Paral·lelització de la simulació de scattering biestàtic

Mouad Naciri Chergui

Resumn– L'ús de senyals d'oportunitat és una manera avantatjosa de detectar la superfície de la Terra i l'oceà en una configuració de radar biestàtica passiva, ja que permet sensors més petits en comparació als radars actius. Mitjançant la reflectometria GNSS és possible obtenir estimacions de la rugositat de l'oceà, la criosfera, les aigües continentals o la humitat del sòl. Aquest treball té com a objectiu desenvolupar un simulador capaç de recrear les reflexions de senyals sobre la superfície del mar amb una configuració biestàtica del transmissor i receptor. Es realitza un estudi del rendiment i s'apliquen deferents tipus d'optimitzacions amb el propòsit de reduir el temps d'execució i satisfer els requisits de memòria que té el programa.

Paraules clau– Scattering biestàtic, Senyals d'oportunitat, Paral·lelisme, C, C++, OpenMp, MPI, HPC.

Abstract– The use of signal of opportunity (SoOp) is an advantageous way to sense the Earth and ocean surface in a passive bistatic radar configuration, as it allows for smaller sensors compared to active radars. Using GNSS reflectometry it is possible to obtain estimates of ocean roughness, cryosphere, inland waters or soil moisture. This work aims to develop a simulator capable of recreating signal reflections over the sea surface with a bistatic configuration of transmitter and receiver. A performance study is carried out and different types of optimizations are applied in order to reduce the execution time and satisfy the memory requirements of the program.

Keywords– Bistatic scattering, Signal of opportunity, Parallelism, C, C++, OpenMp, MPI, HPC.

1 INTRODUCCIÓ

El grup d'observació de la terra de l'Institut de Ciències de l'Espai està desenvolupant un nou mètode de mesura de corrents marins basat en un radar bi-estàtic, on el transmissor del senyal i el receptor es troben en posicions diferents. El receptor rebrà un senyal reflectit, acondicionat per l'estat de la superfície del mar, més el senyal transmès directament del transmissor al receptor. Amb el processat posterior d'una recepció continua dels senyals emesos pel transmissor amb els reflectits a la superfície del mar (autocorrelació), es pot predir mitjançant aquest nou mètode l'estat de la superfície del mar.

El que es vol aconseguir és que faci possible la comprovació d'aquest nou mètode i realitzar simulacions com a primer pas abans de dur a terme el prototipat *Hardware* i fabricar un dispositiu funcional que servirà per a implementar aquesta nova tècnica de predicció. El simulador permetrà crear diferents estats d'onades del mar, posicions del transmissor i receptor, i característiques del senyal transmès (potència, polarització, etc.) també modificables. Aquests elements variaran segons els paràmetres d'entrada. Es partirà de models matemàtics que permetran:

- Generar la superfície rugosa del mar a partir d'uns certs paràmetres d'entrada (temperatura, velocitat i direcció del vent, edat de les onades, mida de la superfície, etc.). [1]
- Calcular les reflexions dels senyals electromagnètics sobre la superfície a partir d'uns certs paràmetres d'entrada (coordenades del transmissor i receptors, temperatura i salinitat del mar, polarització i amplitud del senyal, etc.). [2] [3] [4]

El model matemàtic [1] permetrà generar la superfície ondulada del mar. La idea principal d'aquest model consisteix en aconseguir una superfície ondulada a partir d'un espectre freqüencial en 2 dimensions, i mitjançat les propietats de la transformada inversa de Fourier [5] es passarà del domini freqüencial al domini espacial. Aquest espectre consistirà en una matriu de dues dimensions, on el valor de cada posició indica una ponderació per a cada component freqüencial. Per a aconseguir-ho s'haurà d'especificar la velocitat i direcció del vent, l'edat de les onades, és a dir, quant de temps fa que existeix aquest vent que les ha generat, la mida de la superfície en unitats de distància i el nombre de punts a avaluar. Amb aquestes dades es generarà la densitat espectral de potència bidireccional que corresponen a les corbes de la figura 1. A partir de la corba corresponent, es redirigirà amb l'angle del vent indicat com a paràmetre

• E-mail de contacte: mouad.naciri@autonoma.cat
• Menció realitzada: Enginyeria de Computadors
• Tutor del treball: Juan Carlos Moure (DACSO)
• Director del treball: Serni Ribó (CSIC/IEEC)
• Curs 2021/22

d'entrada. Respecte als paràmetres d'entrada restants, l'edat de les onades influirà en l'alçada d'aquestes, la mida de la superfície en l'amplada de banda de freqüències que contribueixen a les onades i el nombre de punts de la superfície repercuteix en la resolució la superfície resultant. Finalment, en fer el canvi de domini freqüencial-espacial amb la transformada inversa de Fourier, l'ús d'una fase aleatòria permetrà obtenir un segment de superfície ondulat amb propietats molt similars a la superfície del mar. El resultat és similar al segment de figura 2.

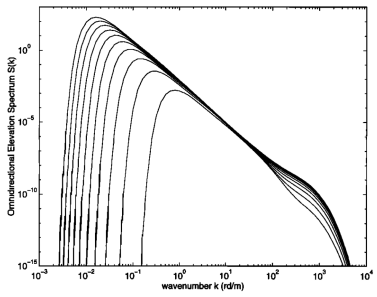

Fig. 1: Fully developed Donelan and Pierson. Aquestes corbes indiquen la densitat espectral de potència de cada component en funció de la longitud d'ona. Cada corba representa l'espectre resultant de diferents velocitats de vent.

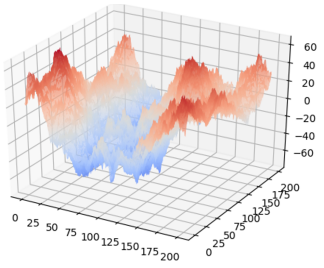

Fig. 2: Segment de la superfície rugosa que genera el model. Té una longitud de 200 metres en cada direcció.

El model de la reflexió del senyal es basa en les equacions de Maxwell, les equacions de Fresnel i la reflexió difusa i especular. Es parteix d'unes coordenades conegudes, tant del transmissor com del receptor, i les característiques de transmissió del senyal, amplitud, freqüència i polarització. Amb aquestes dades es calcularà l'integral del camp elèctric reflectit [1] a la superfície del mar, que arriba al receptor des de l'emissor, aplicant-hi els efectes de la dispersió esfèrica, les rotacions en la polarització mitjançant els coeficients de Fresnel i la direcció i ponderació de la reflexió deguda al pendent de cada punt de la superfície del mar, L'última part està representada a la figura 3.

$$P_{\hat{p},\hat{q}} = \frac{1}{4\pi}\int_S \frac{|E_0 e^{jk|\vec{x}_S-\vec{x}_T|}e^{jk|\vec{x}_R-\vec{x}_S|}|^2}{|\vec{x}_S-\vec{x}_T||\vec{x}_S-\vec{x}_R|}\cdot$$

$$\left[\hat{q}\cdot\left(\hat{p}\cdot\hat{h}_{inc}R_H\hat{h}_{ref}+\hat{p}\cdot\hat{v}_{inc}R_V\hat{v}_{ref}\right)\right]\cdot \qquad (1)$$

$$\left(\frac{\vec{k}_1-\vec{k}_2}{|\vec{k}_1-\vec{k}_2|}\right)\cdot\hat{n}\mathrm{d}S$$

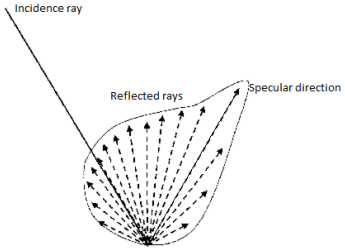

Fig. 3: Model de ponderació utilitzat. A la direcció de la reflexió especular es troba la màxima recepció del senyal, però a mesura que augmenta l'angle de separació respecte d'aquesta direcció, la magnitud disminueix. Aquest model té en compte l'efecte de la reflexió especular i la reflexió difusa.

2 OBJECTIUS

Principalment, s'ha de desenvolupar i validar un codi que implementa els models matemàtics mencionats anteriorment [1] i [2], [3], per tal de generar una superfície rugosa i ondulada que simula la superfície del mar i processar les reflexions a aquesta superfície.

El procés de validació es realitzarà amb una implementació, existent, equivalent en un codi Matlab, que realitzarà els càlculs de forma matemàticament equivalent amb l'objectiu de comparar el resultat de les dues versions.

Tant la funció que genera la superfície com la que processa la reflexió, prenen paràmetres d'entrada modificables, que fan variar les característiques de la superfície i del processament de la reflexió. Les característiques del mar i de l'ambient modifiquen la generació de la superfície. En canvi, la posició i orientació del transmissor i receptor modifiquen les característiques de la transmissió, reflexió i recepció. Per tant, la validació es farà amb una combinació de paràmetres d'entrada que permetin validar els càlculs tant de forma modular com completa.

