UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Universitat Politècnica de València

MÀSTER EN COMPUTACIÓ PARAL·LELA I DISTRIBUÏDA

TREBALL DE MÀSTER

Aplicació de la computació d'altes prestacions i computació distribuïda en càlculs de Física Mèdica y radioteràpia basats en Monte-Carlo

Setembre 2017
Alumne: Vicent Giménez Alventosa
Tutor: Prof. Vicente E. Vidal Gimeno

Resum

Actualment el càncer és responsable del 25% de totes les morts de la UE i és la causa de mortalitat més important de les persones entre 45 i 64 anys. Per tant, la millora en els tractaments tindran un gran impacte en la societat. Per a tractar aquesta dolència, entre les tècniques d'avantguarda, destaca la radioteràpia, la qual s'ha convertit en un estàndard, ja siga sola o combinada amb altres tipus de tractaments. Avui en dia és possible realitzar simulacions precises per calcular la distribució de dosi depositada en un tractament de radioteràpia. No obstant, aquests càlculs empren tècniques de Monte-Carlo i són computacionalment costoses. Per aquest motiu s'empren aproximacions per a planificar els tractaments actuals, tot i que és ben conegut que aquestes aproximacions no són suficientment precises en molts casos. Al llarg d'aquest treball es pretén adaptar les simulacions de Monte-Carlo per a que puguen ser emprades en la planificació de tractaments clínics i, així, millorar-los emprant planificacions més precises, el que pot evitar efectes secundaris deguts a la radiació i millorar l'eficàcia dels tractaments.

Paraules clau: Càncer, Monte-Carlo, Radioteràpia, Simulació, PENELOPE, Física mèdica, Computació d'altes prestacions, Computació distribuïda.

Índex

1 Introducció

A aquest apartat, es plantejarà el problema a abordar en el present treball així com el per què resulta interessant i la seua possible futura repercussió.

1.1 Antecedents i motivació

L'enquesta més recent realitzada per l'Institut Nacional del Càncer [15] arriba a la conclusió de que aproximadament el 40% dels homes i dones de tot el món seran diagnosticats amb càncer en algun moment de la seva vida, i que el 34% d'ells morirà en 5 anys. Sols a la Unió Europea (UE), açò representa 2.6 milions de nous casos de càncer cada any, 1.3 milions de morts i 7.2 milions de persones que sobreviuen al càncer. Per tant, a la UE, el càncer és responsable del 25% de totes les morts i la causa de mortalitat més important de les persones entre 45 i 64 anys [8]. Per aquests motius, qualsevol millora en les modalitats de detecció, tractament del càncer i en la qualitat de vida dels pacients tindrà una gran incidència.

Entre les tècniques d'avantguarda, en el tractament de càncer, la radioteràpia s'ha convertit en una opció estàndard, ja siga sola o combinada amb altres formes de tractament. A més, els avanços en la protecció radiològica del pacient i en la seva qualitat de vida durant i després del tractament, són demandes socials que s'han de tenir en compte com objectius prioritaris del "Horizonte 2020" (H2020).

1.2 Descripció del problema

La radioteràpia és una tècnica terapèutica per al tractament de lesions tumorals que consisteix en irradiar la lesió oncològica amb radiació ionitzant (electrons, fotons, protons). En l'actualitat, la radioteràpia és una de les formes de tractament del càncer que està experimentant un desenvolupament tecnològic més significatiu. Els motius són, el baix cost, la baixa toxicitat quan es compara amb la quimioteràpia i, l'excel·lent tassa de curació i recuperació.

Per a impartir la dosi de radiació, que es defineix com l'energia absorbida per unitat de massa, a la lesió oncològica, a més de les tècniques d'imatge per a localitzar-lo, han d'aplicar-se models de transport de radiació a través del pacient. Aquests models, impliquen càlculs molt complexos que requereixen temps molt prolongats. En la pràctica clínica s'empren models analítics efectius que resolen l'equació de transport de radiació en un temps raonable.

El preu a pagar és una pèrdua en l'exactitud del càlcul.

El mètode considerat actualment com "gold standard" per a aquest càlcul és el mètode de Monte-Carlo. L'avantatge d'aquest model és que proporciona una solució exacta dins de la incertesa estadística inherent a dit mètode. El desavantatge, hui en dia, és l'excessiu temps de càlcul, fet que limita el seu ús en la pràctica clínica, quedant relegat a, principalment, investigació i verificacions pre i post tractament.

Els codis de Monte-Carlo d'ús general més aplicats en estudis de radioteràpia són MCNP [25], Geant4 [24], egs [7] i PENELOPE [22].

1.3 Estat de l'art

La braquiteràpia juga un paper important en el tractament d'una gran varietat de càncers, com els càncers ginecològics, de mama i pròstata. Aquesta tècnica empra llavors radioactives que són bàsicament menudes quantitats d'un cert isòtop radioactiu encapsulat en un recobriment de metall. Els tractaments amb aquestes llavors es divideixen en dos grans grups: un on les llavors són introduïdes temporalment en la zona afectada i després s'extrauen, i un altre on les llavors són introduïdes en la zona afectada i es queden permanentment. Per al segon tipus de tractament s'empren llavors on les partícules produïdes en la desintegració de l'isòtop tenen una energia relativament baixa. Com a conseqüència, ràpidament deixen de ser radioactives (semivida de l'ordre d'hores o dies).

Per a calcular la dosi dipositada en els distints òrgans pròxims al tumor i, al tumor mateix, s'empren planificadors. Aquests proporcionen informació als metges i radiofísics de com es distribuirà la dosi depositada per les llavors emprades en el tractament. Vegem un exemple a la figura 1.

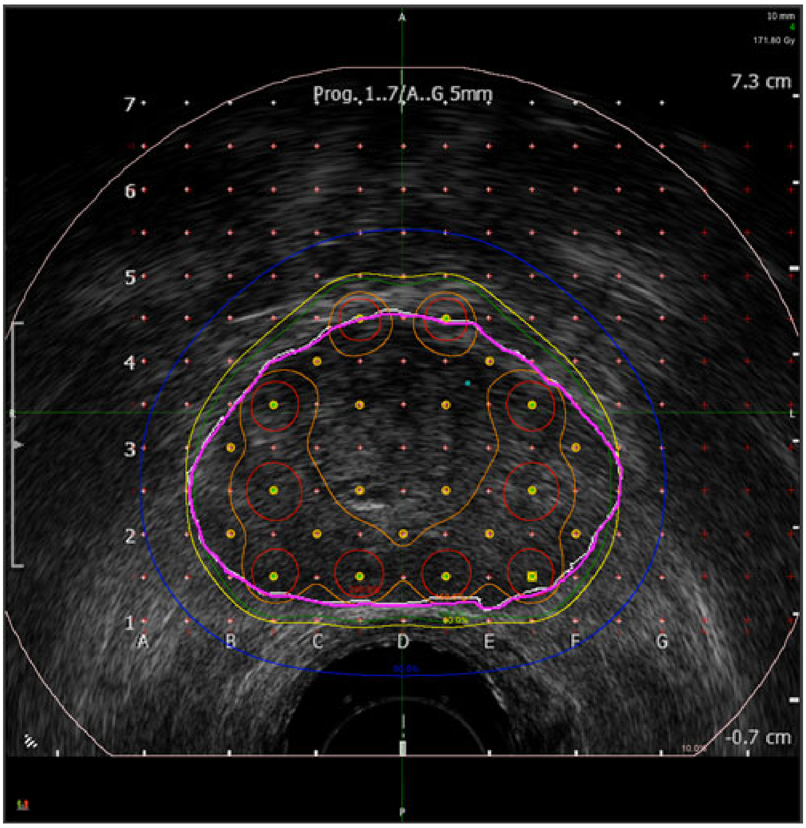


Figura 1: Exemple de planificador per a tractament de càncer de pròstata.

Aquests planificadors continuen emprant el formalisme proposat al “Task Group” No. 43 (TG-43) introduït en 1995 i modificat en publicacions posteriors [26], el qual s’ha convertit en un estàndard per a la dosimetria en braquiteràpia a nivell mundial. El formalisme descriu la dosi dipositada al voltant d’una única font posicionada al centre d’un “phantom” infinit d’aigua. En conseqüència, existeix equilibri de partícules carregades (CPE), exceptuant distàncies menudes pròximes a la càpsula de la font. L’aproximació per a estimar la deposició de dosi en un tractament de braquiteràpia consisteix en superposar les distribucions de dosi precalculades per a una única font en aigua. Aquesta aproximació és únicament vàlida per a un “phantom” homogeni d’aigua, però és un mètode ràpid i per aquest motiu s’empra en la clínica. No obstant, s’ignora la influència de l’heterogeneïtat del teixit, l’atenuació deguda a les llavors i la dimensió finita del pacient, el que trenca l’equilibri de partícules carregades.

És ben conegut que les suposicions del TG-43 no són precises en alguns casos clínics, especialment en la combinació de fotons de baixa energia

($<100keV$). A alguns teixits com l'òs, el coeficient d'atenuació màssic (μ_{en}/ρ) és significativament distint al de l'aigua, ja que, l'elevat nombre atòmic (Z) dels elements en l'estructura de l'òs provoca que l'efecte fotoelèctric siga la interacció predominant. Com a conseqüència apareix una major absorció de fotons poc energètics, la qual no es dona a l'aigua. Un altre exemple són les cavitats plenes d'aire com les observades en el càncer de mama o lesions de pulmó. Aquestes són un altra situació clínica on les suposicions del TG-43 no són vàlides.

Per aquest motiu els "model-based dose calculations algorithms" (MBDCAs), com el Monte-Carlo (MC) i models analítics com ACE (Advanced Calculation Engine - Nucletron - an Elekta Company, Veenendaal, The Netherlands) i ACUROS (Transpire Inc., Gig Harbor, WA), ambdós per a aplicacions amb llavors d'alta tassa amb Ir^{192}, han sigut considerats pel TG-186 [23] com un potencial substitut del formalisme del TG-43.

Els MBDCAs són capaços de tractar les diferents composicions/densitats dels teixits per a determinar la dosi dipositada. Usualment, els càlculs emprats per a la dosi absorbida es realitzen utilitzant la teoria de la cavitat, que permet un temps de càlcul ràpid. En aquesta, les dimensions de la cavitat es comparen amb el rang dels electrons secundaris [21]. Quan la cavitat és major que el rang dels electrons secundaris (en la majoria de casos que tractem), l'absorció de dosi en un teixit distint a l'aigua s'estima utilitzant quocients entre els coeficients d'absorció màssica de aigua i del teixit. Amb aquest factor precalculat, es transforma la dosi precalculada que es deposita en aigua en dosi dipositada en teixit. Aquesta aproximació suposa que la fluència energètica dels fotons en el punt d'interès és pràcticament la mateixa en l'aigua i en el teixit d'interès. En [27] es realitza un estudi que mostra que la suposició anterior no és certa per a certs teixits presents al cos humà, arribant a diferències d'un $50 - 72\%$ entre la fluència energètica de fotons en aigua i en òs i d'un 4% en teixit adipós.

La forma d'obtindre càlculs precisos de la dosi dipositada i evitar així els problemes anteriors és mitjançant simulacions de Monte-Carlo. No obstant, aquestes requereixen un temps de còmput molt superior, i l'objectiu d'aquest treball és intentar reduir aquest temps al màxim possible.

1.4 Objectius

L'objectiu principal d'aquest treball serà intentar adaptar les simulacions de Monte-Carlo per a que puguen ser emprades en l'ús clínic. Per a aconseguir-ho, es seguiran els següents passos:

- Estudiar i elegir un dels softwares de Monte-Carlo disponibles per a realitzar dita adaptació.
- Estudiar el format d'imatge mèdica estàndard DICOM (*Digital Imaging and Communications in Medicine*)[3] i com extraure l'informació necessària per a realitzar les simulacions.
- Realitzar un estudi dels paràmetres emprats en les simulacions per a arribar al millor compromís entre velocitat i precisió.
- Automatitzar tot el processat de la imatge mèdica, la posterior simulació, que dependrà del tractament oncològic, i el processat final dels resultats.
- Estudiar la millor forma d'optimitzar i paral·lelitzar cada apartat.
- Testejar el resultats amb casos d'estudi reals.

