketei beste ikuspuntu bat gehituko dietelako.

Dena den, orokorrean esan dezaket izandako arazo ia guztiak konpondu direla, eta hauei esker asko ikasten da geroago lan-munduko arazoei aurre egiten. Lan-munduko arazo askotan ere, arazoari buruzko informazioa bilatu behar da, eta askotan, lortutako informazioarekin ez da nahikoa eta guk sortu behar dugu konponbideren bat arazoei aurre egiteko, gauza horietan ikasitako graduak bentan irakasten du horri buruz; ez baita informatikari bat bakarrik izaten irakasten, baizik eta ingeniari bat izaten ere.

Egia esan, nik gogokoa nuen konputazio paralelo arloko gaiak; gaur egun, adimen artifiziala guztien ahotan dabilen kontzeptua da, eta jendeak ahazten du mota horietako aplikazioak exekutatu ahal izateko, atzetik azpiegitura bat garatu dela; honen adibide nagusiena *GPU*-ak dira, hauek oso erabiliak baitira adimen artifizialeko algoritmoak exekutatzeko. Horrez gain, aplikazio horiek exekutatzea ahalbidetzen duten tresnak eta plataformak garatu dira; hauek asko errazten dutelako mota ezberdineko programak ahalik eta azkarren exekutatzea. Beraz, nahiez eta jendea askotan ez konturatu, atzetik konputazio paraleloko kontzeptuak barneratzen dituzten tresnak eta gailuak erabiltzen dituzte, nahiz eta programatzailearekiko ikusgai ez den arren. Konputazio paraleloa eta konputagailuen arkitekturen eboluzioari esker, lehen ekin ezinak ziren arazoak orain posiblea da lortzea.

7.3 Etorkizunerako lana

Proiektua garatu ondoren, eta honi buruzko ondorioak azaldu ondoren, hainbat hobekuntza edo jarraipen geratu dira etorkizunean proiektuarekin jarraitu nahi bada, lortutako emaitzak hobetu nahi badira. Beraz, hau guztia esanda, hona hemen etorkizunean egin daitezkeen zenbait hobekuntza eta ataza, proiektua hobetzeko:

- **DIPC-ko ATLAS super-konputagailuan probak egin:** Aurreko ataletan azaldu den bezala, **Atlas** super-konputagailuan probak egiteak proiektua osoago egingo luke. Makina honekin, proiekturako erabili den makina baino askoz ahalmentsuagoa denez, proba ugari egin daitezke, hala nola, partikulen tamaina aldatzea edota iterazio kopurua handitzea. Modu horretan, simulatzailea parametro ezberdinekin exekuta daiteke eta horrekin, lortutako emaitzak aztertu daitezke, simulatzailea baldintza ezberdinetan duen portaera ikusiz.

- **CUDA-ko bertsio bat egitea:** Proiektuan garatutako bertsioetan lortutako emaitzak onak diren arren, faltan botatzen da **CUDA**-ko bertsio bat izatea ikusteko simulatzailea nola maneiatzen duen. Aurreko ataletan azaldu den bezala, *GPU*-aren erabilera proiektu honetarako izugarrizko ahalmena zuen egiten diren kalkulu motengatik. Dena den, aipatutako arazoengatik ezin izan da bertsioa garatzea, baino baliteke etorkizun batean denbora eta baliabide gehiagorekin simulatzailea birmoldatzea **CUDA** erabili ahal izateko.