El principal cas d'ús del programa serà simular les reflexions que mesuraria un receptor a una altura de $1500\,\mathrm{m}$ sobre el nivell del mar de senyals de GPS. El receptor té una directivitat a la recepció de 60°, que es tradueix a una àrea de $100\,\mathrm{km}^2$, $10\,\mathrm{km}$ a cada direcció, segons l'alçada indicada. La freqüència dels senyals de GPS va des dels 1227.60 MHz fins a 1575.42 MHz, que corresponen a $25\,\mathrm{cm}$ i $19\,\mathrm{cm}$ de longitud d'ona respectivament. La longitud d'ona està relacionada amb la resolució amb la qual s'ha de generar la superfície: entre 2 punts adjacents de la superfície hi ha d'haver una distància equivalent al 10% de la longitud d'ona del senyal. El nombre de punts totals en cada direcció es pot calcular com a:

$$Nombre\ de\ punts = \frac{longitud\ de\ la\ superfície}{distancia\ entre\ punts} \qquad (2)$$

La distancia entre punts, ha de ser del 10% de la longitud d'ona, que varia entre $19\,\mathrm{i}25\,\mathrm{cm}$ i la mida de la superfície és de $10\,\mathrm{km}$ en cada direcció. Això dona rang d'entre 400000 i 526316 punts per direcció. El nombre de punts resultant, Nx·Ny, és d'entre 160 i 277G punts. La mida en bytes pel cas on només es requereix un sol valor a memòria per cada punt es troba entre 640GB i 1108GB en el cas de precisió simple. Tenint en compte que els computadors no tenen aquesta quantitat de memòria, es procedirà a utilitzar diversos computadors, distribuint les dades entre les seves

memòries, per a realitzar el càlcul. Així és podrà satisfer els requeriments de l'execució pel que fa a l'ús de la memòria.

Un cop acabat el desenvolupament i la validació, es farà una anàlisi de rendiment amb el propòsit de trobar el *Bottleneck* del rendiment del programa i quin serà l'abast de l'execució amb els recursos disponibles. Principalment, l'anàlisi de rendiment consistirà a fer mesures dels temps d'execució, nombres d'instruccions executades, l'IPC, els encerts i les fallades a *cache*, temps d'execució de cada part del codi i l'evolució del codi màquina generat en aplicar canvis.

Posteriorment, es proposaran diverses optimitzacions per tal de reduir el temps d'execució, que inclouen l'ús de *Threads* i de computació distribuïda.

3 PLANIFICACIÓ

Per la part de desenvolupament inicial i validació se seguiran les estratègies de desenvolupament de software SCRUM. Consistirà a definir els progressos a realitzar per parts, fer una versió inicial i en els *Spints* completar les parts restants.

Per la part d'optimitzacions se seguirà una metodologia de desenvolupament circular. Consisteix a fer de forma iterativa certes etapes, millorant els resultats de forma incremental en cada iteració. Per exemple, un cop fet l'anàlisi de rendiment, es proposaran unes optimitzacions basades en aquests resultats de rendiment, s'aplicaran i s'analitzaran els resultats, depenent dels resultats s'aplicaran unes altres optimitzacions per tal d'aconseguir reduir el *bottleneck*, o bé es tornarà a fer una altra anàlisi de rendiment de forma circular.

Metodologia SCRUM:

- Definir les estructures de dades més adequades per a optimitzar l'execució i els recursos computacionals.
- Desenvolupar el codi de forma seqüencial generant una versió inicial del primer model [1]
- Completar la versió inicial de forma que sigui funcional[1]
- Desenvolupar el codi de forma seqüencial generant una versió inicial del segon model [2][3]
- Completar la versió inicial de forma que sigui funcional[2][3]
- Verificar els resultats que generen les funcions, valors numèrics resultants dels càlculs.

Metodologia circular:

- Anàlisi de rendiment i perfilat
- Proposta d'optimitzacions i reestructuració del codi
- Implementació incremental de les optimitzacions amb diferents estratègies.
- Simulació i comparació amb altres versions
- Integració de les optimitzacions.
- Proposta de codi amb memòria compartida amb l'ús de MPI [6].

4 METODOLOGIA

S'iniciarà amb el desenvolupament del codi d'un dels models. Es genera un codi base que tindrà com a objectiu ser modular durant la fase de desenvolupament i validació. Per a realitzar verificacions unitàries i correccions de forma simple. Es definiran unes estructures de dades coherents amb l'objectiu anterior. Aquestes podran ser canviades a l'hora d'aplicar optimitzacions. Un cop acabat el desenvolupament dels dos models i realitzades les verificacions adients, es començarà amb el procés d'optimitzar.

Un cop es fa l'anàlisi i perfilat del codi, es proposaran les optimitzacions per a tractar el textitBottleneck trobat. Aquestes optimitzacions seran incrementals i s'aplicaran sense afectar el resultat del programa.

El textitBottleneck principal del programa es troba a un bucle que és tipus MAP, per tant, es tractarà de millorar la versió *single thread* al màxim possible, sense limitar l'ús d'altres optimitzacions posteriorment. Aquest tipus de patró és senzillament paral·lelitzable amb *threads*, i moltes de les optimitzacions de la versió *single thread* hi són també compatibles amb la versió *multi-thread*.

En general, la trajectòria a seguir serà llavors optimitzar la part de codi que més temps d'execució consumeix a la versió *single thread*, tenint present que ha de ser compatible amb paral·lelisme a nivell de *thread*, i també en un entorn de memòria distribuïda.

També, de forma paral·lela i a mesura que es progressa al projecte, es prendrà nota de possibles optimitzacions aplicables al codi per a posteriorment considerar-les i implementar-les. D'aquesta manera es segueix un ordre establert i no es corre el risc de barrejar diferents elements i afectar a la consistència dels resultats i conclusions.

Per a realitzar un seguiment de les optimitzacions de forma eficient s'utilitzarà Git. Després d'aplicar modificacions al codi, es procedeix a executar la nova versió i a avaluar el resultat del programa, per a validar la integritat del codi, i per avaluar el rendiment obtingut. S'anirà pujant les versions de forma incremental, amb una breu descripció de cada versió, en una mateixa branca o en diferents branques, depenen del caire de les modificacions que es vulguin aplicar. D'aquesta manera podrem accedir posteriorment a cada versió per separat i comparar ràpidament les modificacions amb altres versions.

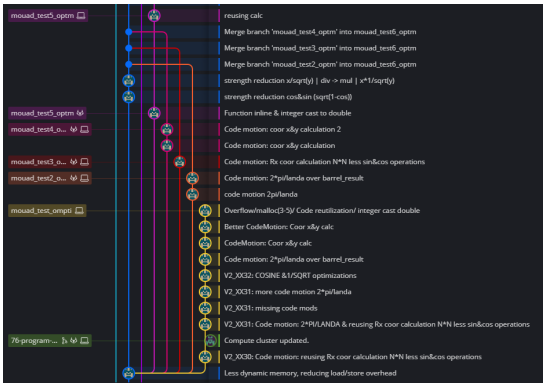

Fig. 4: En aquest esquema es mostren les diferents branques de progrés, que estan dividides en diferents línies d'optimitzacions. Es pot accedir al codi font de cada versió i consultar les diferències des de la mateixa interfície. Està organitzat, de manera que la línia temporal avança cap a dalt de forma paral·lela per a totes les branques.

De forma addicional, aquesta metodologia ens permetrà retrocedir a versions anteriors sincronitzant la versió desitjada per a poder avançar des d'un punt anterior en cas que un camí es quedi estancat, o bé dur a terme millores incrementals de forma paral·lela a diferents branques, per a poder avaluar cadascuna per separat, i després agrupar-les en una resultant.

A part del codi, es pot guardar la informació resultant del perfilat i l'arxiu executable. Cal afegir que totes aquestes versions no són visibles al nostre directori de forma simultània, això ens permet tenir en ordre una gran quantitat de diferents versions i també un control gràfic mitjançat una línia temporal per tenir una traçabilitat visual, que no seria possible si es fes aquesta gestió de forma local.

A la figura 4 es pot veure com des d'una versió base, es van crear noves branques, a les quals es va aplicar una optimització d'un tipus diferent. Finalment, es van ajuntar a una versió resultant de forma incremental. Controlant que les optimitzacions són compatibles entre si, fins a arribar a una versió final. A partir d'aquesta versió s'aplica només un tipus d'optimització, per tant, només es fa progressa una sola branca, fins a una nova etapa on es vulgui realitzar diferents optimitzacions incrementals per separat a una versió base donada.

5 Estructura del programa

El programa bàsicament consta de dues funcions principals, *generate_surface* i *compute_surface_reflection*. La primera genera la superfície del mar, un vector de Nx*Ny posicions. La segona processa les reflexions d'un número de punts Nx*Ny a aquesta superfície amb el càlcul de la integral 1. El resultat és un conjunt de valors que corresponen a les diferents posicions del receptor.

Tant la complexitat temporal com espacial del programa és O(Nx*Ny). El temps de còmput i la memòria necessària tenen un creixement lineal, proporcional a la quantitat de punts avaluar.