2 Conceptes previs

A aquest apartat discutirem alguns conceptes teòrics bàsics necessaris per a la comprensió del treball.

2.1 Simulació del transport de radiació

En aquesta secció descriurem els conceptes essencials de la simulació del transport de radiació aplicat a medis homogenis (gasos, líquids i sòlids amorfs) on suposem que les molècules es distribueixen de forma aleatòria amb una densitat uniforme. La composició d'un medi s'especifica mitjançant la seva fórmula estequiomètrica, és a dir, nombre atòmic Z_i i el nombre d'àtoms per molècula n_i de tots els elements presents. Amb aquestes dades, la massa molar (kg en un mol de material) de l'element ve donada per $A_m = \sum n_i A_i$ on A_i és el pes atòmic de l'element i-èsim. El nombre de molècules per unitat de volum ve donat per,

$$N = N_A \frac{\rho}{A_M}, \tag{1}$$

on N_A és el nombre d'Avogadro (nombre de molècules en un mol de material) i ρ la densitat de massa del material.

2.1.1 Secció eficaç d'interacció

Les partícules interaccionen amb els àtoms o molècules del medi a través de diferents mecanismes. Cada interacció es caracteritza per la secció eficaç diferencial (DCS), la qual és funció de les variables que caracteritzen l'estat de la partícula i es modifiquen en el transcurs de la interacció.

Per exemplificar, considerem un experiment on mesurem la dispersió d'un feix de partícules tal i com es mostra en la figura 2. Un feix monoenergètic de partícules amb energia E i direcció de moviment $\hat{\mathbf{d}}$ paral·lel a l'eix z que col·lisiona amb un àtom o molècula T. Considerem que el feix és homogeni i que la seua amplada és molt major que la dimensió de l'objectiu. El feix queda caracteritzat per la densitat de partícules per unitat d'àrea (perpendicular al feix) $\mathbf{J}_{inc}$. Assumim que les partícules interaccionen sols mitjançant un mecanisme, al qual, la partícula incident perd una quantitat d'energia W i és dispersada. Un detector que cobreix un diferencial d'angle sòlid $d\Omega$ detecta i compta totes les partícules que entren al volum sensible amb una energia en l'interval $(E - W - dW, E - W)$, és a dir, les partícules que han perdut una quantitat d'energia entre W i $W + dW$. Anomenem $\dot{N}_{count}$ al nombre de comptatges del detector per unitat de temps. La secció eficaç doble-diferencial (per unitat d'angle sòlid i energia perduda) es defineix com,

$$\frac{d^2\sigma}{d\Omega dW} \equiv \frac{\dot{N}_{count}}{|\mathbf{J}_{inc}| d\Omega dW}. \tag{2}$$

Com es mostra a l'equació anterior, la DCS té dimensions d'àrea/(angle sòlid x energia), el producte $[d^2\sigma/(d\Omega dW)] \cdot d\Omega dW$ representa l'àrea de la superfície d'un pla tal que, perpendicular al feix incident de partícules, és travessat per tantes partícules com les que són dispersades en la direcció $\hat{\mathbf{d}}'$ dins de $d\Omega$ amb una energia perduda entre W i $W + dW$.

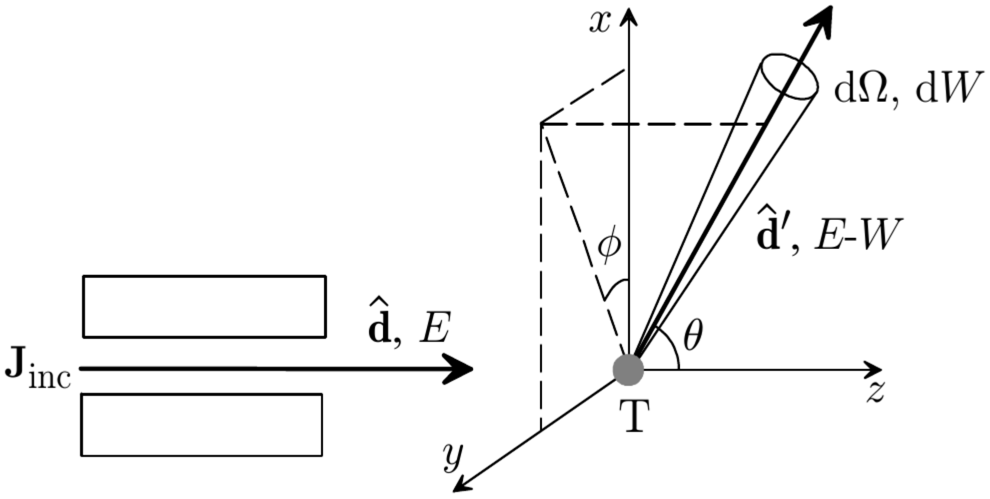


Figura 2: Esquema d'un experiment per a mesurar la DCS. Les partícules incidents es mouen en la direcció de l'eix z, θ i ϕ son els angles polar i azimutal dispersats, respectivament.

La secció eficaç total σ es defineix com la integral respecte a l'energia perduda, de la integral de la secció eficaç doble-diferencial sobre tot l'angle sòlid,

$$\sigma \equiv \int_0^E \frac{d\sigma}{dW} dW = \int_0^E \left(\int \frac{d^2\sigma}{d\Omega dW} d\Omega \right) dW. \tag{3}$$

Geomètricament, la secció eficaç total proporciona l'àrea d'un pla que, situat perpendicularment al feix incident, és travessat pel mateix nombre de projectils que interaccionen amb qualsevol dispersió angular i pèrdua d'energia resultant.

Es pot demostrar que, quan tenim diferents interaccions, cada una amb la seva secció eficaç, la secció eficaç total d'interacció (probabilitat de que interaccione amb qualsevol mètode), és la suma de les seccions eficaces de cada tipus d'interacció,

$$\sigma_T = \sigma_A + \sigma_B + \sigma_C + ... \tag{4}$$

2.1.2 Recorregut lliure mitjà

Considerem una partícula que es mou en un medi amb N molècules per unitat de volum. Volem determinar la funció distribució de probabilitat (PDF)

$p(s)$ de la longitud del recorregut s d'una partícula des de la seva posició actual fins al punt de la següent interacció.

En primer lloc estudiem un cas simple. Suposem un feix homogeni de partícules incidint perpendicularment a un material amb un gruix infinitesimal ds. Aquest material està constituït per àtoms o molècules distribuïts de forma aleatòria i uniforme. El feix de partícules veurà per tant Nds esferes per unitat de superfície. Si J és la densitat del feix incident, la densitat de partícules transmeses a través del material sense interaccionar és $J - dJ$, on dJ és el nombre de partícules que interaccionen per unitat de temps i superfície i es defineix com $dJ = JN\sigma ds$, sent σ la probabilitat total (suma de les seccions eficaces de totes les interaccions possibles) de que interaccione amb un àtom o molècula del material. Finalment, la probabilitat d'interacció per unitat de longitud serà el quocient de les partícules que interaccionen entre les que arriben i dividit per la distància recorreguda, és a dir,

$$\frac{dJ}{J}\frac{1}{ds} = N\sigma. \tag{5}$$

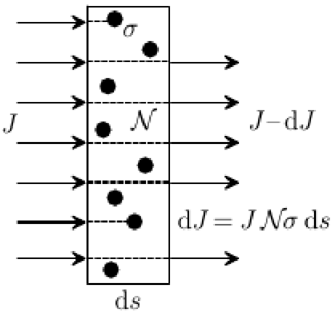


Figura 3: Esquema de l'atenuació d'un feix de partícules al travessar un material de gruix infinitesimal ds.

D'altra banda, la probabilitat de que una partícula recórrega una distància s sense interaccionar ve donada per,

$$F(s) = \int_s^\infty p(s')ds'. \tag{6}$$

La probabilitat $p(s)$ d'interaccionar al recórrer la distància entre el interval $(s, s + ds)$ equival al producte de $F(s)$ (probabilitat d'arribar a s sense interaccionar) i $N\sigma$ (la probabilitat d'interaccionar dins del ds),

$$p(s) = N\sigma F(s) = N\sigma \int_s^\infty p(s')ds'. \tag{7}$$

La solució de l'integral anterior, tenint en compte la condició de contorn $p(\infty) = 0$, és la família de la distribució exponencial, ja que, a l'integrar-se ha de donar ella mateixa.

$$p(s) = N\sigma e^{-sN\sigma}. \tag{8}$$

A partir d'aquest resultat obtenim el recorregut lliure mitjà com la distància mitjana entre col·lisions:

$$\lambda \equiv \langle s \rangle = \int_0^\infty sp(s)ds = \frac{1}{N\sigma}. \tag{9}$$

Donat que, en general, tenim diferents tipus d'interaccions amb les corresponents seccions eficaces, el recorregut lliure mitjà s'ha de calcular amb la secció eficaç total,

$$\frac{1}{\lambda_T} = N\sigma_T = N\left(\sigma_A + \sigma_B + \sigma_C + ...\right), \tag{10}$$

on hem aplicat el resultat de l'equació 4. Finalment obtenim que,

$$\frac{1}{\lambda_T} = \frac{1}{\lambda_A} + \frac{1}{\lambda_B} + \frac{1}{\lambda_C} + ... \tag{11}$$

2.1.3 Generació aleatòria de les traces

Cada traça de les partícules comença amb una posició, direcció i energia inicials que depenen de la configuració de la font. L'estat de la partícula queda definit per la posició $\mathbf{r} = (x, y, z)$, energia E i els cosinus directors de la direcció de vol $\mathbf{d} = (u, v, w)$, en el sistema de referència laboratori (sistema de referència que es situa sobre la partícula). Cada traça simulada es caracteritza per una sèrie d'estats $\mathbf{r}_n$, E_n, $\mathbf{d}_n$ on el subíndex n indica el número d'esdeveniment.

A continuació passem a explicar la generació de les traces. Suposem que la traça s'ha simulat fins a un estat $\mathbf{r}_n$, E_n, $\mathbf{d}_n$. La distància recorreguda fins a la pròxima col·lisió, el mecanisme d'interacció, el canvi de direcció i l'energia perduda en la interacció són variables aleatòries generades seguint la funció distribució de probabilitat corresponent.

Per a generar de forma aleatòria la distància recorreguda fins la següent interacció emprem la PDF donada per l'equació 8. Partim de la funció acumulativa de la corresponent PDF,

$$F(s) = \int_0^s \frac{1}{\lambda} e^{-s'/\lambda} ds' = 1 - e^{-s/\lambda}, \tag{12}$$

que és la probabilitat de sofrir una interacció al recórrer la distància entre 0 i s. Cal remarcar que la probabilitat està normalitzada a 1, $F(s) \in [0, 1]$. Calculant la funció acumulativa inversa, obtenim una funció que, donada una probabilitat acumulada, dona la distància recorreguda s corresponent,

$$F^{-1}(\zeta) = s, \ \ \zeta \in [0, 1]. \tag{13}$$

Partint de l'equació 12,

$$\begin{aligned} F(s) &= \zeta = 1 - e^{s/\lambda} \\ e^{-s/\lambda} &= 1 - \zeta \\ -\frac{s}{\lambda} &= \ln(1 - \zeta) \\ s &= -\lambda \ln(\zeta) \end{aligned} \tag{14}$$

on s'ha emprat que, al ser ζ una variable aleatòria tal que $\zeta \in [0, 1]$ es satisfà l'equivalència $1 - \zeta_1 = \zeta_2$ on ζ_2 és una variable aleatòria tal que $\zeta_2 \in [0, 1]$. Per tant, donat un valor aleatori uniforme $\zeta \in [0, 1]$ i substituint a l'equació 15 obtenim una variable aleatòria distribuïda segons la PDF donada en 8.

Una vegada calculada la distància recorreguda abans de la interacció, aquesta tindrà lloc a la posició,

$$\mathbf{r}_{n+1} = \mathbf{r}_n + s\hat{\mathbf{d}}_{\mathbf{n}}. \tag{15}$$

Per a elegir quin tipus d'interacció té lloc a aquest punt es calcula la probabilitat normalitzada de que tinga lloc cada interacció. Suposem, per exemple, que tenim dues interaccions possibles A i B,

$$\sigma_T = \sigma_A + \sigma_B \tag{16}$$

La probabilitat que tinga lloc la interacció A o la B ve donada per,

$$p_A = \frac{\sigma_A}{\sigma_T} \quad p_B = \frac{\sigma_B}{\sigma_T} \tag{17}$$

Donat que sempre s'acomplirà que $p_A + p_b = 1$, l'elecció de la interacció es realitza generant un nombre aleatori $\zeta \in [0, 1]$.