Eranskinak

```
//
//  main.cpp
//  eris
//
//  Created by Nuno de Sousa on 27/12/13.
//  Copyright (c) 2013 Nuno de Sousa. All rights reserved.
//
//#include </usr/include/armadillo> //casa
//#include </usr/local/include/armadillo>
//
//

#define ARMA_64BIT_WORD
#include <iostream>
#include <complex>
#include <cmath>
#include <cstdlib>
#include <iomanip>
#include <armadillo>
#include <cstdio>
#include <sstream>
#include <fstream>
#include <string>
#include <ctime>
#include <omp.h>
#include <stdio.h>
#include "include/charger.h"
#include "include/particle.h"
#include "include/parameters.h"
#include "include/load_equations.h"
#include "include/power.h"
#include "include/sections.h"
#include "include/output.h"
#include "include/configurations.h"
#include "include/materials.h"
#include "include/fields.h"
#include "include/ab_coefficients.h"
#include "include/solver.h"
#include "include/io.h"
//#include "include/forces.h"
//#include "include/mie_polarizability/mie_polarizability.h"


using namespace std;
using namespace arma;
//using namespace sp_bessel; //necessary for complex_bessel

typedef complex<double> dcmplx;


int main(int argc, const char * argv[])
{
    cout << "Armadillo version: " << arma_version::as_string() << endl;

    //Clock operations
    clock_t start;
    double duration;
    start = clock();
    time_t prenow = time(0);

    // File to load the simulation parameter

    string filename = "simul_parameters.txt";
    io values;

    string filename_material = "material_list.txt";
```

```
double min = values.getLambdaMin(); //1000
double max = values.getLambdaMax(); //2000
int steps = values.getStep(); //200
double increment = (max-min)/double(steps);
double lambda;

int n_materials = n_lines_material(filename_material);

int threads = omp_get_max_threads();
cout << " threads ->" << threads << endl;


int div_int  = steps / threads;
double div_db  = steps / threads;

int rest = steps - (div_int * threads);

int rest_aux = rest;




#pragma omp parallel private (lambda)
{
  Particle *part;
  Parameters param;
  Material *mater;


  param.set_parameters_eps_0(values.getEps0());
  param.set_parameters_eps_m(values.getEpsM());
  param.set_E0(values.getE0());
  param.set_N_part(values.getNPart());


  int thread = omp_get_thread_num();


  double lambda_min;
  double lambda_max;
  int vecsize;

  //eskuz egindako iterazioen banaketa
  if (thread < rest){
    lambda_min = min +(increment*(div_int*thread+thread)); // 1000 + increment * (zatidura + rest eginda)
      //lambda_max = lambda_min +(increment*(div_int)); // 1000 + increment *(zatidura - 1 + rest_(1) )
      vecsize = div_int+1;
    }

  else {
    lambda_min = min +(increment*(div_int*thread+rest)); // 1000 + increment * (zatidura + rest eginda)
  //  lambda_max = lambda_min +(increment*(div_int-1)); // 1000 + increment *(zatidura - 1)
    vecsize = div_int;

  }


  double datavec[vecsize];
  datavec[0]= lambda_min;

  for( int i=1; i<vecsize; i++){
    datavec[i] = datavec[i-1]+increment;
  }
```

```
#pragma omp critical
{
  if (thread < steps){
    cout << "thread ->" << thread << " lambda_min ->" << lambda_min << "\n" << endl;

  }
 }

 mater = new Material[n_materials];

 for(int i = 0; i < n_materials; i ++)
 {
     mater[i].loader(i, filename_material);
 }







   for(int i= 0; i<vecsize; i++) //begizta nagusia
  {

    lambda = datavec[i];





       part = new Particle[param.N_part];

       Equations eq;
       vec absorption(3);
       absorption.zeros();

       cx_mat epstemp;
       vec Power1(3),Power2(3);
       epstemp.eye(3,3);
       param.set_parameters_wavenumber(sqrt(param.eps_m)*2*M_PI/lambda);


       cx_mat pol (3,3);
       dcmplx eps_medium(param.eps_m,0);

       //Load file
       load_structure(param.N_part, part, values.getNameLoadFile());

       time_t has = time(0);
       for(int i = 0; i < param.N_part; i = i + 1)
       {
           //cout << "loading particle n.º = " << i << endl;
           int material_control = 0;

           for(int j = 0; j < n_materials; j = j + 1)
           {
               if(mater[j].material_nome() == part[i].material)
               {
                   part[i].set_dielectric_tensor(mater[j].return_interp_value(lambda));
                   part[i].set_sphere_polarizability(pow((3./(4.*M_PI))*part[i].volume,(1./3.)), param);
                   cx_mat test = mater[j].return_interp_value(lambda);
                   material_control = 1;

               }