5.1 Estructures de dades

L'estructura principal del programa és una estructura de vectors, *Structure of arrays (SoA)*. On es desen les 3 coordenades de cada punt de la superfície generada i els valors intermedis de cada punt necessaris per calcular l'integral. A l'apèndix A.1 es pot veure en detall l'estructura inicial. Per altra banda, els resultats del programa es desen a una matriu de nombres reals.

5.2 Generar la superfície

La funció *generate_surface* consta de 4 etapes per a generar la superfície. La primera consisteix en un bucle de tipus MAP que genera Nx*Ny*2 valors que corresponen a la part real i part imaginària densitat espectral de potència. A l'apèndix A.2.1 es pot consultar el codi corresponent. A la segona etapa s'aplica una fase aleatòria hermítica, simètrica entre la part superior i inferior de la matriu, (A.2.2). L'etapa següent consisteix fer una transposició de la matriu en 4 blocs, (A.2.3). Finalment, la quarta etapa realitza la transformada inversa de Fourier bidimensional, (A.2.4). A les figures següents es poden observar de forma gràfica les etapes s'han descrit.

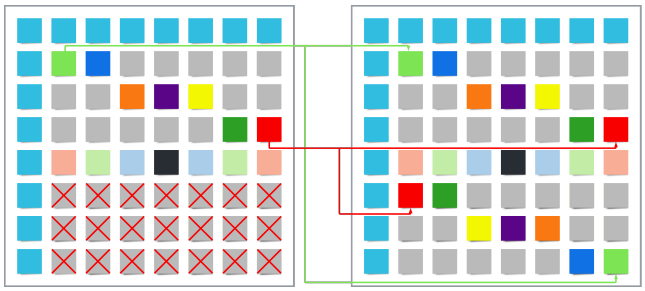

Fig. 5: Aquesta figura representa la primera i segona etapa per generar la superfície: Generar els valors de la matriu amb una fase aleatòria hermítica. Cada punt de la matriu destí (part dreta) és generat a partir d'un únic punt de la matriu original (part esquerre). Existeix una simetria que és representada amb la correspondència en colors iguals. Les condicions de contorn, els punts representats amb el mateix color blau, tenen una correspondència punt a punt sense cap simetria. El punt del mig té valor 0.

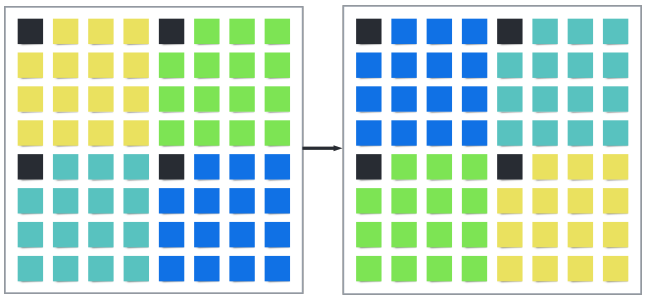

Fig. 6: A la part esquerra es mostra la matriu original, a la dreta es mostra la transposició de la matriu en 4 blocs. Que correspon a la tercera etapa per a generar la superfície.

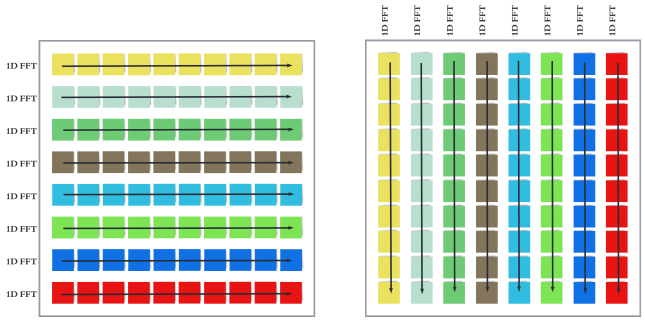

Fig. 7: Representació del patró d'accés que realitza la transformada de Fourier bidimensional. Primerament, té un patró stride-1, posteriorment stride-n.

5.3 Processar la superfície

Després de generar la superfície, el programa té un bucle principal on es crida a la funció *compute_surface_reflection*. Aquest bucle fa un escombrat de les diferents posicions de receptor que s'han definit. Aquesta funció és tipus MAP-Reduce, calcula l'aportació de cada punt de la superfície per separat i ho acumula a una variable amb l'aportació de la resta de punts, (A.3.1). Per cada iteració del bucle principal, es calculen les reflexions que arribarien a cadascuna de les coordenades del receptor, sobre les quals s'itera, i es guarda el resultat de la reducció de cada iteració en una posició d'un vector (*barrel_result*). La figura 8 representa gràficament el càlcul que es realitza al bucle principal per a cada posició del receptor i la figura 9 esquematitza les variables i operacions que es fan a cada punt de la superfície.

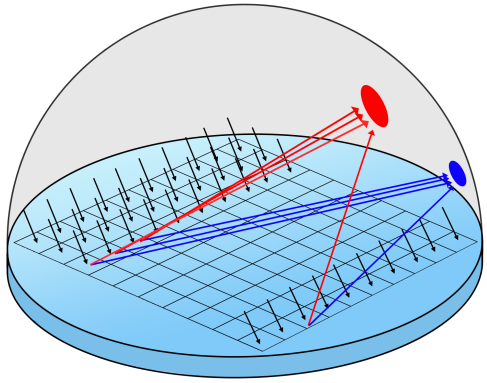

Fig. 8: Cada quadrat de la malla representa un punt de la superfície. Es calculen les reflexions punt a punt, transmissor-superfície-receptor. El receptor, representat pel punt blau i vermell, realitza una trajectòria esfèrica. Per a cada coordenada del receptor es processa tota la superfície per a calcular les reflexions en aquella direcció.

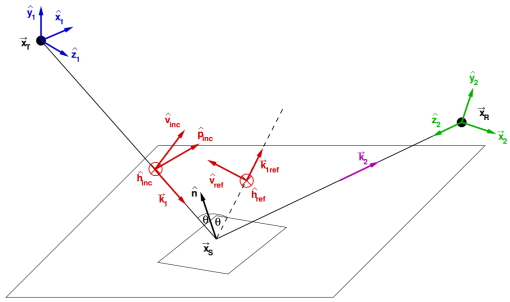

Fig. 9: Es defineix X_T com a transmissor, X_S punt de la superfície i X_R com a receptor. Per a cada punt de la superfície es calcula el vector incident, normal i reflectit, l'atenuació deguda a la distància i s'hi apliquen les rotacions als vectors de polarització.

6 ENTORN I CONFIGURACIÓ D'EXECUCIÓ

S'utilitzarà el compilador *gcc* per les primeres versions del codi. Per a paral·lelitzar amb *threads* i distribuir el còmput es faran servir les APIs d'*OpenMP* i *MPI*.

L'arquitectura on s'executarà el programa és un clúster amb 4 nodes amb una cua *SLURM*. Cada node disposa d'una configuració de 2 *sockets* × 16 cores (Intel Xeon Gold 5218) i 384 GB de RAM.

7 OPTIMITZACIONS

A continuació es descriuran les diferents optimitzacions que han estat aplicades, categoritzades segons on es troba el seu impacte en el rendiment. Es faran servir termes específics per referir el tipus d'optimitzacions. Part d'aquesta terminologia quedarà definida a continuació. La resta s'anirà introduint de forma incremental en ser utilitzada.

Strength reduction: Es coneixen com a *Strength reduction* aquelles optimitzacions on operacions computacionalment costoses són reemplaçades per unes equivalents de menys costoses.

Code motion: També coneguda com a *loop-invariant code*, consisteix a evitar el càlcul repetitiu de resultats constants, movent aquest a fora de regions iteratives, com ara bucles, i utilitzant aquest valor com a variable constant.

Loop fusion: Tracta de fusionar diferents bucles en un sol. Redueix el *overhead* de tenir més d'una estructura de control i, depenen del cas, pot ajudar a aprofitar la localitat de les dades, per haver d'accedir menys vegades a memòries lentes.

Loop interchange: Intercanviar l'ordre dels bucles niats. Pot aconseguir realitzar un patró d'accés diferent sobre les dades, aprofitant la disponibilitat d'aquestes a la *cache*.

Loop collapse: Convertir bucles niats en un únic bucle. Ajuda a reduir l'*overhead* dels bucles i beneficia en la predicció de salts. També pot millor l'eficiència de la vectorització. Es tractarà un cas pràctic amb detall més endavant.

Vectorization: També conegut com a paral·lelisme *SIMD*, aprofita l'amplada dels ports d'execució del processador, per a computar en una única operació diverses dades alhora. La millora que s'obté depèn del H/W en què s'executa i la mida en bits de les dades amb les quals s'opera.

Multi-threading: També conegut com a paral·lelisme *MIMD*, tracta d'utilitzar diferents *threads* per a augmentar el throughput de l'execució. Els *threads* poden ser afins, ser d'un mateix nucli d'execució, o no, pertànyer a diferents *cores*. El paral·lelisme s'obté quan es fan servir diferents *cores*. Si es fan servir *threads* d'un mateix *core*, s'obté un efecte de paral·lelisme més ben conegut com a concurrència.