Finalment es passa a calcular la nova direcció i l'energia perduda en la interacció (figura 4). La nova direcció es calcula obtenint de forma aleatòria els angles azimutal i polar. L'angle azimutal es genera d'acord amb la distribució uniforme $\phi = 2\pi\zeta$. Açò pressuposa que el material no té estructura, sinó que és amorf i els àtoms i molècules estan distribuïts de forma aleatòria i uniforme. Si no fora així, la dispersió azimutal dependria de l'estructura i es deurien tindre en compte altres fenòmens físics. Aquesta aproximació és bona per a materials gasosos, líquids i sòlids amorfs o policristalins, però no monocristalls.

D'altra banda, la distribució de probabilitat de l'angle polar i de l'energia perduda dependrà del fenomen físic de la interacció elegida i de la modelització de dita interacció.

Després d'obtindre els valors de l'energia perduda W i els angles dispersats θ i ϕ, es redueix l'energia de la partícula $E_{n+1} = E_n - W$ i es canvia la direcció de moviment d'aquesta a $\hat{\mathbf{d}}_{n+1} = (u', v', w')$, obtinguda aplicant les matrius de rotació pertinents sobre l'antiga direcció de propagació $\hat{\mathbf{d}}_n = (u, v, w)$.

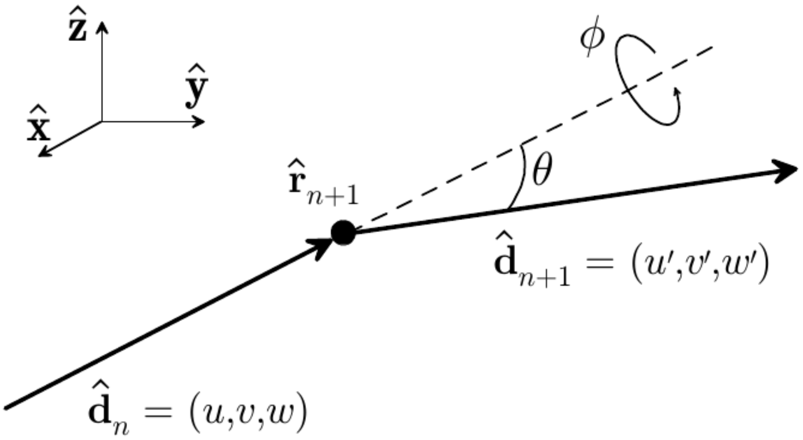


Figura 4: dispersió angular en una interacció de la traça.

Vegem clarament que la simulació de les traces de les partícules involucra prendre decisions a partir de nombres aleatoris, el que força necessàriament l'aparició de condicionals al codi. La simulació de la traça acaba quan, repetint aquest procés en bucle, la partícula sobrepassa el límit de la geometria de simulació o l'energia baixa d'un cert llindar definit per l'usuari E_{abs}.

2.1.4 Transport de partícules com processos de Markov

En el transport de les partícules suposem que, aquest, es pot modelitzar com un procés o cadena de Markov, és a dir, els futurs valors d'una variable aleatòria (procés d'interacció) estan determinats estadísticament pels esdeveniments actuals i depenen únicament de l'esdeveniment que el precedeix immediatament. Per tant, podem parar la generació de la traça d'una partícula en un estat arbitrari (qualsevol punt de la traça) i continuar la simulació sense introduir cap biaix al resultat.

Abans hem considerat el transport en un medi homogeni. En un cas real, en general, la traça de la partícula travessarà múltiples regions amb diferents materials. Durant la simulació, quan una partícula arriba a una interfície, aquesta es para i la simulació es reprèn amb les propietats del nou medi. Òbviament, aquest procediment és consistent amb la propietat Markoviana inherent al transport.

2.1.5 Valors mitjans i incerteses estadístiques

Per a exemplificar l'explicació, sense pèrdua de generalitat, considerem la simulació d'un feix d'electrons incidint sobre un "phantom" d'aigua semi-infinit. Cada electró primari, originarà una cascada d'electrons i fotons, les traces dels quals seran simulades fins arribar a la corresponent energia d'absorció. Qualsevol magnitud que es vulga calcular Q s'avalua com la mitjana d'un gran nombre N de cascades aleatòries simulades. Formalment Q es pot expressar com,

$$Q = \int q(x)p(x)dq, \tag{18}$$

on $p(x)$ representa la PDF d'un conjunt de variables x que determinen el valor $q(x)$. No obstant, aquesta distribució de probabilitat és, normalment, desconeguda. La generació de cascades individuals proporciona un mètode pràctic per a generar un conjunt de valors aleatoris de les variables x i del valor $q(x)$ associat. El valor Q estimat pel Monte-Carlo és,

$$\bar{Q} = \frac{1}{N}\sum_{i=1}^{N} q_i, \tag{19}$$

on q_i es el valor associat a la cascada i-èssima. Per exemple, l'energia mitjana dipositada al "phantom" d'aigua per electró incident és,

$$E_{dep} = \frac{1}{N}\sum_{i=1}^{N} e_i, \tag{20}$$

on e_i és l'energia dipositada per totes les partícules de la cascada i-èssima. L'incertesa estadística (desviació estàndard) del estimador de Monte-Carlo ve definida per,

$$\sigma_Q = \sqrt{\frac{var(q)}{N}} = \sqrt{\frac{1}{N}\left[\frac{1}{N}\sum_{i=1}^{N} {q_i}^2 - \bar{Q}^2\right]}. \tag{21}$$

on l'interval $\bar{Q} \pm 3\sigma_Q$ conté el valor real de Q amb una probabilitat del 99.7%. Cal remarcar que, per a que la propagació dels errors siga consistent, cada cascada ha de guardar els seus valors de q_i, i no barrejar els q de diferents històries. Açò s'haurà de tenir en compte a l'hora de paral·lelitzar el codi si volem simular històries de forma concurrent.

2.2 Simulacions amb espais de fase

Sovint ens trobem en situacions on la font emissora de partícules és comuna per a un gran nombre de simulacions. Per exemple, en l'àmbit clínic, un tractament amb radiació externa empra acceleradors amb els que es radia al pacient i/o aplicadors. Aquests, són comuns a una gran varietat de tractaments d'un mateix hospital. Un altre exemple és el cas de la braquiteràpia, on existeixen un nombre reduït de models comercials de llavors radioactives per a cada tipus de tractament.

Simular les fonts abans mencionades sol ser un procés molt costós, ja que, depenent del cas, degut a les dimensions reduïdes de les llavors, les simulacions han de ser detallades i no es poden aplicar certes aproximacions, especialment al transport d'electrons. Per aquest motiu s'empren els anomenats espais de fase, els quals són fitxers que guarden la informació de les partícules inicials que serviran d'entrada en futures simulacions.

El mecanisme consisteix en simular la font amb tota la seva geometria de forma detallada i creant un fitxer d'espai de fase amb les variables que caracteritzen l'estat de les partícules que escapen de dita geometria. Aquest fitxer s'emprarà per a un gran nombre de simulacions posteriors que compartisquen aquesta font. Exposem com a cas pràctic el que s'emprarà en aquest treball com a test. Disposem d'un DICOM amb una imatge d'ultrasons d'un escàner de pròstata on tenim 71 llavors radioactives, totes del mateix model (6711-OncoSeed). Per a crear l'espai de fases dissenyem la geometria de la llavor, la qual vegem a la figura 5.

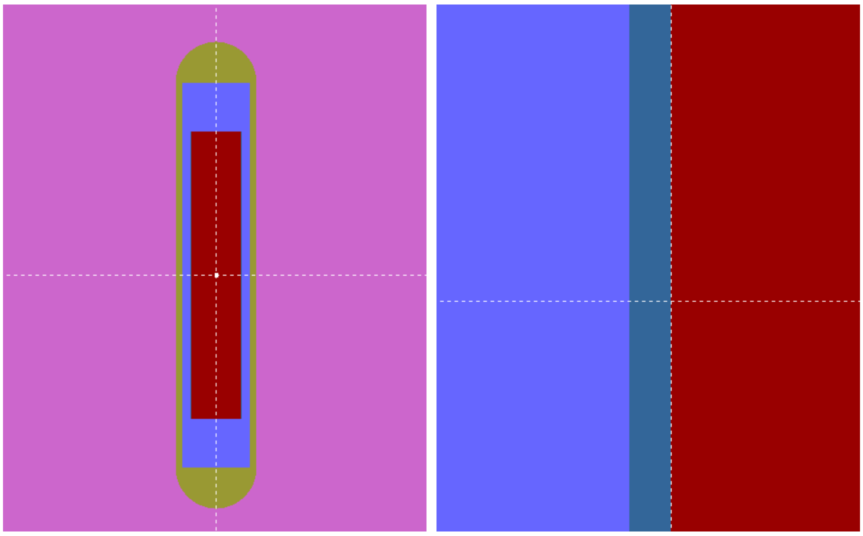


Figura 5: Geometria de la llavor 6711-OncoSeed amb element radioactiu I^{125}. Cada color correspon a un material distint, on el rosa és un material "detector" que absorbeix les partícules per a ser registrades al fitxer d'espai de fase. A la dreta vegem la imatge ampliada per a apreciar la pintura que conté el material radioactiu situat sobre el cilindre roig.

La font de partícules és la pintura radioactiva situada sobre el cilindre roig (figura 5 dreta), amb un gruixut de $1.75 \cdot 10^{-4}$ cm. Les partícules que arriben a l'exterior de la llavor (material rosa) són absorbides i guardades en el fitxer d'espai de fase. Posteriorment, aquest fitxer serà emprat en les simulacions de tractaments que utilitzen aquest model de llavor.

Com que aquests fitxers guarden totes les variables d'estat de milers de milions de partícules (podem estar parlant d'entre $10^8 - 10^{10}$ partícules), es llegeixen les partícules quan cal simular-les, pel que el fitxer és accedit contínuament al llarg de la simulació.

Finalment cal tenir en compte que, recordant el que s'ha explicat a la secció 2.1.4, com que estem tractant en processos de Markov, el reprendre la simulació a partir de l'espai de fases no introduirà cap biaix al resultat. A part, si cada simulació que empre aquest fitxer és independent de la resta, no apareixeran correlacions estadístiques.

Aquesta tècnica permet augmentar de forma significativa la velocitat de la simulació. Com veurem posteriorment als tests, aconseguim més d'un factor 180 en la velocitat.

2.3 *Digital Imaging and Communications in Medicine* (DICOM)

Les imatges procedents d'escàners mèdics es guarden en un format d'imatge anomenat "Digital Imaging and Communications in Medicine" (DICOM), la informació completa del qual la podem trobar a [3]. Per a poder realitzar simulacions de Monte-Carlo en l'entorn clínic és necessari poder extraure la informació necessària dels fitxers amb aquest format d'imatge. Cada imatge està formada per un nombre N de fitxers que guarden els distints plans de la imatge tridimensional (en el cas que ens interessa d'imatges 3D), i un parell de fitxers amb informació addicional sobre el tractament, com els contorns realitzats pels metges, la posició i tipus de les llavors radioactives, si s'han implantat, etc. Tant la informació de la imatge com part d'aquesta informació addicional és necessària per a les simulacions. Llistem a continuació la informació requerida així com la etiqueta corresponent per a accedir a aquesta dins del DICOM:

- **"Modality"** $(0008, 0060)$: Aquest element guarda la modalitat del fitxer. Subdividim les modalitats en dos categories, imatge i no imatge. Els fitxers amb una modalitat d'imatge ens proporcionen informació del tipus d'escàner que ha pres la imatge. Únicament acceptarem imatges procedent d'escàners d'ultrasons (US) o tomografia computeritzada (CT).

  D'altra banda, les modalitats "RTSTRUCT" i "RTPLAN" guarden els contorns realitzats pels metges i la informació de les llavors radiactives implantades en el pacient respectivament.

- **"ImagePositionPatient"** $(0020, 0032)$: Cada fitxer amb modalitat d'imatge pot contindre, en general, un nombre distint de plans de la imatge. Aquest element ens permet conèixer l'origen x, y, z del primer pla del fitxer DICOM, per a, posteriorment, ordenar els plans.