               if("Meyer_gold" == part[i].material)
               {
                   cx_mat id33;
```

```
            id33.eye(3,3);
            part[i].set_dielectric_tensor(epsilon_gold(lambda)*id33);
            part[i].set_sphere_polarizability(pow((3./(4.*M_PI))*part[i].volume,(1./3.)), param);
            //cx_mat test = mater[j].return_interp_value(lambda);

            material_control = 1;
        }

        if("resonant" == part[i].material)
        {
            part[i].set_resonant_polarizability(param);

            material_control = 1;
        }
    }

    if(material_control == 0)
    {
        cout << "MATERIAL DOESN'T EXIST IN THE LIST." << endl;
        exit(1);
    }

}




//transport
//-------------
//incident wave direction
if(values.getMod() == 0)
{

    param.set_plw_direction(values.getUx(), values.getUy(), values.getUz());

    if(values.getLoaderMethod() == "static")
    {
    eq.load_equations(param, part); //denbora hemen
    }
    eq.load_ind_term_plw(param, part);
}
//-------------

//emission
//-------------
if(values.getMod() ==1)
{
    param.set_source_pos(values.getXsource(),values.getYsource(),values.getZsource());
    eq.load_equations(param, part);
    eq.load_ind_term_source(param, part);
}
//-------------


//Solver
//-------------
//Direct Solver
if(values.getSolverMethod() == "Direct")
{direct_solver(&eq, &eq.A, &eq.B);}
//Biconjugate gradient stabilized method

if(values.getSolverMethod() == "BiCStab")if(values.getLoaderMethod()=="static")
    {{solver_BiCStab_static(&eq, &eq.A, &eq.B, param, values.getSolverNiterations(),
    values.getSolverError());}}
//Biconjugate gradient stabilized method with dynamic load
if(values.getSolverMethod() == "BiCStab")if(values.getLoaderMethod()=="dynamic")
{{solver_BiCStab_dynamic(&eq, &eq.A, &eq.B, part, param, values.getSolverNiterations(),
values.getSolverError());}}
```

```
    //-------------
    //Solver Error Message
    if(values.getSolverMethod() != "Direct" && values.getSolverMethod() != "BiCStab" &&
values.getLoaderMethod() != "static" && values.getLoaderMethod() != "dynamic")
    {
        cout << "Error: The method or loader doesn't exist.\nProgram stopped." << endl;

        exit(1);
    }
    //-------------

    if(values.getSwitchPolarizations() == 1)
    {
    string namepolx = values.getOutputString() + "_polarizations_xpol.dat";
    print_polarizations(eq, param, part, lambda, namepolx);
    string namepoly = values.getOutputString() + "_polarizations_ypol.dat";
    print_polarizations(eq, param, part, lambda, namepoly);
    string namepolz = values.getOutputString() + "_polarizations_zpol.dat";
    print_polarizations(eq, param, part, lambda, namepolz);
    }

    if(values.getSwitchFields() == 1)
    {
        string namefieldx = values.getOutputString() +"_fields_xpol.dat";
        print_fields(eq, param, part, lambda, namefieldx, "x");
        string namefieldy = values.getOutputString() +"_fields_ypol.dat";
        print_fields(eq, param, part, lambda, namefieldy, "y");
        string namefieldz = values.getOutputString() +"_fields_zpol.dat";
        print_fields(eq, param, part, lambda, namefieldz, "z");
    }

    long duration3 = 0;
    time_t buk = time(0);
    duration3 = ( has - buk);
    int ordu=duration3/3600.;
    int segundu=duration3 % 3600;
    int minutu=segundu/60.;
    segundu %= 60;
//  cout << "Denbora (s) = " << duration3 << "s. " << ordu << "h" << minutu << "min" << segundu <<
"sec" <<'\n';

    //transport
    //----------

    if(values.getMod() == 0 && values.getSwitchSections() == 1)
    {

        vec opt_theorem(3);
        vec sca_sec(3), abs_sec(3);
        vec ext_sec(3);

        abs_sec = abs_cross_section(&eq.X, param, part);
        ext_sec = ext_cross_section(&eq.X, param, part);
        sca_sec = sca_cross_section(&eq.X, &eq.im_parts, param, part);

        cout.precision(12);
        cout << "abs_cross_section = ";
        abs_sec.raw_print(cout);
        cout << "ext_cross_section = ";
        ext_sec.raw_print(cout);
        cout << "sca_cross_section = ";
        sca_sec.raw_print(cout);


        string sectionsx = values.getOutputString() + "_sections_xpol.dat";
        print_sections(lambda, abs_sec(0), ext_sec(0), sca_sec(0),sectionsx);
        string sectionsy = values.getOutputString() + "_sections_ypol.dat";
        print_sections(lambda, abs_sec(1), ext_sec(1), sca_sec(1),sectionsy);
```

```
        string sectionsz = values.getOutputString() + "_sections_zpol.dat";