7.1 Versió inicial

El resultat de perfilar la primera versió del codi indica que el 99% del temps d'execució es dedica a la funció: *compute_surface_reflection*. El 40% del temps dedicat a aquesta funció correspon a crides a una llibreria per fer operacions de sinus i cosinus. Es conclou, llavors, que el programa actual està més limitat per latències/capacitat de còmput que pels accessos a memòria.

S'aplicaran en un inici optimitzacions relacionades amb el còmput, amb l'objectiu de reduir el temps d'execució sense afectar els resultats del programa original.

7.2 Versió 2: optimitzacions de còmput

7.2.1 Strength reduction

Per a normalitzar els vectors de 3 coordenades, es dividia cada component entre el mòdul del vector. S'ha canviat el còmput de 3 divisions de tipus *floating-point* per multiplicacions amb el valor invers. Aquest càlcul, a més, s'aprofita en un bucle més intern on es realitzaven dues divisions, a cada iteració, amb el mateix valor.

La següent optimització està enfocada al càlcul de ***Rp***. Calcular ***Rp*** (3) comporta realitzar el cosinus d'un nombre complex, ja que el quocient d'***n1/n2*** resulta en un nombre complex. Aquest cosinus imaginari s'acaba traduint en 4 operacions trigonomètriques reals, 2 multiplicacions i una resta (4).

```
// INITIAL
    double mod =  expression(vX[i],vY[i],vZ[i]);

    vX[i] /= mod;
    vY[i] /= mod;
    vZ[i] /= mod;

    ...

    for (int j = 0; j < M; j++)
    {
        ...

        resA = cos(fase)/mod;
        resB = sin(fase)/mod;

        ...
```

```
// STRENGTH REDUCTION OPTM.
    double mod =  expression(vX[i],vY[i],vZ[i]);

    double INV_mod = 1.0 / mod;

    vX[i] *= INV_mod;
    vY[i] *= INV_mod;
    vZ[i] *= INV_mod;

    ...

    for (int j = 0; j < M; j++)
    {
        ...

        resA = cos(fase)*INV_mod;
        resB = sin(fase)*INV_mod;

        ...
```

Fig. 10: Pseudocodi de l'optimització que redueix el nombre de divisions, detallada a l'apartat 7.2.1

Utilitzant el teorema fonamental de la trigonometria (5) s'aconsegueix obtenir una expressió matemàticament equivalent i computacionalment menys costosa (6). El resultat és que ja no es computen aquestes operacions trigonomètriques a canvi de realitzar una arrel quadrada addicional.

$$R_{\mathrm{p}} = \left| \frac{n_1 \cos(arcsin(\frac{n1}{n2} sin\theta_{\mathrm{i}})) - n_2 \cos\theta_{\mathrm{i}}}{n_1 \cos(arcsin(\frac{n1}{n2} sin\theta_{\mathrm{i}})) + n_2 \cos\theta_{\mathrm{i}}} \right|^2 \tag{3}$$

$$cos(a+bi) = cos(a)cosh(b) - isin(a)sinh(b) \tag{4}$$

$$\sin^2\alpha + \cos^2\alpha = 1 \tag{5}$$

$$R_{\mathrm{p}} = \left| \frac{n_1 \sqrt{1 - (\frac{n_1}{n_2}\sin\theta_{\mathrm{i}})^2} - n_2 \cos\theta_{\mathrm{i}}}{n_1 \sqrt{1 - (\frac{n_1}{n_2}\sin\theta_{\mathrm{i}})^2} + n_2 \cos\theta_{\mathrm{i}}} \right|^2 = \left| \frac{n_1 \sqrt{1 - (\frac{n_1}{n_2})^2(1 - (cos\theta_{\mathrm{i}})^2)} - n_2 \cos\theta_{\mathrm{i}}}{n_1 \sqrt{1 - (\frac{n_1}{n_2})^2(1 - (cos\theta_{\mathrm{i}})^2)} + n_2 \cos\theta_{\mathrm{i}}} \right|^2 \tag{6}$$

Com mostra la figura [11], es realitzava un càlcul de l'arccosinus per a posteriorment calcular-ne el cosinus, com són operacions inverses, el càlcul del cosinus es pot evitar, ja que es parteix d'aquest valor. Pel que fa al càlcul del sinus, aquest, es pot obtenir computacionalment més ràpid utilitzant el teorema mencionat amb anterioritat (5). Aquest canvi també queda reflectit a la figura [11] i a la part dreta de l'equació (6).

```
// INITIAL
for (int i = 0; i < N; i++)
{
    ...

    val = expression();

    angle[i] = acos(val);

    ...
}
for (int i = 0; i < N; i++)
{
    cos_angle = cos(angle[i]);
    sin_angle = sin(angle[i]);

    Rp = expression(cos_angle, sin_angle);
    ...
}
```

```
// STRENGTH REDUCTION OPTM.
for (int i = 0; i < N; i++)
{
    ...

    val = expression();

    cos_angle[i] = val;

    ...
}
for (int i = 0; i < N; i++)
{
    cos_angle = cos_angle[i];
    sin_angle = sqrt(1.0-cos_angle*cos_angle);

    Rp = expression(cos_angle, sin_angle);
    ...
}
```

Fig. 11: Pseudocodi de l'optimització detallada al final de l'apartat 7.2.1

7.2.2 Code motion

Per tal de no repetir càlculs amb molta latència a l'execució, s'ha creat una *lookup-table* per a desar valors calculats amb anterioritat. Utilitzant una indexació especifica, es llegiran de memòria els valors precalculats corresponents a cada iteració. Aquesta estratègia té sentit, ja que la quantitat de memòria que ocupen els valors desats a aquesta taula és molt inferior a la mida de la *cache* L1, i la latència d'accés a aquest nivell és inferior a la dels càlculs per generar els valors.

La mida d'aquesta taula és de 36*9*8 Bytes = 2.6 kB. La mida més habitual a les arquitectures actuals de la *cache* L1 és de 32 kB. Això permet que totes les dades hi càpiguen a aquest nivell.

Aquesta optimització té més rellevància quan s'aplica conjuntament amb el *Loop interchange* que es descriu en un apartat posterior.

Per a acabar, s'ha tret com a factor comú un càlcul que genera un valor constant a totes les iteracions del bucle més intern: 2*pi/Lambda. A més, com es tracta d'una multiplicació que s'aplicava a tots els valors abans de sumar-los, es pot treure i aplicar-la en acabar l'operació de reducció de tota la integral.

7.2.3 Vectorization

En perfilar el codi, s'ha observat que el compilador no genera codi vectoritzat. Per a què el compilador generi un codi amb instruccions *SIMD*, a part d'utilitzar el *flag: ftree-vectorize* que activa l'ús d'aquestes instruccions, s'han de realitzar modificacions al codi actual. Una de les raons que no fa possible la vectorització és la instrucció *UCOMISS*. Aquesta instrucció és escalar i no té una variant *SIMD*. Aquesta instrucció s'encarrega de realitzar una comparació i comprova que cap dels valors sigui QNaN o SNaN. El seu ús es pot evitar compilant amb l'opció *-fno-math-errno*.

La següent causa que no permet la vectorització és el possible *aliasing* entre els vectors. Es pot indiciar al compilador que un vector no té problemes d'*aliasing* amb un altre fent ús de la paraula clau *__restrict* davant la declaració dels vectors.

L'ús d'una indexació complicada dificulta la vectorització al compilador, per tant, tots els accessos a memòria a posicions constants al bucle més intern s'hi han substituït per a variables locals. Un altre impediment a la vectorització és l'ús de la llibreria *std::complex*. Aquesta llibreria sobrecarrega l'operador '*' de manera que en multiplicar dos nombres complexos, genera 4 multiplicacions i 2 sumes. Com que el compilador substitueix l'operador per a una trucada a una funció de la llibreria, el resultat és que no vectoritza el bucle. S'eviten llavors l'ús de les sobrecàrregues i es desenvolupen les operacions. El resultat és un codi més simple i vectoritzable pel compilador.

El càlcul del cosinus al bucle més intern impedeix la vectorització pel fet que no existeix una instrucció màquina que calculi el cosinus i menys una variant vectoritzada d'aquesta. S'ha utilitzat una implementació d'una aproximació amb sèries de Taylor de les funcions cosinus i sinus [7], i s'ha adaptat el codi per a què es pugui vectoritzar.

Finalment, per aprofitar més la vectorització, es fa un *loop collapse* dels dos bucles més interns del programa, això permet que sigui més fàcil aconseguir que el nombre total d'iteracions del bucle vectoritzat sigui múltiple del factor de vectorització. En el cas de l'execució que es vol realitzar, els bucles fan 36x9 iteracions, com que 9, el nombre d'iteracions del bucle més intern, no és múltiple de cap fac-

tor de vectorització que ofereix l'arquitectura (2, 4, 8...), hi haurà una iteració que es calcularà de forma escalar durant les 36 iteracions del bucle més proper a aquest. En fer la unió dels dos bucles, el resultat són 324 iteracions, com és múltiple de 4, permetrà vectoritzar totes les iteracions amb un mínim factor de vectorització de 4, a diferència d'abans que hi haurien 36 iteracions que es faria de forma escalar.