- **"StructureSetROISequence"** $(3006, 0020)$: Aquest element el contenen els fitxers amb modalitat "RTSTRUCT" i conté informació dels contorns, com nom, descripció etc. Aquest element conté un vector on cada posició correspon a un contorn distint. Dins de cada contorn trobem el següent element d'interès:

  - “**ROIName**” $(3006, 0026)$: Proporciona el nom del contorn.

- “**ROIContourSequence**” $(3006, 0039)$: Aquest element el contenen els fitxers amb modalitat “RTSTRUCT” i conté la informació dels punts que formen cada contorn. Està format per un vector on cada posició correspon a un contorn distint.

  - “**ContourSequence**” $(3006, 0040)$: Conté un vector on cada element guarda la informació d’un pla en l’eix z.

    - “**ContourData**” $(3006, 0050)$: Està format per un vector amb els punts d’un determinat pla z del contorn actual. El format del vector és $(x_0, y_0, z_0, x_1, y_1, z_1, ...)$

- “**ApplicationSetupSequence**” $(300a, 0230)$: Conté un vector on cada element representa un tipus de llavor radioactiva. Aquest element es troba al fitxer amb modalitat “RTPLAN”.

  - “**ChannelSequence**” $(300a, 0280)$: Dins d’aquest element trobem un vector on cada posició representa un “canal” a través del qual es col·loquen les llavors radioactives. En braquiteràpia equivaldria a les agulles emprades per a posicionar les llavors.

    - “**BrachyControlPointSequence**” $(300a, 02d0)$: Aquest element conté un vector amb els diferents “checkpoints” del canal actual, els quals són punts dins d’aquest canal. Cada sub-element d’aquest element guarda un “timestamp” i una posició (x, y, z).

      - “**CumulativeTimeWeight**” $(300a, 02d6)$: Conté el “timestamp” de l’actual “checkpoint”.
      - “**ControlPoint3DPosition**” $(300a, 02d4)$: Conté la posició (x, y, z) de l’actual “checkpoint”.

- “**PixelSpacing**” $(0028, 0030)$: Dimensió x i y de cada vòxel.

- “**SliceThickness**” $(0018, 0050)$: Dimensió z de cada vòxel.

- “**PhotometricInterpretation**” $(0028, 0004)$: En els fitxers amb modalitat d’imatge, guarda el format dels píxels (RGB, monocromàtic, ...). És necessari, ja que no podrem transformar el valor dels píxels que no siguen monocromàtics.

- “**PixelRepresentation**” $(0028, 0103)$: Ens dirà si el valor del píxel és amb signe o sense signe.

- **"RescaleIntercept"** (0028, 1052): Al valor del píxel se li ha d'aplicar una recta de cal·libració, aquest valor és el pendent de la recta.
- **"RescaleSlope"** (0028, 1053): Aquest valor conté l'origen de la recta abans mencionada.
- **"SamplesPerPixel"** (0028, 0002): Conté el nombre de components d'un píxel. Sols tractarem imatges amb una component per píxel.
- **"BitsStored"** (0028, 0101): Nombre de bits reservats per a emmagatzemar informació en un píxel.
- **"HighBit"** (0028, 0102): Guarda la posició de l'últim bit. Amb aquesta informació comprovarem si els píxels estan emmagatzemats en "big-endian" o "little-endian".

Tot i que l'estàndard DICOM conté molta més metadata, únicament l'informació anterior és la que necessitarem per als estudis clínics.

2.4 Escàners emprats

En aquesta secció descriurem el procediment per a convertir els valors dels píxels dels fitxers DICOM a densitats (g/cm^3) i assignar un material a cada vòxel. Aquest pre-processat serà necessari per a realitzar simulacions sobre la imatge i dependrà de la modalitat de la imatge (CT o US).

2.4.1 Modalitat CT

Un escàner de tomografia computeritzada (CT), es constitueix bàsicament de tres parts principals, la font de raigs-X, el pacient o objecte a escanejar i el detector. Com vegem a la figura 6, el pacient es situa entre la font de raigs-X i el detector.

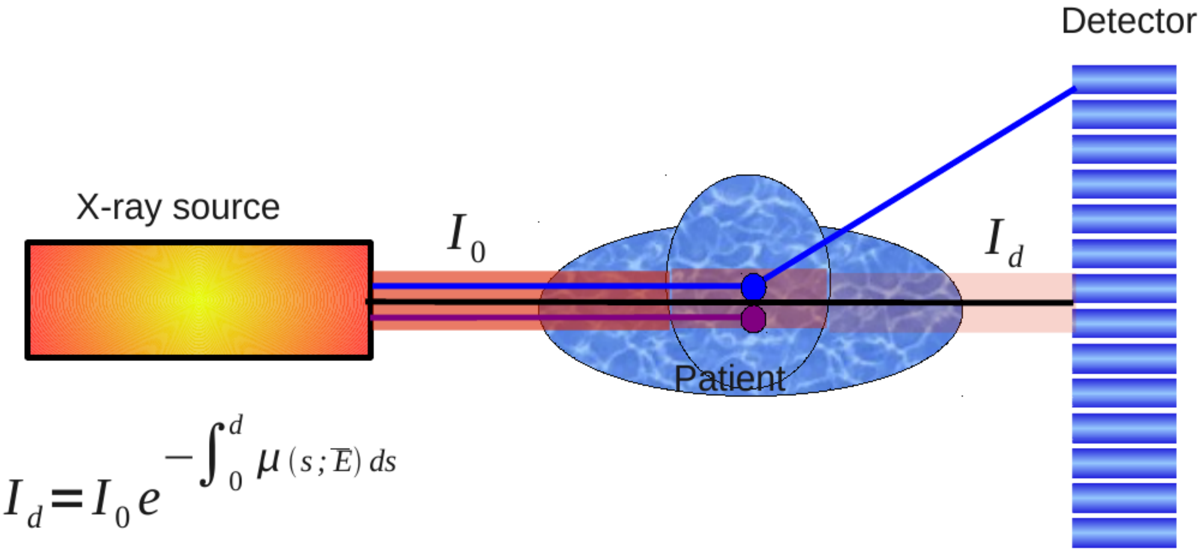


Figura 6: Esquema d'un escàner CT.

El feix de fotons emesos per la font serà parcialment absorbit o atenuat pel pacient. Aquesta atenuació segueix la llei exponencial,

$$I_d = I_0 e^{-\int_0^d \mu(s;E)ds}, \tag{22}$$

on I_d és la intensitat o nombre de fotons que arriba al detector, I_0 és la intensitat inicial de la font, μ és el coeficient d'atenuació, E l'energia dels fotons i d la distància recorreguda. El coeficient d'atenuació dependrà tant de la posició, ja que en cada zona el material i densitat d'aquest poden canviar, com de l'energia del fotó. Per tant, al realitzar varies projeccions rotant la font i el detector, aconseguim finalment un mapa dels coeficients d'atenuació en cada unitat de volum (vòxel) del pacient, el que proporciona una imatge estructural del seu interior.

En una imatge procedent d'una CT, el valor de cada vòxel, una vegada transformat mitjançant la recta formada pels elements del DICOM "RescaleIntercept" i "RescaleSlope" vists a la secció 2.3, s'associa a un valor adimensional en Unitats "Hounsfield" (HU). L'escala HU relaciona el coeficient d'atenuació d'un cert material amb el de l'aigua segons la fórmula,

$$HU = 1000\frac{\mu_t - \mu_{aigua}}{\mu_{aigua}}, \tag{23}$$

on μ_{aigua} és el coeficient d'atenuació de l'aigua destil·lada i μ_t el coeficient d'atenuació del teixit o material d'interès. Típicament, els escàners CT disposen d'una corba de cal·libració que permet transformar les HU a densitat electrònica relativa a l'aigua $\rho_{e,rel}$, definida com

$$\rho_{e,rel} = \rho N_g / \left(\rho N_g\right)_w , \tag{24}$$

on N_g és la densitat d'electrons per unitat de massa i el subíndex w denota aigua. Una vegada s'obté $\rho_{e,rel}$, aquesta es relaciona amb la densitat en g/cm^3 mitjançant la relació que podem trobar al TG-186 [23],

$$\rho = -0.1746 + 1.176\rho_{e,rel}. \tag{25}$$

Aplicant tot el procediment anterior per a cada vòxel, obtenim un voxelitzat tridimensional amb les densitats de cada vòxel. Finalment, se li associa un material a cada vòxel tenint en compte les relacions mostrades al TG-186 [23] entre densitat i material, les quals es mostren a la figura 7.

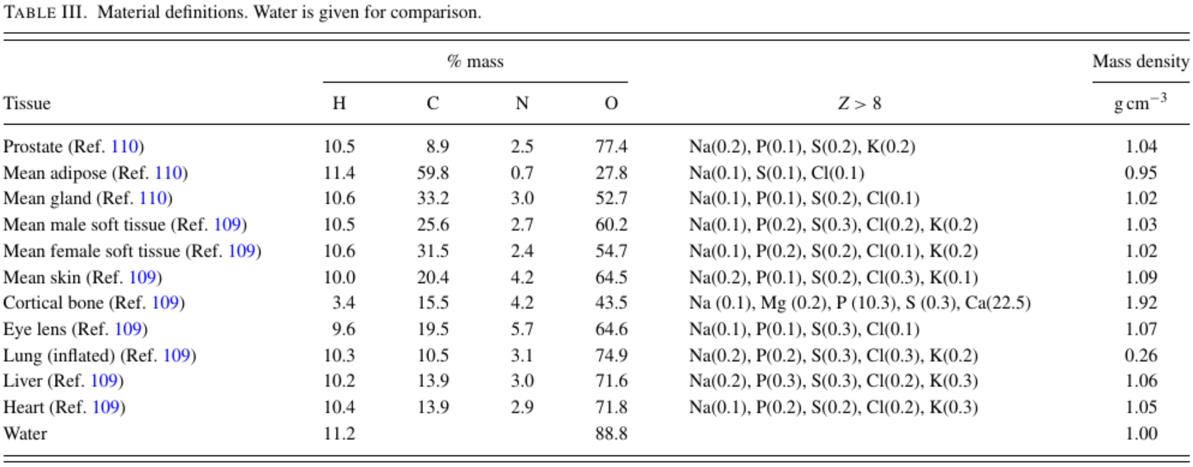


TABLE III. Material definitions. Water is given for comparison.

	% mass					Mass density
Tissue	H	C	N	O	$Z > 8$	g cm^{-3}
Prostate (Ref. 110)	10.5	8.9	2.5	77.4	Na(0.2), P(0.1), S(0.2), K(0.2)	1.04
Mean adipose (Ref. 110)	11.4	59.8	0.7	27.8	Na(0.1), S(0.1), Cl(0.1)	0.95
Mean gland (Ref. 110)	10.6	33.2	3.0	52.7	Na(0.1), P(0.1), S(0.2), Cl(0.1)	1.02
Mean male soft tissue (Ref. 109)	10.5	25.6	2.7	60.2	Na(0.1), P(0.2), S(0.3), Cl(0.2), K(0.2)	1.03
Mean female soft tissue (Ref. 109)	10.6	31.5	2.4	54.7	Na(0.1), P(0.2), S(0.2), Cl(0.1), K(0.2)	1.02
Mean skin (Ref. 109)	10.0	20.4	4.2	64.5	Na(0.2), P(0.1), S(0.2), Cl(0.3), K(0.1)	1.09
Cortical bone (Ref. 109)	3.4	15.5	4.2	43.5	Na (0.1), Mg (0.2), P (10.3), S (0.3), Ca(22.5)	1.92
Eye lens (Ref. 109)	9.6	19.5	5.7	64.6	Na(0.1), P(0.1), S(0.3), Cl(0.1)	1.07
Lung (inflated) (Ref. 109)	10.3	10.5	3.1	74.9	Na(0.2), P(0.2), S(0.3), Cl(0.3), K(0.2)	0.26
Liver (Ref. 109)	10.2	13.9	3.0	71.6	Na(0.2), P(0.3), S(0.3), Cl(0.2), K(0.3)	1.06
Heart (Ref. 109)	10.4	13.9	2.9	71.8	Na(0.1), P(0.2), S(0.2), Cl(0.2), K(0.3)	1.05
Water	11.2			88.8		1.00

Figura 7: Taula amb les definicions de diferents teixits presents al cos humà junt a la densitat (g/cm^3) de cada un. Extreta de [23].