        print_sections(lambda, abs_sec(2), ext_sec(2), sca_sec(2),sectionsz);

        for(int j = 0; j < 3; j ++)
        {
            opt_theorem = ((abs_sec + sca_sec)-(ext_sec))/((abs_sec + sca_sec)+(ext_sec));
        }
          cout << "thread   -> " << thread << "  lambda  " << lambda << endl;
          cout << "Optical theorem verification -> " << opt_theorem << endl;



    }



    //Projection over VSH basis
    if(values.getSwitchProjections() == 1)
    {
        dcmplx ae11, ao11, ae01, be11, bo11, be01;
        string name_aeo, name_beo;

        for(int pol = 0; pol < 3; pol = pol + 1)
        {
            ae11 = print_ae11(&eq.X, param, part, lambda, pol);
            ao11 = print_ao11(&eq.X, param, part, lambda, pol);
            ae01 = print_ae01(&eq.X, param, part, lambda, pol);
            be11 = print_be11(&eq.X, param, part, lambda, pol);
            bo11 = print_bo11(&eq.X, param, part, lambda, pol);
            be01 = print_be01(&eq.X, param, part, lambda, pol);

            if(pol == 0)
            {
                name_aeo = values.getOutputString() + "_projectionaeo_xpol.dat";
                name_beo = values.getOutputString() + "_projectionbeo_xpol.dat";
            }

            if(pol == 1)
            {
                name_aeo = values.getOutputString() + "_projectionaeo_ypol.dat";
                name_beo = values.getOutputString() + "_projectionbeo_ypol.dat";
            }

            if(pol == 2)
            {
                name_aeo = values.getOutputString() + "_projectionaeo_zpol.dat";
                name_beo = values.getOutputString() + "_projectionbeo_zpol.dat";
            }

            print_aeo(lambda, ae01, ae11, ao11, name_aeo);
            print_beo(lambda, be01, be11, bo11, name_beo);
        }
    }

    //Evaluation of the optical forces

    if(values.getSwitchForces() == 1)
    {
        mat force(3.*param.N_part,3);
        bool numerical_verification = false; //compare with the numerical calculation of the green tensor
        //force = forces_eval(&eq.X, param, part, lambda, numerical_verification);
        //string forcex = values.getOutputString() + "_force_xpol.dat";
        //print_forces(force, param, part, lambda,forcex, "x");
        //string forcey = values.getOutputString() + "_force_ypol.dat";
        //print_forces(force, param, part, lambda,forcey, "y");
        //string forcez = values.getOutputString() + "_force_zpol.dat";
        //print_forces(force, param, part, lambda,forcez, "z");
    }
```

```
        //emission
        //----------
        if(values.getMod() == 1)
        {

            Power1 = power_method_1(&eq.X, param, part);
            Power2 = power_method_2(&eq.X, &eq.im_parts, param, part, &absorption);

            cout << "Absoption = " << absorption << endl;
            cout << "power method one = " << Power1 << endl;
            cout << "power method two = " << Power2 << endl;
            cout << "Optical theorem verification = " << (Power1-Power2)/(Power1+Power2);


            //Pass the name to param_sim
            string name_1 = values.getOutputString() + "power1.dat";
            string name_2 = values.getOutputString() + "power2.dat";
            string name_3 = values.getOutputString() + "absorption.dat";
            print_Power(Power1, name_1);
            print_Power(Power2, name_2);
            print_Power(absorption, name_3);

        }



        //Evaluation of the field in a set of points defined in a file

        full_field_evaluation(values.getOutputString(), values.getInputStringPos(), eq, lambda, param, part);


        delete []part;






} //main forloop


    //Print running time
    //This is a closed function

    #pragma omp barrier

    if (thread ==0){
      duration = ( clock() - start ) / (double) CLOCKS_PER_SEC;
      long duration2 = int(duration);
      int hour=duration2/3600.;
      int second=duration2 % 3600;
      int minute=second/60.;
      second %= 60;
      cout << "Computation time (s) = " << duration << "s. " << hour << "h" << minute << "min" << second <<
"sec" <<'\n';
      time_t now = time(0);
      duration2 = ( now - prenow);
      hour=duration2/3600.;
      second=duration2 % 3600;
      minute=second/60.;
      second %= 60;
      cout << "Lapsed time (s) = " << duration2 << "s. " << hour << "h" << minute << "min" << second << "sec"
<<'\n';
      //End running time
```

```
        cout << "Program terminated." << endl;
    }



}

    return 0;
}
```