7.3 Versió 3: optimitzacions de memòria

En aquest punt, es considera que el principal *bottleneck* són els accessos a memòria degut a la quantitat de fallades a textitcache que hi ha. Les diverses funcions que es criden a cada iteració del bucle més extern accedeixen a vectors que són més grans que latextitcache i en generen uns altres temporals. A la següent versió es proposaran optimitzacions per a tal d'aprofitar la localitat de les dades i, en la mesura que es pugui, reduir aquests accessos.

7.3.1 Loop fusion

La funció *computi_surface_reflection* fa crides a altres funcions, on aquestes generen vectors intermedis per a passar a les següents i continuar amb el càlcul. Totes aquestes funcions fan el mateix nombre d'iteracions, i el seu patró d'accés i còmput és tipus MAP. S'ha fusionat tots els bucles en un sol: d'aquesta manera s'aprofita la localitat temporal de les dades i es redueix el nombre d'accessos a memòria. Un cop aplicat aquest canvi, es fa innecessari desar els resultats parcials als vectors temporals. Es canvia l'estructura de dades inicial eliminant els vectors, a canvi, es fan servir variables locals a cada iteració. A l'apèndix A.4 es pot trobar la nova estructura de dades.

7.3.2 Loop interchange

En canviar l'ordre en què s'accedeix a les dades s'aconsegueix més localitat temporal. D'aquesta forma, les dades de la superfície es llegeixen el mínim nombre de cops possible de memòria principal i es realitza tot el còmput necessari quan les dades es troben als nivells de *cache* més propers al processador. Es redueix encara mes la quantitat de lectures a memòria i s'aprofita més la localitat temporal de les dades.

D'altra banda, es redueix la quantitat de dades que es generen fent ús de les mateixes variables d'iteració del bucle per a calcular les coordenades de la superfície en 2 dels 3 eixos.

També, tenint les coordenades del transmissor i la superfície fixes, es calculen les reflexions per a cada posició del receptor s'ha d'avaluar. El resultat és que la superfície es llegirà un sol cop. Una part dels càlculs es mouran a un bucle més, per tant, es reutilitzaran les dades al bucle més intern perquè seran constants per a totes les iteracions d'aquest.

Els resultats de les optimitzacions fins aquest punt es poden consultar a la TAULA 1. Han disminuït la quantitat d'instruccions que s'executen gràcies de les optimitzacions i simplificacions en còmput i a la reducció de lectures i escriptures. També el nombre de fallades a *cache*, ja que ara s'aprofita la localitat de les dades.

Versió	Original	V3	V3 SIMD
Instructions	1815G	87G	17.9G
L1-dcache-loads	270G	23G	7.5G
L1-dcache-load-misses	2G	38M	73M
L1-dcache-stores	109G	6G	2.3G
IPC	1.27	1,13	0.76
Time elapsed (s)	427,89	23,12	7.04

TAULA 1: AQUESTA TAULA MOSTRA LES DADES DE RENDIMENT MESURADES D'UNA EXECUCIÓ DE NX*NY=1024*1024 A UN CORE NEHALEM

7.4 Versió 4: Multi-threading

Un cop s'ha assolit el màxim potencial d'un *core*, podem aprofitar la resta del processador utilitzant *threads* amb les directives d'*OpenMp* [8]. L'ús de *threads* comporta un *overhead* en la creació d'aquests i assignar-los còmput a realitzar. Per tant, per a minimitzar aquest *overhead*, s'ha de declarar la regió paral·lela i assignar treball als *threads* el mínim número de vegades possible. Això es pot aconseguir paral·lelitzant el bucle més extern. Per a poder paral·lelitzar el bucle més extern i evitar condicions de carrera, es crearà un *buffer* privat per a cada *thread* on desarà els resultats dels càlculs de forma privada. Finalment, en acabar tot el bucle i mitjançant una operació de reducció, se sumaran les dades resultants en un únic espai de memòria. Es podria evitar l'ús d'aquest *buffer* privat, fent ús d'un compartit i estructures de control com *pragma omp atomic* o *pragma omp critical*, però el que s'aconseguiria és seqüencialitat a l'execució d'aquella part del codi. A la figura 12 es troba el fragment paral·lelitzat amb *threads*.

```
#pragma omp parallel  num_threads(4)
{
    int idthread = omp_get_thread_num();
    ...
    ...
    #pragma omp for
    for(int i=0; i < SIZE_XX; i++){
        ...
        ...
        ...

        for(int j=0; j < SIZE_YY; j++){
```

Fig. 12: Pseudocodi de la regió paral·lela de bucle principal.

Versió	V3 SIMD	V4 4-Threads
Instructions	17.9G	18G
L1-dcache-loads	7.5G	7.5G
L1-dcache-load-misses	73M	138M
L1-dcache-stores	2.3G	2.4G
IPC	0.76	0.76
Time elapsed (s)	7.04	1.94

TAULA 2: AQUESTA TAULA ES MOSTREN LES DADES DE RENDIMENT MESURADES D'UNA EXECUCIÓ SINGLE THREAD I AMB 4 THREADS

El fet d'utilitzar *threads* no garanteix la reducció del temps d'execució de forma proporcional a nombre d'aquests. Hi ha factors que no permet arribar a aquest límit,

com pot ser l'amplada de banda a memòries compartides. La TAULA 3 mostra com el temps d'execució no es redueix de forma proporcional al nombre de *threads* utilitzats. S'ha paral·lelitzat el bucle més extern,s que consumeix el 99% del temps d'execució, i aquest no té cap classe de dependències recurrents. Per tant, el factor que limita que el *throughput* de l'execució escali amb un nombre *threads* és l'amplada de banda a algun nivell de memòria compartit entre els *threads*, L3 o RAM.

Versió 4 (nº Threads)	2	4	8
Time elapsed (s)	55.8	27.6	19.4

TAULA 3: AQUESTA TAULA CONTÉ ELS TEMPS D'EXECUCIÓ DE NX*NY= 4096*4096 AMB DIFERENTS NOMBRES DE THREADS

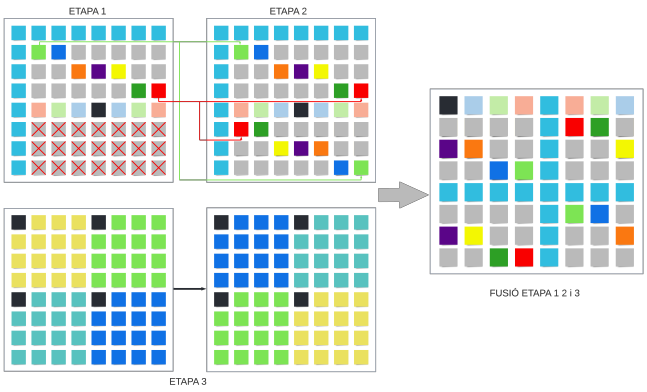

Fig. 13: Aquesta figura mostra el canvi realitzat al procés d'unificar les tres etapes de la funció que genera la superfície.

7.5 Versió 5: MPI

L'ús de *MPI* ens permetrà utilitzar els recursos de diversos computadors per a distribuir el còmput i satisfer els requeriments de memòria del programa. Per aconseguir-ho s'hauran de fer modificacions al codi i establir comunicacions entre els diferents nodes. Aquest procés s'ha de completar sense afectar els resultats del programa i tenint en compte l'*overhead* que generen les transferències de dades per xarxa.

La idea és que cada node disposi d'un segment de la superfície i que es facin els càlculs de forma local. Quan es requereixi fer sincronitzacions. I en finalitzar el còmput enviar els resultats parcials generats per cada node.

7.5.1 Generar la superfície amb memòria distribuïda

A la segona etapa, figura 5, s'ha de conservar la simetria de la fase aleatòria de forma distribuïda. S'implementa una funció que genera un valor aleatori, però igual per a dos punts simètrics. Això s'aconsegueix creant el valor a partir de la relació de coordenades dels punts simètrics, apèndix A.5.3 . A més s'afegeix una aportació a la fase. Aquest valor afegit és aleatori entre diferents execucions del programa i constant per a tots els punts per una mateixa execució. D'aquesta manera, s'obté una fase aleatòria amb les propietats de simetria requerides i un factor d'aleatorietat entre diferents execucions.

Seguidament, s'unifiquen les 3 primeres etapes que creen les dades de la superfície en una sola etapa. El que s'aconsegueix és que les dades es generin i es desin directament a la seva posició final, evitant fer moviments de les dades com mostra la figura 6. Aquest canvi ens estalvia realitzar transferències de dades entre nodes per aquestes 3 etapes. La figura 13 mostra el resultat de la fusió de les etapes. A l'apèndix A.5.1 i A.5.2 es pot trobar el codi de la fusió de les tres etapes.