2.4.2 Modalitat US

A partir d'una imatge procedent d'un escàner d'ultrasons (US) no tenim un mètode fiable per a assignar densitat als vòxels de la imatge ni, per tant, el material corresponent com el que el TG-186 ens proporciona per a CT. Per aquest motiu, la forma d'assignar el material i densitat als vòxels procedent d'una imatge d'US és utilitzant els contorns que realitzen els metges durant la presa de la imatge. Depenent del contorn que continga cada vòxel assignarem un material o un altre. El contornejat també servirà per a realitzar l'estudi clínic controlant la dosi dipositada a cada òrgan.

3 Material i mètodes

3.1 PENELOPE

Entre els codis de Monte-Carlo a l'abast dels investigadors, adequats per a radioteràpia, PENELOPE destaca per la seua versatilitat i filosofia de codi obert. PENELOPE s'ha desenvolupat durant els últims 20 anys a la Universitat de Barcelona i s'ha convertit en un dels estàndards en simulació Monte-Carlo de propòsit general per a la simulació d'electrons, positrons i fotons en materials i geometries arbitraries.

Compta amb una biblioteca de materials detallada que inclou dades de composició i paràmetres físics per a 280 materials extrets de la base de dades ESTAR [9], pel que es pot simular qualsevol combinació de materials.

Respecte al model de geometries complexes, PENELOPE incorpora dues possibilitats: els volums es poden obtindre com una combinació de superfícies quàdriques (superfícies definides com polinomis tridimensionals quadràtics) o com volums voxelitzats, és a dir, unitats de volum homogènies definides a partir de prismes rectangulars que poden obtenir-se directament de les tècniques d'imatge mèdica en la pràctica clínica (típicament una tomografia computeritzada). No obstant, no llegeix directament d'una imatge DICOM, s'ha de realitzar el preprocessat d'aquesta per separat.

Respecte al transport de partícules dins d'un material, PENELOPE també ofereix dos vistes distintes. Les partícules, poden ser simulades en un esquema mixt on les interaccions "hard" (aquelles amb una gran transferència d'energia o una gran deflexió) són simulades mentre que la resta d'interaccions "soft" s'agrupen segons un formalisme de dispersió múltiple, el qual accelera la simulació sense introduir biaix al resultat final. Com abans, aquests dos formalismes es poden combinar depenent dels requeriments de la simulació, el que permet una major acceleració de la simulació mentre es manté la precisió en les zones en que, a causa de la seua dimensió, la partícula no interaccione un nombre mínim de vegades per a que l'aproximació d'agrupar les interaccions "soft" siga correcta.

Les capacitats de personalització i adaptació abans mencionades són una de les diferències més importants respecte a la resta de codis de Monte-Carlo, els quals no implementen un esquema mixt de transport o ho fan de forma parcial.

Tot i que PENELOPE conté la implementació més completa de tots els processos físics rellevants per a la radioteràpia clínica, la seva implementació en un entorn hospitalari s'ha vist limitada per la dificultat d'optimitzar els càlculs de forma eficient per a reduir el temps de còmput.

Pels motius abans esmentats, s'ha elegit PENELOPE com a codi de Monte-Carlo base per a introduir aquest formalisme a la pràctica clínica.

3.2 Estructura del programa principal

A continuació s'explicarà l'estructura bàsica del programa principal necessari per a realitzar les simulacions. Podem veure un esquema a la figura 8. Per a una descripció més completa es referencia al manual [22].

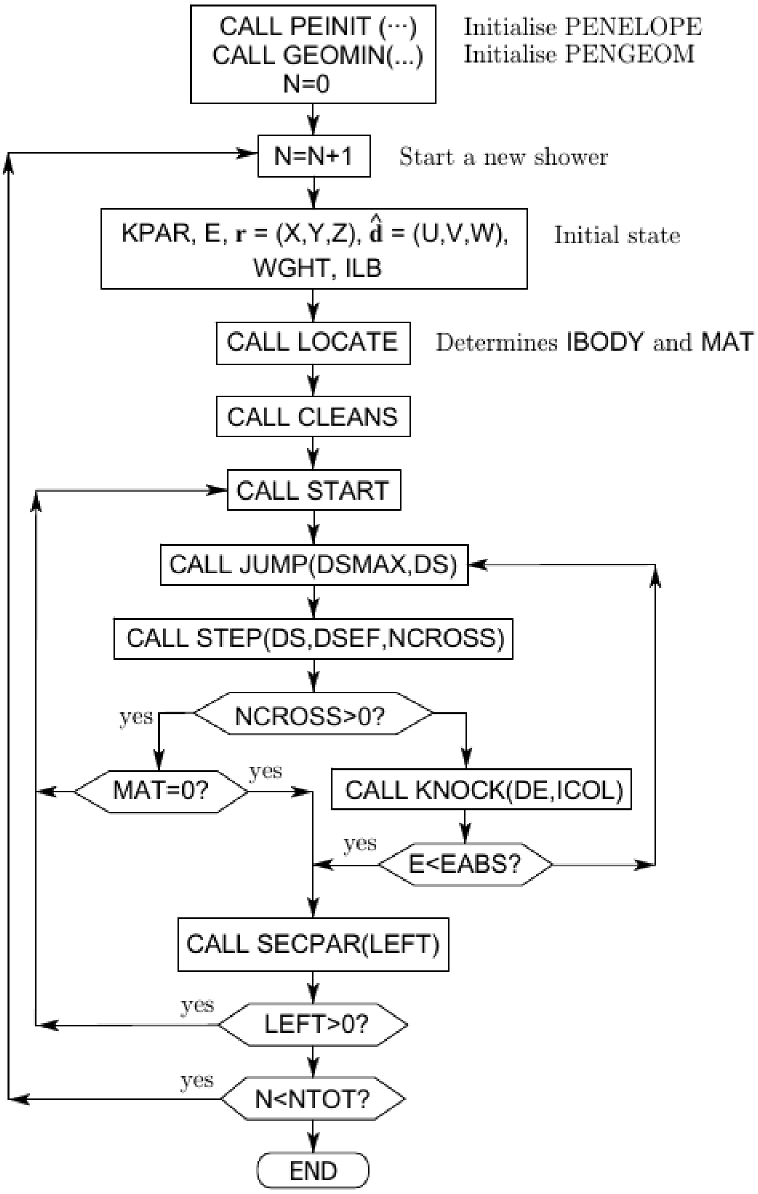


Figura 8: Esquema del programa principal per a realitzar les simulacions. Extret de [22].

La funció PEINIT s'executa a l'inici de la simulació per a llegir els diferents fitxers de materials i calcular les taules amb les propietats físiques rellevants per a la simulació actual. Aquestes dependran de les energies de les partícules que vagen a emprar-se durant la simulació. L'usuari haurà especificat altres paràmetres, com les energies d'absorció de les partícules, la caracterització de la font de partícules (posició, direcció, obertura, espectre d'energies...) etc.

La funció GEOMIN llegeix el fitxer amb la definició de la geometria, on tindrà lloc la simulació, i la crea.

Una vegada acabada la inicialització, comença la simulació. L'usuari ha de definir el nombre de partícules inicials o "històries" (*showers*) a simular, així com el tipus de partícula (electrons, fotons o positrons). Quan una d'aquestes històries comença, se li assignarà una posició, direcció i energia inicial depenent de la configuració definida per l'usuari. Començarà un bucle fins que es simulen totes les partícules inicials.

A continuació, la funció "LOCATE" calcula, a partir de la geometria creada en la inicialització del "GEOMIN" l'objecte i material on es troba la partícula, depenent de la seva posició.

Determinat l'objecte i material, la funció "CLEANS" neteja la pila de partícules secundàries. En aquesta s'emmagatzemaran les partícules creades a partir de les interaccions de la partícula que s'està simulant actualment. Una vegada aquesta siga absorbida o "escape" de la geometria, es passarà a simular la següent partícula de la pila de secundàries.

Tot seguit, comença un bucle que acaba quan totes les partícules secundàries generades han sigut simulades, és a dir, acaba quan la pila de partícules secundàries està buida. Quan açò passe, començarà la simulació de la següent història.

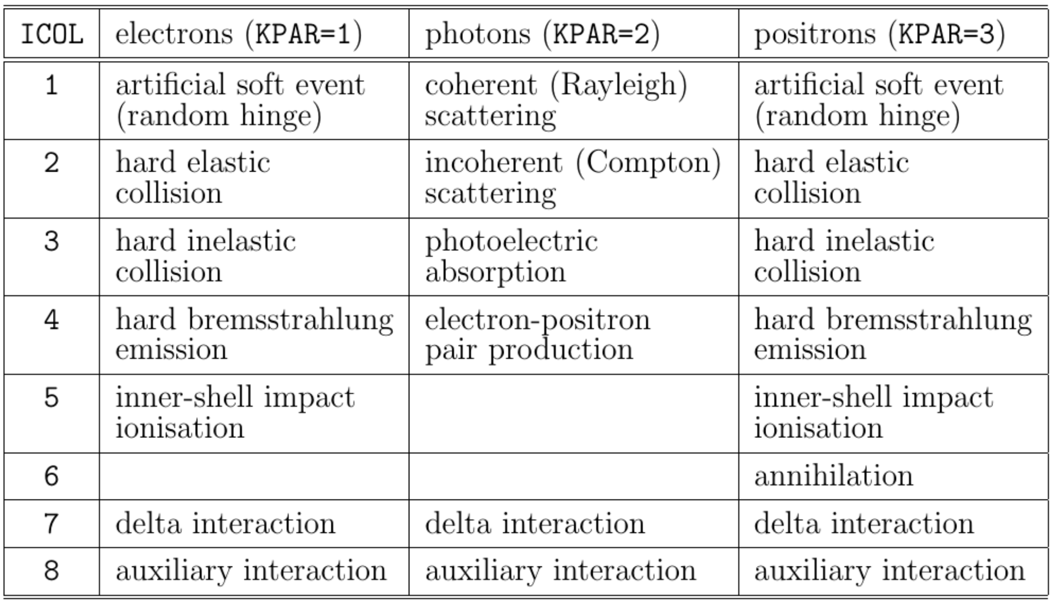
"START" prepara la simulació de la nova partícula, entre altres coses, comprovant que l'energia està dins dels rangs definits per l'usuari i dels rangs que PENELOPE pot simular (rangs de validesa dels models físics implementats).

Després de l' "START" comença un bucle on tenen lloc les interaccions, el qual finalitza quan la partícula és absorbida (perquè l'energia no arriba a un cert llindar definit per l'usuari) o escapa de la geometria definida per l'usuari.

La funció "JUMP" determina la distància que recorre la partícula abans de sofrir una interacció amb el medi ("DS"). Aquesta distància dependrà del material en el que es trobe així com el tipus i energia de dita partícula, tal i com s'ha vist a la secció 2.1.

A continuació, “STEP” comprovarà si la partícula ha travessat alguna de les superfícies que delimiten els diferents cossos de la geometria. Si és així, la variable “NCROSS” serà major que zero i es comprovarà si el material corresponent al nou cos on arriba la partícula és zero, el qual correspon al buit. Si ho és, la partícula no interaccionarà mai i, per tant, acaba la simulació d’aquesta. Si el material no és zero, caldrà tornar a executar l’ “START”, ja que, al canviar de material, el recorregut lliure mitjà amb el que s’ha calculat la distància que recorre la partícula i altres paràmetres físics també canvien. Per tant, s’ha de tornar a executar la funció “JUMP”. Tot aquest procediment no introdueix cap biaix en el resultat, ja que, com hem vist a la secció 2.1, el transport de les partícules és una cadena de Markov.