Bibliografia

[1] [2015] Norman Matloff. *Parallel Computing for Data Science: With Examples in R, C++ and CUDA*.

[2] [2011] Peter S. Pacheco. *An Introduction to Parallel Programming*.

[3] [2007] Barbara M. Chapman. *Using OpenMP*.

[4] [2019] Alexander A. Kokhanovsky. *Light Scattering Reviews: Single and Multiple Light Scattering* .

[5] [1989] John L. Hennessy, David A. Patterson. *Computer Architecture*.

[6] [1985] Bjarne Stroustrup. *C++ Programming Language*.

[7] [2020] Blaise Barney, Lawrence Livermore National Laboratory. Introduction to Parallel Computing. *https://computing.llnl.gov/tutorials/parallel_comp*.

[8] [2020] OpenMP. *https://www.openmp.org*.

[9] [2020] OpenMPI. *https://www.open-mpi.org*.

[10] [2020] CUDA Toolkit Documentation. *https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html*.

[11] [2020] GeeksforGeeks. Computer Architecture, Flynn's Taxonomy. *https://www.geeksforgeeks.org/computer-architecture-flynns-taxonomy*.

[12] [2019] CPU-Worlds. AMD Ryzen 3900x. *https://www.cpu-world.com/CPUs/Zen/AMD-Ryzen%209%203900X.html*.

[13] [2019] AnandTech. AMD Zen 2 Microarchitecture Analysis. *https://www.anandtech.com/show/14525/amd-zen-2-microarchitecture-analysis-ryzen-3000-and-epyc-rome*.

[14] [2020] Armadillo C++ library for linear algebra & scientific computing. *http://arma.sourceforge.net*.

[15] [2020] Intel. Intel Math Kernel Library. *https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html*.

[16] [2020] DIPC. ATLAS system. *http://dipc.ehu.es/cc/computing_resources/systems/atlas-fdr/*.