En arribar a la 4a etapa, cada node disposarà d'una fracció del total de les dades. Consistirà en un conjunt de files completes com es mostra a la primera part de la figura 31. Amb aquesta disposició de les dades es podran calcular les IFFTs en la 1a dimensió de forma paral·lela, a cada node, sense realitzar cap transferència o sincronització. Per les IFFTs verticals es reordenen les dades de la matriu per a millorar el rendiment del patró d'accés del càlcul posterior i es comparteixen les dades necessàries entre els nodes. Amb la funció *MPI_Alltoall* podrem transferir les dades de forma que cada node enviarà i rebrà només els valors necessaris. A l'apèndix A.5.4 es pot trobar el codi de reordenació i transferència de les dades.

Per acabar, cada node ha de compartir la fila superior i inferior amb el node anterior i posterior respectivament del seu segment de superfície. Pel càlcul del vector normal de cada punt es requereixen els valors de contorn, cosa que implica disposar de la fila superior i inferior als segments de la superfície. L'apèndix A.5.5 conté el codi d'aquestes transferències.

7.5.2 Processar la superfície amb memòria distribuïda

Per la part del càlcul de les reflexions, en ser tipus MAP, els canvis són menors. Com que cada node té un segment de la superfície, s'ha de mantenir la concordança de les coordenades dels punts. S'ha d'afegir un *offset* a cada node per a recuperar el valor original de les coordenades. Un cop acabat el càlcul, amb *MPI_Reduce* s'aplica una operació de reducció per a sumar els resultats parcials de cada node, a l'apèndix A.6.1 es pot troba el codi esmentat.

Versió MPI	1024*1024	4096*4096	8192*8192
Time elapsed (s)	2.8	36.1	144.2

TAULA 4: AQUESTA TAULA CONTÉ ELS TEMPS D'EXECUCIÓ AMB 4 NODES I 4 THREADS

Com mostra la taula anterior, l'ús de la computació distribuïda comporta un *overhead* inicial per la creació dels processos i les transferències de les dades respecta la versió amb un computador per una mateixa dimensió Nx*Ny, TAULA 3. Però a partir d'un cert volum de treball a realitzar aquest *overhead* no té un gran pes. Tot i això, ens permet poder arribar a avaluar casos que no seria possible a causa de la limitació de recursos disponibles en sol computador.

8 RESULTATS GRÀFICS

Visualitzar el resultat de forma gràfica pot ajudar a fer una validació inicial més ràpidament. En el cas del programa desenvolupat, el resultat és una matriu de valors, on cada valor representa la magnitud calculada per una posició dl receptor. Aquest conjunt de posicions té una forma semiesfèrica.

Per mostrar aquests resultats de forma gràfica, en forma de semiesfera, s'ha implementat una funció utilitzant OpenGL [9]. Es genera una forma semiesfèrica a partir de la unió de punts i línies, formant triangles, apèndix A.7.1, i es representaran els valors a mostrar segons el color de cada posició.

Es defineix una funció que crea un triangle a partir de tres punts que rep com a entrada. A partir d'aquests triangles, on els seus vèrtexs són definits segons l'equació d'una esfera, es fa un escombrat de 0 a 90 graus en un eix i de 0 a 360 en l'altre eix. Aquests passos permeten generar una esfera a base de triangles amb una resolució variable, que dependrà dels increments en l'angle en cada eix, , apèndix A.7.2.

El següent pas és representar el resultat del càlcul de les reflexions per a cada punt en una escala de color. On a partir d'un nombre real genera un codi RGB, apèndix A.7.3.

Llavors, la resolució de l'esfera correspondrà al nombre de punts que s'avaluen a la trajectòria del receptor, i el color de cada punt al resultat obtingut per a cadascun.

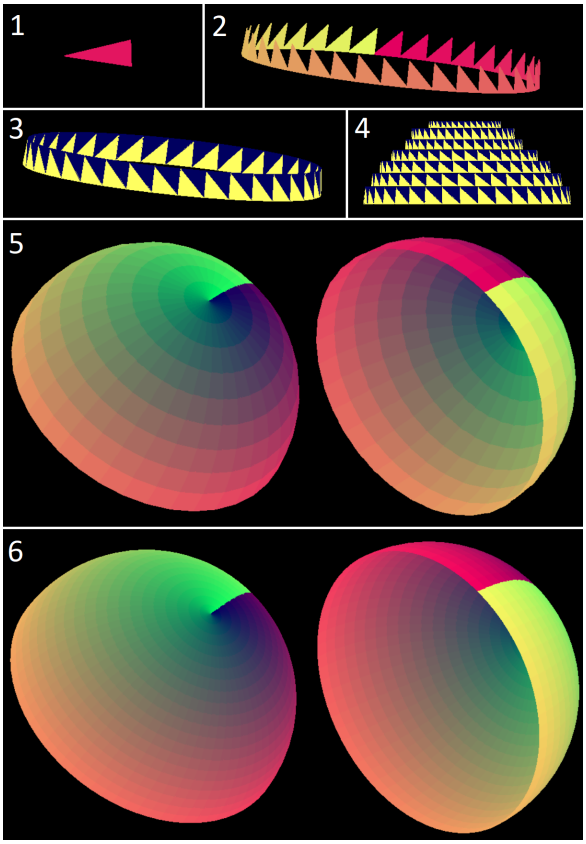

Fig. 14: En primer lloc, es mostra la unitat bàsica de resolució, el triangle. A la segona secció es genera una sèrie de triangles en forma circular. Posteriorment, s'afegeix la part superior a aquests triangles. Aquest procés es repeteix a diferents alçades com es mostra a la 4a secció. Finalment, s'ajusten les coordenades dels vèrtexs superiors amb els dels vèrtexs inferiors del següent nivell, per a passar d'una forma esglaonada a una forma esfèrica. A les seccions 5 i 6 es mostren dues figures amb resolucions diferents, 36x9 i 72x20 respectivament. El color de cada punt s'ha generat a partir de les seves coordenades

9 CONCLUSIONS

Pel que fa als resultats, es considera que s'assolit un bon nivell d'optimització al programa. Aquests mostren que la gestió i utilització adequada de les dades i dels recursos disponibles pot donar lloc a una gran millora en el rendiment. També, que prescindir de l'estructura i modularitat d'un programa pot millorar l'eficiència de l'execució. El *bottleneck* del rendiment és una porció, normalment petita, de programa. Per tant, prescindir d'aquesta modularitat, a una petita secció del codi, a canvi d'una possible gran millora en el rendiment és una bona estratègia a tenir en compte.

Cal esmentar que, definir de forma clara els requisits del programa i els objectius a assolir, i poder comunicar correctament els resultats és una part important per a aconseguir un progrés adequat.

El projecte ha sigut una bona introducció al desenvolupament i optimització del software. S'han posat en pràctica els coneixements adquirits al llarg dels cursos a les diferents etapes del treball i s'han adquirit uns de nous.

Per a un treball futur, es podria afegir la possibilitat d'utilitzar un nombre imparell de nodes per la generació de la superfície. També fer ús la llibreria SVML per vectoritzar les operacions trigonomètriques en lloc de fer servir l'aproximació Taylor.

AGRAÏMENTS

Voldria agrair al meu tutor, Juan Carlos Moure, pel suport i orientació rebuda durant aquest treball. A Serni Ribó per la proposta del treball i el seguiment durant el projecte. També als companys pel suport i les experiències durant la carrera.

REFERÈNCIES

[1] T. Elfouhaily, B. Chapron, and K. Katsaro, "A unified directional spectrum for long and short wind-driven waves," 1997. [Online]. Available: "https://www.researchgate.net/publication/23871550¨

[2] J. C. Maxwell, "A dynamical theory of the electromagnetic field," December 1864.

[3] A. Fresnel, "Ecuaciones de fresnel," 1820.

[4] S. Ribó, "Scattering simulation requirements document," December 2020.

[5] R. N. Bracewell and R. N. Bracewell, *The Fourier transform and its applications*. McGraw-Hill New York, 1986, vol. 31999.

[6] T. 5 University of Tennessee, Knoxville, "Mpi: A message-passing interface standard version 3.1," https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdfsection.2.3.

[7] @hkBattousai, "Implementation of sin/cos." [Online]. Available: stackoverflow.com/a/30908953/13047590

[8] "Openmp application programming interface," https://www.openmp.org/.