D’altra banda, si no s’ha travessat cap superfície, s’executarà la funció “KNOCK”, que simula les interaccions. Aquesta, depenent del tipus de partícula, elegirà de forma aleatòria la interacció (figura 9) que sofreix segons el mètode vist a la secció 2.1.3. Per a fer-ho, té en compte les seccions eficaces de cada interacció a l’energia de la partícula i en el material on es troba. Com a conseqüència d’aquesta interacció, la partícula podrà, en general, perdre energia i/o produir partícules secundàries. Per exemple, un fotó pot sofrir una interacció “Compton” amb un electró lligat a un àtom del material. Com a conseqüència s’arranca un electró de l’orbital atòmic i es reemet un fotó amb menor energia i distinta direcció. A més es podran produir fotons com a conseqüència de la relaxació atòmica dels orbitals més energètics. En definitiva es produirien com a mínim dues partícules. Les noves partícules generades s’emmagatzemaran en la pila de partícules secundàries a l’espera de ser simulades més endavant.



`ICOL`	electrons (`KPAR=1`)	photons (`KPAR=2`)	positrons (`KPAR=3`)
1	artificial soft event (random hinge)	coherent (Rayleigh) scattering	artificial soft event (random hinge)
2	hard elastic collision	incoherent (Compton) scattering	hard elastic collision
3	hard inelastic collision	photoelectric absorption	hard inelastic collision
4	hard bremsstrahlung emission	electron-positron pair production	hard bremsstrahlung emission
5	inner-shell impact ionisation		inner-shell impact ionisation
6			annihilation
7	delta interaction	delta interaction	delta interaction
8	auxiliary interaction	auxiliary interaction	auxiliary interaction

Figura 9: Identificadors de les diferents interaccions segons el tipus de partícula. Figura extreta de [22].

Una vegada la partícula interacciona, es comprova que l'energia final no siga menor que el llindar especificat per l'usuari. Si no ho és, es continuarà amb la següent iteració del bucle d'interaccions, tornant a la rutina “JUMP”. D'altra banda, si l'energia és suficientment baixa, la partícula serà absorbida en aquest punt i es cridarà a la funció “SECPAR” per a extraure l'última partícula guardada a la pila de secundàries i simular-la. Si no quedaren partícules a la pila, s'acabaria aquesta “història” i es passaria a la següent fins acabar amb totes.

A part de l'explicat anteriorment, entre les funcions mostrades a l'esquema s'invoquen els anomenats “tallys”. Aquests recullen informació que es vol extraure de la simulació. Per exemple, si volem obtindre l'energia dipositada en cada material, després de la funció “KNOCK” extrauríem l'energia dipositada per la partícula que ha interaccionat.

3.3 Adaptació del software

PENELOPE està escrit en una mescla de versions de FORTRAN 66, 77, 90 i 95. A més, conté zones de codi amb directrius en desús com “GOTO” i blocs de memòria global en forma de mòduls i “commons”, que dificulten la paral·lelització del codi, i l'ús de declaració implícita de variables. Per aquests motius es va decidir traduir completament PENELOPE a un llenguatge més modern, C++.

En una primera iteració de la traducció del codi, aquest s'ha traduït "literalment" des de la versió FORTRAN. El motiu d'açò és facilitar la comprovació del codi en C++ i poder reproduir resultats idèntics entre les dues versions emprant les mateixes llavors en el generador de nombres aleatoris. Els principals problemes que han aparegut durant la traducció han sigut:

- FORTRAN no distingeix entre minúscules i majúscules, al contrari que C/C++. Lamentablement, en la versió FORTRAN, hi ha parts de codi escrites completament en majúscules i altres parts en minúscules. Amb l'ajuda del compilador, tot i que tediós, aquest problema ha sigut relativament senzill de solucionar.

- A FORTRAN la primera posició d'un vector és la (1) mentre que en C/C++ és la [0]. Córrer els índex de tots els vectors del codi ha sigut una font important d'errors i, en general, no senzills de depurar.

- Blocs de memòria global, "commons" i mòduls: PENELOPE empra una gran quantitat de variables globals en forma de "common blocks" i mòduls. En aquests guarda, per exemple, les variables que defineixen l'estat de la partícula en simulació (energia, posició...), les variables que guarden els resultats de la simulació etc. A més, hi ha variables que comparteixen nom entre diferents blocs. Per aquest motiu, la solució que s'ha emprat és la de crear un "namespace" per cada "common" o mòdul. Actualment el codi consta de més de 95 "namespace". Dins de cada rutina s'emprarà la directriu "using namespace" per a cada bloc de variables emprat en dita rutina tal i com es fa de forma equivalent en la versió FORTRAN amb els "commons" i mòduls.

- La traducció de blocs de codi amb directius "GOTO" i "LABEL", aquestes directrius es troben sobretot a les parts més "antigues" del codi, com la part de lectura i creació de la geometria.

- L'ús al llarg de tot el codi FORTRAN de la declaració implícita de variables, la qual assigna el tipus de variable segons la primera lletra del nom d'aquesta sense necessitat d'especificar el tipus variable. L'ús d'aquest mecanisme junt a les variables globals ha provocat errors com declarar la mateixa variable varies vegades, ja siga perquè es tractava d'una variable global dins d'un dels "namespace" o perquè dins d'una mateixa rutina ja s'ha declarat abans. Aquests errors no sempre son detectats pel compilador, i s'han corregit durant les simulacions de testeig.

Actualment el codi traduït a C++ conté totes les rutines necessàries per a realitzar qualsevol tipus de simulació (rutines de geometria, de transport de les partícules, models físics, generador de nombres aleatoris...) així com la major part de funcionalitats exceptuant la creació d'espais de fase i alguns dels "tallys".

Feta dita traducció, s'ha comprovat que els resultats obtinguts en la versió en C++ són idèntics a la versió en FORTRAN. Els compiladors emprats han sigut el g++ i gfortran de GNU [10]. S'han realitzat simulacions específiques per a comprovar cada part del codi emprant la mateixa llavor per al generador de nombres pseudoaleatoris en ambdues versions.

3.4 Ampliacions de funcionalitat al codi original

Per a que la versió C++ del PENELOPE siga capaç de realitzar simulacions sobre casos clínics, el primer que necessita és poder extraure informació de fitxers amb format DICOM. Per a aconseguir-ho, hem emprat la llibreria de codi obert dicomsdl [5], la qual ens permetrà obrir i accedir a les dades dels DICOM. A l'apèndix A vegem la funció creada per a llegir i processar els DICOMs, en aquesta s'extrau la informació mostrada a la secció 2.3 com la posició de les llavors radioactives en un tractament de braquiteràpia i els contorns realitzats pels metges entre altres. Finalment, crea un voxelitzat apte per a simular sobre ell. Aplica les transformacions mostrades en la secció 2.4 per a assignar densitat i material als vòxels, depenent del tipus d'escàner.

A més, cal un "tally" capaç de post-processar les dades de la simulació per a generar les corbes d'isodosi sobre els plans del DICOM com hem vist a la figura 1 i, calcular la dosi dipositada als contorns realitzats pels metges per tal que els radiofísics i aquests avaluen si el tractament és correcte o cal modificar-lo. Per a fer aquesta tasca gràfica hem emprat el "gnuplot" [11] de forma que s'execute des del propi codi en C++ i quede tot automatitzat. A continuació explicarem el procediment per a obtindre aquestes dades.

En primer lloc, es requereixen una serie de dades especificades per l'usuari, aquestes són:

- Dosi prescrita per al tractament en unitats de Gy (Gray), que es defineixen com,

  $$1Gy = 1\frac{J}{Kg}. \tag{26}$$

  Teòricament, aquesta dosi ha d'arribar a tots els punts de l'òrgan a tractar.

- Activitat de la font en unitats de Bq (desintegracions/segon).
- Semivida de la font ($T_{\frac{1}{2}}$) en unitats de dies. Es defineix com el temps que tarda la font en disminuir la seua activitat a la meitat segons la llei de desintegració,

  $$A(t) = A_0 e^{-\lambda t}, \tag{27}$$

  on A_0 és l'activitat inicial de la font, $A(t)$ és l'activitat en l'instant t, λ és la constant de desintegració amb unitats de $1/segons$ i t el temps transcorregut. La semivida és un paràmetre típic per a caracteritzar les fonts radioactives.

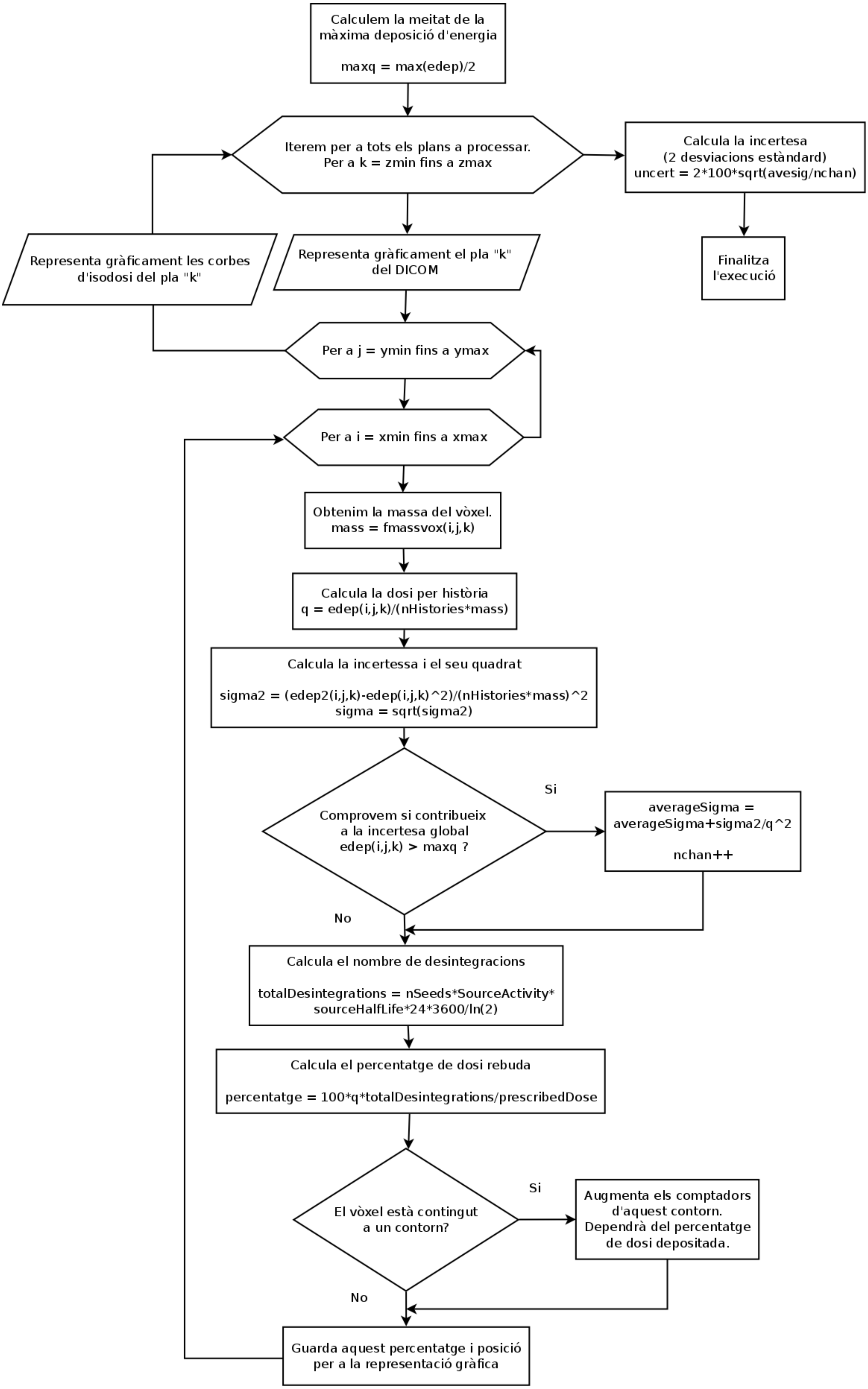
- Dosi mínima, dosi màxima i nombre de corbes d'isodosi. Els dos primers valors limiten el rang de les corbes d'isodosi i tenen unitats de tant % respecte a la dosi receptada.

Una vegada es realitza la simulació, s'obté d'aquesta la dosi depositada a cada vòxel del DICOM. Necessitarem com a paràmetres d'entrada:

- **dicomImage**: Guarda la imatge llegida del fitxer DICOM.
- **edep**: Matriu de vòxels que guarda l'energia depositada a cada vòxel.
- **edep2**: Matriu de vòxels que guarda la suma dels quadrats de l'energia depositada per cada història a cada vòxel.
- **prescribedDose**: Dosi prescrita pels metges.
- **nHistories**: Nombre total d'històries simulades.