[9] "Getting started — opengl wiki,." [Online]. Available: https://www.khronos.org/opengl/wiki/Getting$_started$

A APÈNDIX

A.1 Estructures de dades inicials

```
struct sea_surface {

    //Coor superficie
    REAL * s_x;
    REAL * s_y;
    REAL * s_z;
    //vecotr normal  UNITARIO
    REAL * vec_L_x;  //
    REAL * vec_L_y;
    REAL * vec_L_z;
    //vector incidente
    REAL * vec_inc_x;
    REAL * vec_inc_y;
    REAL * vec_inc_z;
    //angulo incidente, && =reflejado
    REAL * angle;
    //vector reflejado
    REAL * vec_ref_x;
    REAL * vec_ref_y;
    REAL * vec_ref_z;
    //vector modulo incidente
    REAL * vec_module_inc;
    //vector modulo reflejado
    REAL * vec_module_ref;
    //vector ponderacio
    REAL * vec_ponderacio;
    //vector polarizacio
    std::complex<REAL> * polarization_total_x;
    std::complex<REAL> * polarization_total_y;
    std::complex<REAL> * polarization_total_z;
    //Vector campo electrico
    std::complex<REAL> * E_field;
};
```

Fig. 15

A.2 Codi inicial: generate_surface

A.2.1 Etapa 1: Generar matriu inicial

```
for(int i=0; i<Nx; i++){
  for(int j=0; j<Ny; j++){

    kx_vec = (double)(-Nx/2+i)*deltakx;
    ky_vec = (double)(-Ny/2+j)*deltaky;

    k_vec = sqrt(kx_vec*kx_vec + ky_vec*ky_vec);

    phi_vec = atan2(ky_vec, kx_vec);

    c_vec = sqrt(g/k_vec *(1.0+pow(k_vec/km, 2)) );

    Gmma_vec = exp(-(pow(sqrt(k_vec/kp)-1,2))/(2*sigma*sigma));
    Jp_vec = pow(gmma, Gmma_vec);

    Lpm_vec = exp(-1.25*pow(kp/k_vec,2));

    Fp_vec = exp(-(Omega/sqrt(10.0))*(sqrt(k_vec/kp)-1));

    Bl_vec = 0.5*alphap*(cp/c_vec)*Fp_vec;

    Fm_vec = exp(-0.25*(pow((k_vec/km) -1 ,2)));


    Bh_vec = 0.5 * alpham *(cm/c_vec)*Fm_vec;

    S_vec = pow(k_vec,(-3))*(Bl_vec + Bh_vec)*Lpm_vec*Jp_vec;

    Delta_vec = tanh(a0 + ap*pow(c_vec/cp,2.5) + am*pow(cm/c_vec,2.5));


  // ----- Computing the directional spectrum
    PHI_vec = 1.0/(2.0*M_PI) * (1.0 + Delta_vec * cos(2.0*phi_vec));

    PSI[i*Ny+j] = S_vec*PHI_vec/k_vec;
  }
}

  PSI[Nx*Ny/2+Ny/2]=0.0;
```

Fig. 16

A.2.2 Etapa 2: Fase aleatòria

```
void Generate_Random_fase(double *PSI_COMPLEX, double *PSI, int Nx, int Ny){

  for(int i=0; i < Ny; i++){
      PSI_COMPLEX[2*i] = sqrt(PSI[i]);
      PSI_COMPLEX[2*i+1]=0.0;
      }

  for(int i=0; i < Nx; i++){
      PSI_COMPLEX[2*i*Ny] = sqrt(PSI[i*Ny]);
      PSI_COMPLEX[2*i*Ny+1]=0.0;
      }

  for(int i=1; i < Ny/2; i++){
      double randomphase = 2.0*M_PI*(random()/(double)RAND_MAX);
      PSI_COMPLEX[Nx*Ny+2*i] = sqrt(PSI[Ny*Nx/2 +i])*cos(randomphase);
      PSI_COMPLEX[Nx*Ny+2*i+1] = sqrt(PSI[Ny*Nx/2 +i])*sin(randomphase);

       //simetric values as above
      PSI_COMPLEX[Nx*Ny+2*Ny-2*i] = PSI_COMPLEX[Nx*Ny+2*i];
      PSI_COMPLEX[Nx*Ny+2*Ny-2*i+1] = -PSI_COMPLEX[Nx*Ny+2*i+1];

    }

  PSI_COMPLEX[Nx*Ny+Ny]=sqrt(PSI[Nx*Ny/2+Ny/2]);
  PSI_COMPLEX[Nx*Ny+Ny+1]=0.0;

  for(int i = 1; i < Nx/2; i++){
    for (int j = 1; j < Ny; j++) {


          double randomphase = 2.0*M_PI*(random()/(double)RAND_MAX);
          PSI_COMPLEX[2*i*Ny+2*j]=sqrt(PSI[i*Ny+j])*cos(randomphase);
          PSI_COMPLEX[2*i*Ny+2*j+1]=sqrt(PSI[i*Ny+j])*sin(randomphase);

          //simetric values as above
          PSI_COMPLEX[2*Nx*Ny-2*(i-1)*Ny-2*j]=PSI_COMPLEX[2*i*Ny+2*j];
          PSI_COMPLEX[2*Nx*Ny-2*(i-1)*Ny-2*j+1]=-PSI_COMPLEX[2*i*Ny+2*j+1];

    }
  }
}
```

Fig. 17

A.2.3 Etapa 3: Transposició

```
void matrix_transform(double * PSI, int Nx, int Ny){

  double  aux_table;

  for(int i=0; i < Nx/2; i++)
    for(int j=0; j < Ny/2; j++){
        aux_table = PSI[i*Ny+j];
        PSI[i*Ny+j] = PSI[(i+Nx/2)*Ny+j+Ny/2];
        PSI[(i+Nx/2)*Ny+j+Ny/2] = aux_table;
  }



  for(int i=0; i < Nx/2; i++)
    for(int j=0; j < Ny/2; j++){
        aux_table = PSI[i*Ny+j+Ny/2];
        PSI[i*Ny+j+Ny/2] = PSI[(i+Nx/2)*Ny+j];
        PSI[(i+Nx/2)*Ny+j] = aux_table;
    }

  }
```

Fig. 18

A.2.4 Etapa 4: 2d FFT

```
void matrix_2d_FFT(double *PSI_COMPLEX, int Nx, int Ny){

  for (int i = 0; i < Ny; i++)
    gsl_fft_complex_radix2_backward(PSI_COMPLEX+2*i,Ny,Nx);
  for (int i = 0; i < Nx; i++)
   gsl_fft_complex_radix2_backward(PSI_COMPLEX+2*i*Ny,1,Ny);

  }
```

Fig. 19

A.3 Codi inicial: compute_surface_reflection

A.3.1 Bucle principal amb inline de la funció

```
init_normal_vector_surface(&surface, SIZE_XX, SIZE_YY, ...);

calculate_incidence_unit_vec(&surface, SIZE_XX, SIZE_YY, ...);

calculate_incidence_angle(&surface, SIZE_XX, SIZE_YY, ...);


for(int i =0; i< inclination; i++){

    for(int j =0; j< azimut; j++){


        sat_coor_receiver->x =  (RX_RADIO * sin(f(i))*cos(g(j)));
        sat_coor_receiver->y =  (RX_RADIO * sin(f(i))*sin(g(j)));
        sat_coor_receiver->z =  (RX_RADIO * cos(f(i)));

        init_polarization(polarization_ref, sin(f(j)), -cos(g(j)));

        calculate_reflected_unit_vec(&surface, SIZE_XX, SIZE_YY, ...);

        calculate_ponderacio(&surface, SIZE_XX, SIZE_YY, ...);

        calculate_propagation(&surface, SIZE_XX, SIZE_YY, ...);

        calculate_polarization(&surface, SIZE_XX, SIZE_YY, ...);

        for(int p=0; p <  SIZE_XX*SIZE_YY; p++)
            barrel_result[i*azimut + j] += surface.E_field[p].real();
    }

}
```

Fig. 20

A.4 Estructures de dades final

```
struct sea_surface {

    REAL * s_z;
}
```

Fig. 21

A.5 Codi final: generate_surface

A.5.1 Etapa etapes 1, 2 i 3: Posicions internes

```
//inner positions
for(int i=0; i<Nx; i++){
 for(int j=0; j<Ny; j++){


    rand = 2.0*M_PI*myIndexedRandom((Nx*start_XX_rank+i)*Ny+j, Nx*Ny*size_procs+Ny, &sign)*rseed;

    PSI = elfouhaily_spectrum_iter_calc((Nx*start_XX_rank+i+Nx*size_procs/2)%(Nx*size_procs), \
    (j+Ny/2)%Ny, wind, theta, Omega, Nx*size_procs, Ny, deltakx, deltaky);

    PSI_COMPLEX[i*Ny*2+j*2]    = PSI*cos(rand);   //TRASLATING REAL PART OF 1RST HALF OF MATRIX
    PSI_COMPLEX[i*Ny*2+j*2+1]  =  PSI*sin(rand)*sign;//TRASLATING IMAG PART OF 1RST HALF OF MATRIX
  }
}
```

Fig. 22

A.5.2 Etapa etapes 1, 2 i 3: Fila 0 inicial

```
if(start_XX_rank*Nx<=(Nx*size_procs/2) && (start_XX_rank+1)*Nx>(Nx*size_procs/2)){

  int i = (Nx*size_procs/2)%Nx;

  for(int j=0; j<Ny; j++){

      double PSI;

      PSI = elfouhaily_spectrum_iter_calc(0, (j+Ny/2)%Ny, ...);
      PSI_COMPLEX[i*Ny*2+j*2] = PSI;
      PSI_COMPLEX[i*Ny*2+j*2+1] = 0.0;

  }
}
```

Fig. 23

A.5.3 Fase aleatòria

```
double myIndexedRandom(int indx, int max_indx, int *sign) {

  *sign = (indx > max_indx/2) ? -1 : 1;
  double A = (double)(max_indx-indx)/(double)max_indx;
  double B = (double)(indx)/(double)max_indx;

  return  (indx > max_indx/2) ? A : B;
}
```

Fig. 24

A.5.4 Reordenació i transferència: MPI_ALLtoall

```
double * aux_buff = (double*)malloc((2*Nx*Ny/size_procs)*sizeof(double));

for(int p=0; p<size_procs; p++){
  for (int i = p*(Ny/size_procs),  s=0; i < (p+1)*(Ny/size_procs); i++, s++){
    for(int j=0; j<Nx; j++){

        aux_buff[s*2*Nx+p*2*Nx+j*2] = PSI_COMPLEX[j*2*Ny+i*2];
        aux_buff[s*2*Nx+p*2*Nx+j*2+1] = PSI_COMPLEX[j*2*Ny+i*2+1];

    }
  }
}


for (int i = 0; i < Ny/size_procs; i++){

  MPI_Alltoall(&aux_buff[i*2*Nx*size_procs], Nx, MPI_C_DOUBLE_COMPLEX,
    &PSI_COMPLEX[i*2*Nx*size_procs], Nx, MPI_C_DOUBLE_COMPLEX, MPI_COMM_WORLD);

}
```

Fig. 25

A.5.5 Condicions de contorn: MPI_Send/Recv

```
//Sending last & firstrow of each for normal vec calc
int prev=  (rank-1)%size_procs;
int next =  (rank+1)%size_procs;
if(prev == -1) prev = size_procs-1;

MPI_Isend(surface+(Nx*Ny/size_procs)-Nx, Nx, MPI_DOUBLE, next,...);
MPI_Recv(surface-Nx, Nx, MPI_DOUBLE, prev, 0, MPI_COMM_WORLD, &status);


MPI_Isend(surface, Nx, MPI_DOUBLE, prev, 1, MPI_COMM_WORLD, &request);
MPI_Recv(surface+Nx*Ny/size_procs, Nx, MPI_DOUBLE, next, 1, ...);
```

Fig. 26

A.6 Codi final: surface_reflection

A.6.1 Suma de resultats parcials: MPI_Reduce

```
MPI_Reduce(barrel_result, barrel_result_Reduce, azimut*inclination, \
                                    MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Fig. 27

A.7 OpenGL: Resultats gràfics

A.7.1 Dibuixar Triangle

```
struct RGB {

    float R;  //max value 1.0
    float G;
    float B;
};
void E_to_RGB(double * E_field, double max_Efield, double min_Efield, RGB * color) {

 double ratio = (E_field[0]-min_Efield) / (max_Efield-min_Efield);
    double inv_ratio = 1.0 - ratio;

    color->G = ratio * 1.00000f;
    color->R = 1.0f;
    color->B = inv_ratio * 1.00000f;


    if (E_field[0]==max_Efield)
      color->G = 0.0f;
};

```

Fig. 28

A.7.2 Dibuixar semiesfera

```
for (int j = 0; j < inc_max; j++)
{
   ...
   ...

    for (int i = 1; i < azimut_max; i++)
    {
        float x_1 = radio * cos(two_pi_sweep*((float)i / azimut_max) );
        float z_1 = radio * sin(two_pi_sweep * ((float)i / azimut_max));

        second_low_point[0] = x_1 * sin_theta_low;
        second_low_point[1] = radio * cos_theta_low;
        second_low_point[2] = z_1 * sin_theta_low;

        drawTriangle_sample(first_low_point, second_low_point, first_top_point,...);

        second_top_point[0] = x_1 * sin_theta_high;
        second_top_point[1] = radio * cos_theta_high;
        second_top_point[2] = z_1 * sin_theta_high;

        drawTriangle_sample(second_top_point, second_low_point, first_top_point,...);

        float* aux = first_low_point;
        first_low_point = second_low_point;
        second_low_point = aux;

        aux = first_top_point;
        first_top_point = second_top_point;
        second_top_point = aux;

    }
    ...
    ...
}
```

Fig. 29

A.7.3 Valor a RGB

```
struct RGB {

    float R;  //max value 1.0
    float G;
    float B;
};
void E_to_RGB(double * E_field, double max_Efield, double min_Efield, RGB * color) {

 double ratio = (E_field[0]-min_Efield) / (max_Efield-min_Efield);
    double inv_ratio = 1.0 - ratio;

    color->G = ratio * 1.00000f;
    color->R = 1.0f;
    color->B = inv_ratio * 1.00000f;


    if (E_field[0]==max_Efield)
      color->G = 0.0f;
};

```

Fig. 30

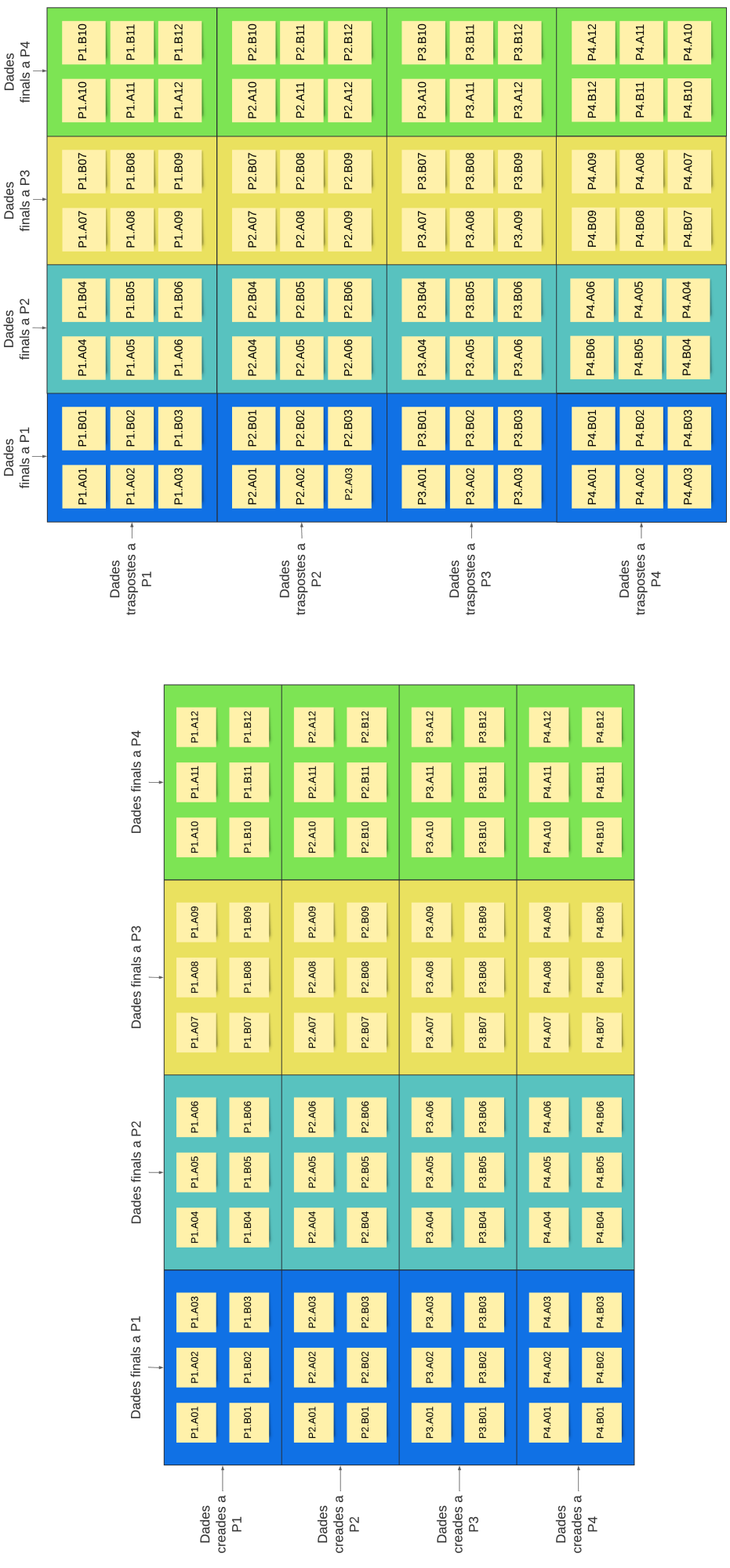

Fig. 31: Matriu de dades de la superfície. Cada fila horitzontal conté dades consecutives a memòria. Cada fila de 4 blocs indica les dades que es troben a cada node en un moment donat. Les columnes d'un bloc del mateix color indica les dades que tindrà cada procés després de realitzar les transferències de dades corresponents. El pas entre la primera i la segona matriu cada node fa una transposició per blocs que fa cada node amb les dades que disposa. Amb aquesta nova disposició de dades, el còmput posterior tindrà un patró d'accés més eficient i coherent amb la memòria. Les transferències de les dades entre node poden dur a terme abans o després de portar a cap la transposició.