- **xmin, ymin, zmin, xmax, ymax, zmax**: Índex dels vòxels que limiten la zona d'interès a analitzar.
- **dicomHeight**: Altura, en vòxels, de la imatge DICOM.
- **dicomWidth**: Amplada, en vòxels, de la imatge DICOM.
- **sourceActivity**: Activitat inicial, en desintegracions per segon, de la font.
- **sourceHalfLife**: Semivida de la font radioactiva, en unitats de dies.
- **nSeeds**: Nombre de llavors emprades en el tractament.
- **contourmatID**: Vector que guarda el material de cada contorn.
- **matvox**: Matriu que guarda el material de cada vòxel.
- **binSize**: Amplada dels bins, en tant per cent, que guarden el nombre de vòxels on s'ha depositat un cert % de dosi respecte la dosi prescrita en la variable "contourDat".

Per a superposar les corbes d'isodosi i calcular la dosi acumulada a cada contorn, emprem el codi mostrat a l'apèndix D, en el qual es segueix procés següent,


Calculem la meitat de la màxima deposició d'energia
maxq = max(edep)/2
Iterem per a tots els plans a processar. Per a k = zmin fins a zmax
Calcula la incertesa (2 desviacions estàndard)
uncert = 2*100*sqrt(avesig/nchan)
Finalitza l'execució
Representa gràficament les corbes d'isodosi del pla "k"
Representa gràficament el pla "k" del DICOM
Per a j = ymin fins a ymax
Per a i = xmin fins a xmax
Obtenim la massa del vòxel.
mass = fmassvox(i,j,k)
Calcula la dosi per història
q = edep(i,j,k)/(nHistories*mass)
Calcula la incertessa i el seu quadrat
sigma2 = (edep2(i,j,k)-edep(i,j,k)^2)/(nHistories*mass)^2
sigma = sqrt(sigma2)
Comprovem si contribueix a la incertesa global
edep(i,j,k) > maxq ?
Si
averageSigma = averageSigma+sigma2/q^2
nchan++
No
Calcula el nombre de desintegracions
totalDesintegrations = nSeeds*SourceActivity*sourceHalfLife*24*3600/ln(2)
Calcula el percentatge de dosi rebuda
percentatge = 100*q*totalDesintegrations/prescribedDose
El vòxel està contingut a un contorn?
Si
Augmenta els comptadors d'aquest contorn. Dependrà del percentatge de dosi depositada.
No
Guarda aquest percentatge i posició per a la representació gràfica

A les figures 10 i 11 vegem un exemple de superposició de les corbes d'isodosi i una gràfica de l'acumulació de dosi als diferents contorns del DICOM respectivament.

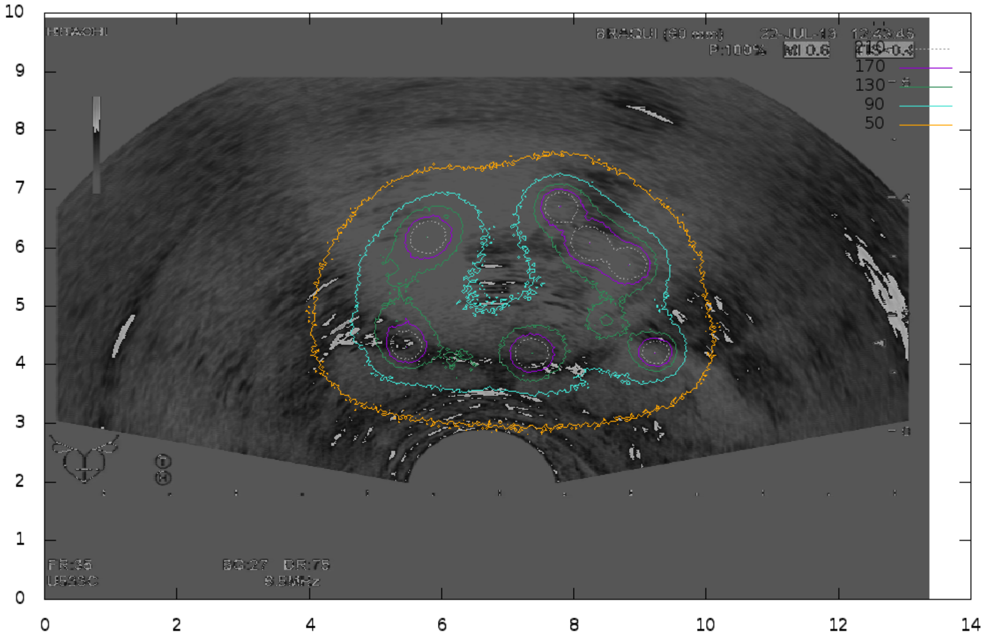


Figura 10: Corbes d'isodosi (% respecte de la dosi receptada) superposades a la imatge DICOM.

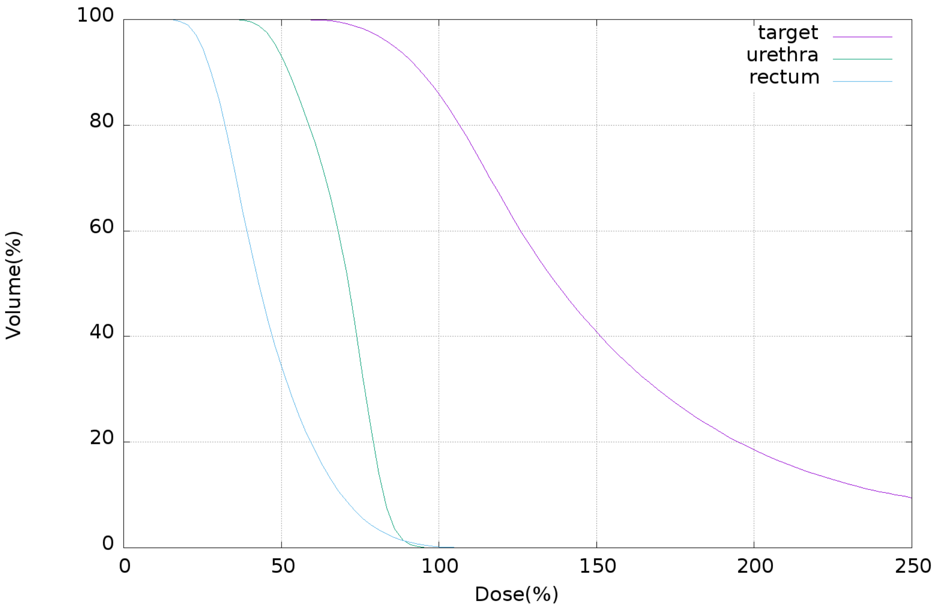


Figura 11: Volum de cada contorn (eix y) que rep, com a mínim, el percentatge indicat (eix x) de la dosi prescrita pel metge.

Una vegada implementades totes les funcionalitats necessàries per al cas clínic, passem a paral·lelitzar el codi per a que es realitze en un temps acceptable. A alguns tractaments el temps és crucial, per exemple, a braquiteràpia de pròstata el pacient està parcialment sedat al quiròfan mentre es realitza la planificació en eixe moment amb l'ajuda d'un escàner d'ultra sons. A açò s'ha d'afegir que el procés de planificació s'ha de repetir fins que la distribució de dosi és correcta i, en alguns cassos, el moviment del pacient pot forçar a que tot el procés haja de repetir-se. Per tant, és essencial que tant la inicialització, com la simulació, com el post-processat siguen el més ràpid possible. Veurem les diferents estratègies que s'han estudiat a les seccions següents.

4 Paral·lelització del codi

En primer lloc estudiarem com paral·lelitzar la pròpia simulació, ja que, normalment, aquesta requerirà la major part del temps de còmput. Per a aquest objectiu, ens hem recolzat al llibre [20]. Com ja hem vist a les seccions anteriors, cada història o cascada és independent de la resta i, per tant, poden ser simulades de forma concurrent. No obstant, apareix una limitació en el tractament de les incerteses, com hem vist a la secció 2.1.5. Cada història simulada de forma concurrent ha de mantindre comptadors independents, el que pot ocasionar problemes quan simulem el transport de la radiació sobre un voxelitzat gran (podem trobar entre 10^7 i 10^9 vòxels a les imatges clíniques), on cada vòxel ha de guardar el seu propi comptador d'energia dipositada per cada història concurrent.

4.1 Paral·lelització de la simulació amb GPUs

A priori, l'ús de GPUs per a la simulació del transport de les partícules presenta diferents inconvenients:

- Com hem vist a la secció 2.1.3, al codi trobem contínuament condicionals que ramifiquen el flux del programa a partir de nombres aleatoris. Tot i que no s'ha comentat abans, aquesta ramificació s'agreuja encara més quan es simulen les interaccions on, depenent del tipus d'interacció i, per tant, del model físic, es fan més eleccions a partir de nombres aleatoris. A part de les produïdes amb nombres aleatoris, també trobem altres ramificacions, per exemple depenent de l'energia i tipus de la partícula, de l'energia de l'anterior interacció etc. Eliminar tots els condicionals del codi resulta impossible, el que dificulta el seu ús amb GPUs.

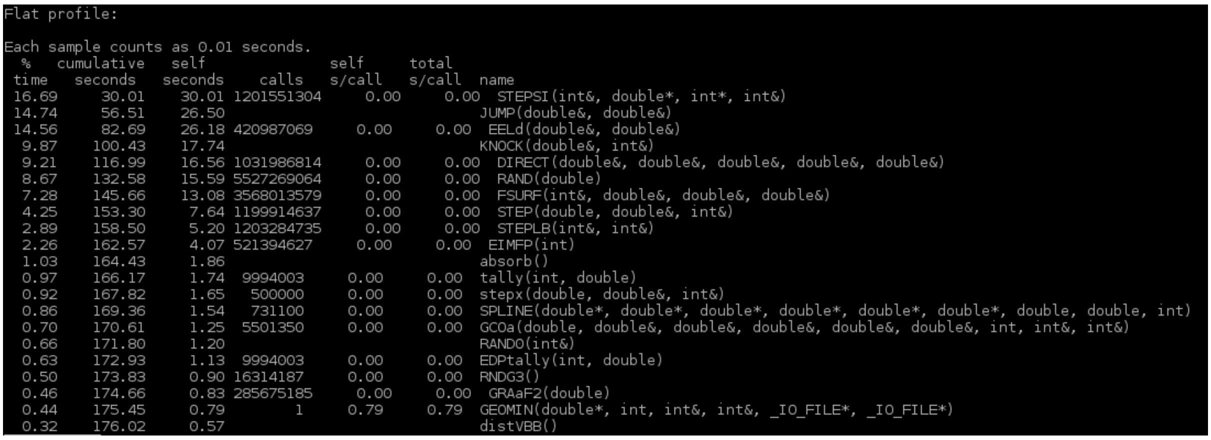
- Per a millorar el rendiment emprant GPUs deuríem simular, de forma simultània, un gran nombre de partícules. Com hem vist a la secció 2.1.5, per a que la propagació i estimació de les incerteses siga correcta i rigorosa cada història o "shower" ha de guardar els seus resultats parcials a una variable "privada" fins que acabe de simular-se. Després sumarà el seu resultat parcial i el quadrat d'aquest als comptadors globals corresponents. Per tant deuríem guardar comptadors per a cada història simulada simultàniament.

  Aquest fet, depenent del resultat que s'intente extraure, pot donar problemes o no. Per exemple, considerem que tenim 1000 simulacions simultànies. Si únicament necessitem extraure l'energia dipositada en els diferents materials que entren en joc en la simulació, típicament podem considerar 10, cada història simultània guardarà 10 variables de tipus *double* per al resultat parcial, el que serien $10 * 8 * 1000 = 80000$ *bytes*, que és un cost assumible. No obstant, per a un estudi clínic necessitem guardar la dosi dipositada a cada vòxel de la imatge. Per mostrar un exemple, considerem la imatge DICOM que emprarem més avant com a test. Aquesta pertany a la modalitat d'US, les quals són considerablement més menudes que les que provenen d'una CT de tòrax. Les dimensions d'aquesta imatge són $767x576x71$ vòxels, i per tant necessitaríem 240 Mb per a emmagatzemar el resultat d'una única simulació. Si sols simularem 100 històries simultàniament necessitaríem 23.4 Gb de memòria principal per a guardar tots els resultats parcials.

A més, cal tenir en compte que el cas anterior és un dels més favorables, una imatge provinent d'una CT pot contindre 10 vegades més plans que la que estem tractant. Una possible solució seria emprar formats per a matrius disperses, ja que, en molts casos, gran part dels vòxels no reben radiació o una quantitat molt menuda. Açò es donaria, per exemple, en un tractament que empra radiació externa procedent d'un accelerador, on el feix que incideix en el pacient està col·limat i té una direcció ben definida.

- Caldrà reestructurar el codi per a que siga capaç de simular de forma simultània un nombre predeterminat de "showers" i eliminar condicionals en la mesura del possible.

Tot i les dificultats abans exposades es va decidir intentar executar a la GPU únicament algunes de les rutines que més temps de còmput consumeixen emprant el llenguatge de nvidia CUDA [2]. S'ha utilitzat el "gprof" [12] per a obtindre un "profiling" d'una simulació de prova i comprovar així els temps d'execució de cada subrutina. Vegem el resultat a la figura 12.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 16.69     30.01    30.01 1201551304     0.00     0.00  STEPSI(int&, double*, int*, int&)
 14.74     56.51    26.50                             JUMP(double&, double&)
 14.56     82.69    26.18 420987069     0.00     0.00  EELd(double&, double&)
  9.87    100.43    17.74                             KNOCK(double&, int&)
  9.21    116.99    16.56 1031986814     0.00     0.00  DIRECT(double&, double&, double&, double&, double&)
  8.67    132.58    15.59 5527269064     0.00     0.00  RAND(double)
  7.28    145.66    13.08 3568013579     0.00     0.00  FSURF(int&, double&, double&, double&)
  4.25    153.30     7.64 1199914637     0.00     0.00  STEP(double, double&, int&)
  2.89    158.50     5.20 1203284735     0.00     0.00  STEPLB(int&, int&)
  2.26    162.57     4.07 521394627     0.00     0.00  EIMFP(int)
  1.03    164.43     1.86                             absorb()
  0.97    166.17     1.74  9994003     0.00     0.00  tally(int, double)
  0.92    167.82     1.65   500000     0.00     0.00  stepx(double, double&, int&)
  0.86    169.36     1.54   731100     0.00     0.00  SPLINE(double*, double*, double*, double*, double*, double*, double, double, int)
  0.70    170.61     1.25  5501350     0.00     0.00  GCOa(double, double&, double&, double&, double&, double&, int, int&, int&)
  0.66    171.80     1.20                             RANDO(int&)
  0.63    172.93     1.13  9994003     0.00     0.00  EDPtally(int, double)
  0.50    173.83     0.90 16314187     0.00     0.00  RNDG3()
  0.46    174.66     0.83 285675185     0.00     0.00  GRAaF2(double)
  0.44    175.45     0.79        1     0.79     0.79  GEOMIN(double*, int, int&, int&, _IO_FILE*, _IO_FILE*)
  0.32    176.02     0.57                             distVBB()
```

Figura 12: Eixida del "gprof" amb els temps d'execució de cada subrutina.

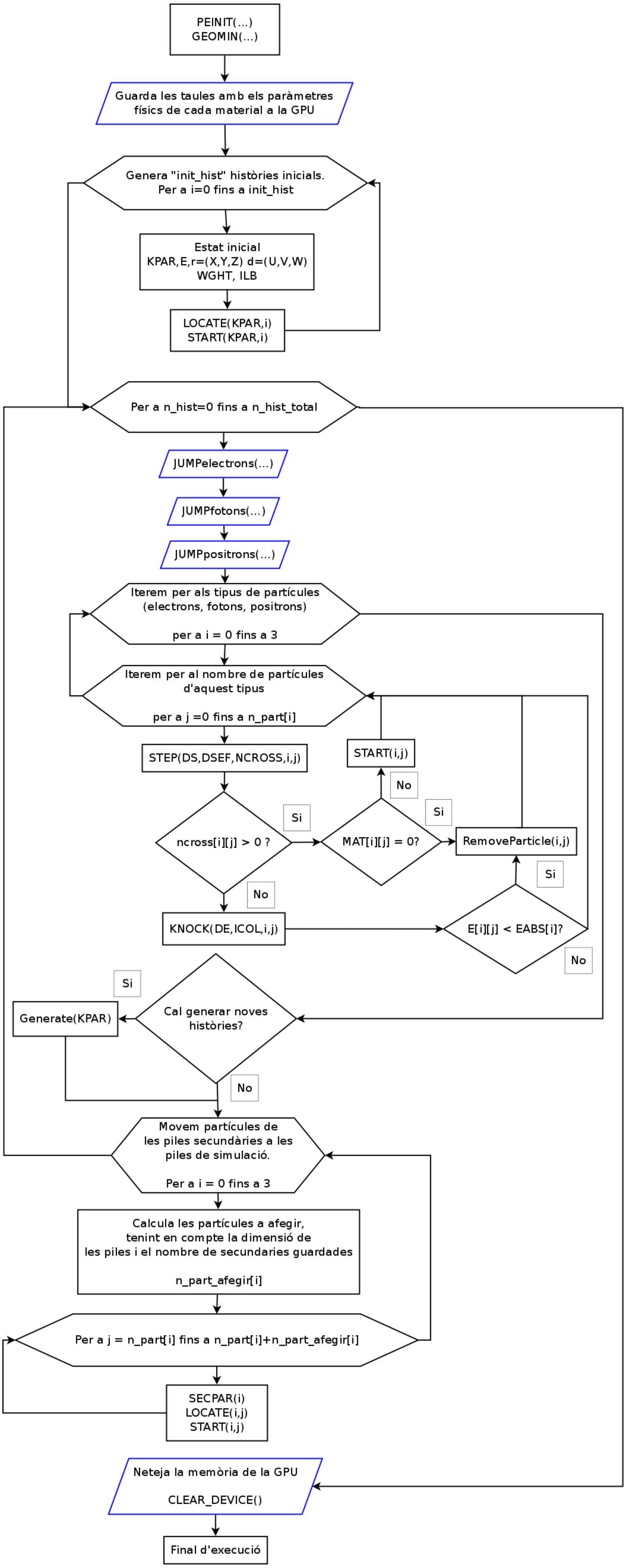
De les més costoses, la subrutina JUMP sembla la que podria donar un millor rendiment en GPUs, ja que és en la que més ramificacions podem evitar i és invocada directament pel programa principal.

El primer pas va ser reestructurar tot el codi per a que simulara simultàniament un nombre de partícules predefinit per l'usuari i separar la simulació en 3 parts, segons el tipus de partícula. Evitarem així les ramificacions inicials. Per a aconseguir-ho, cada variable que defineix l'estat de la partícula s'ha convertit en una matriu de dimensió 2D, nombre de tipus de partícules per nombre de partícules simulades simultàniament (per exemple $3x1000$ si simulem 1000 partícules concurrentment).

Tenim per tant tres piles de partícules que es simulen simultàniament: una per a electrons, una per a fotons i una última per a positrons. Igual passa amb la pila per a partícules secundàries. A més, l'usuari defineix un nombre màxim d'històries simultànies. Amb aquest mecanisme, es decideix la dimensió dels comptadors parcials de resultats i s'evita generar més partícules secundàries que les que caben a la pila. Cada partícula guarda a quina història o "shower" pertany, d'aquesta forma guardarà el resultat al comptador corresponent. Al final de cada bucle d'interacció es comprova si alguna pila de partícules en simulació té alguna posició lliure. Si és així s'extrauran les partícules del tipus corresponent de la pila secundària o, si aquesta estiguera parcial o totalment buida, es generaran noves històries.

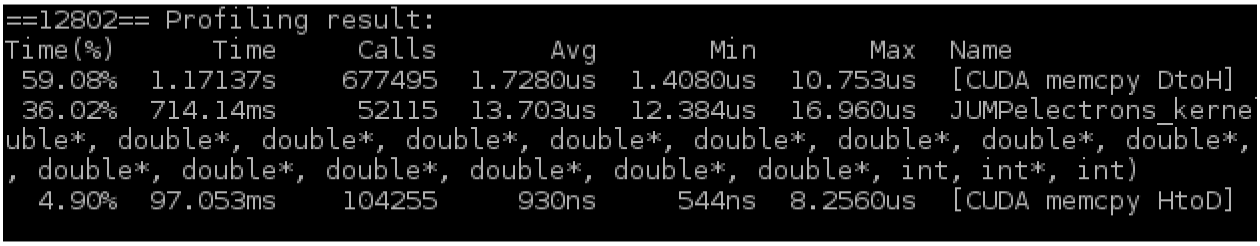
Una vegada reestructurat el codi i validat amb les mateixes simulacions de test que vam emprar en la validació de la versió C++, creem una funció JUMP específica per a cada tipus de partícula, a l'apèndix C podem veure l'exemple de la funció JUMP per a electrons en CUDA.

Podem veure un diagrama del flux a continuació, on el color negre indica l'executat a la CPU i el blau a la GPU.


PEINIT(...)
GEOMIN(...)
Guarda les taules amb els paràmetres físics de cada material a la GPU
Genera "init_hist" històries inicials. Per a i=0 fins a init_hist
Estat inicial KPAR,E,r=(X,Y,Z) d=(U,V,W) WGHT, ILB
LOCATE(KPAR,i)
START(KPAR,i)
Per a n_hist=0 fins a n_hist_total
JUMPelectrons(...)
JUMPfotons(...)
JUMPpositrons(...)
Iterem per als tipus de partícules (electrons, fotons, positrons)
per a i = 0 fins a 3
Iterem per al nombre de partícules d'aquest tipus
per a j =0 fins a n_part[i]
STEP(DS,DSEF,NCROSS,i,j)
START(i,j)
No
Si
ncross[i][j] > 0 ?
Si
MAT[i][j] = 0?
RemoveParticle(i,j)
Si
No
KNOCK(DE,ICOL,i,j)
E[i][j] < EABS[i]?
No
Si
Generate(KPAR)
Cal generar noves històries?
No
Movem partícules de les piles secundàries a les piles de simulació.
Per a i = 0 fins a 3
Calcula les partícules a afegir, tenint en compte la dimensió de les piles i el nombre de secundaries guardades
n_part_afegir[i]
Per a j = n_part[i] fins a n_part[i]+n_part_afegir[i]
SECPAR(i)
LOCATE(i,j)
START(i,j)
Neteja la memòria de la GPU
CLEAR_DEVICE()
Final d'execució

Per a minimitzar el temps de comunicacions, durant la inicialització del PENELOPE es transfereixen totes les taules amb els paràmetres de cada material a la GPU. Aquestes taules es mantenen constants durant tota la simulació i per tant no cal reenviar-les cada vegada. A més, com vegem a l'apèndix C s'han emprat "streams" per a solapar els temps de comunicació amb els de càlcul.

Per a fer els tests i comparar la velocitat, el maquinari emprat han sigut un processador Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz front a una GPU GeForce GTX 750. El resultat obtingut en emprar la GPU per a executar la rutina JUMP és que fa més lenta la simulació. És possible que estiga causat perquè, com vegem al segment del "profile" realitzat amb el nvprof a la figura 13, la major part del temps s'inverteix en comunicació des del dispositiu al host.



```
==12802== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 59.08%  1.17137s    677495  1.7280us  1.4080us  10.753us  [CUDA memcpy DtoH]
 36.02%  714.14ms     52115  13.703us  12.384us  16.960us  JUMPelectrons_kerne
uble*, double*, double*, double*, double*, double*, double*, double*, double*,
, double*, double*, double*, double*, double*, double*, int, int*, int)
  4.90%  97.053ms    104255     930ns     544ns  8.2560us  [CUDA memcpy HtoD]
```

Figura 13: Temps d'execució obtinguts del nvprof de CUDA.

El principal problema és que, tot i que les variables d'entrada són poques, s'extrauen molts resultats del "kernel". L'única solució aparent per a evitar aquest problema, és que no necessitem extraure tantes dades de la GPU i, per tant, que la major part dels càlculs es realitzen en aquesta.

Aquesta opció és la seguida en [28], on tradueixen el PENELOPE a C++, modifiquen tot el codi i l'adapten a CUDA.

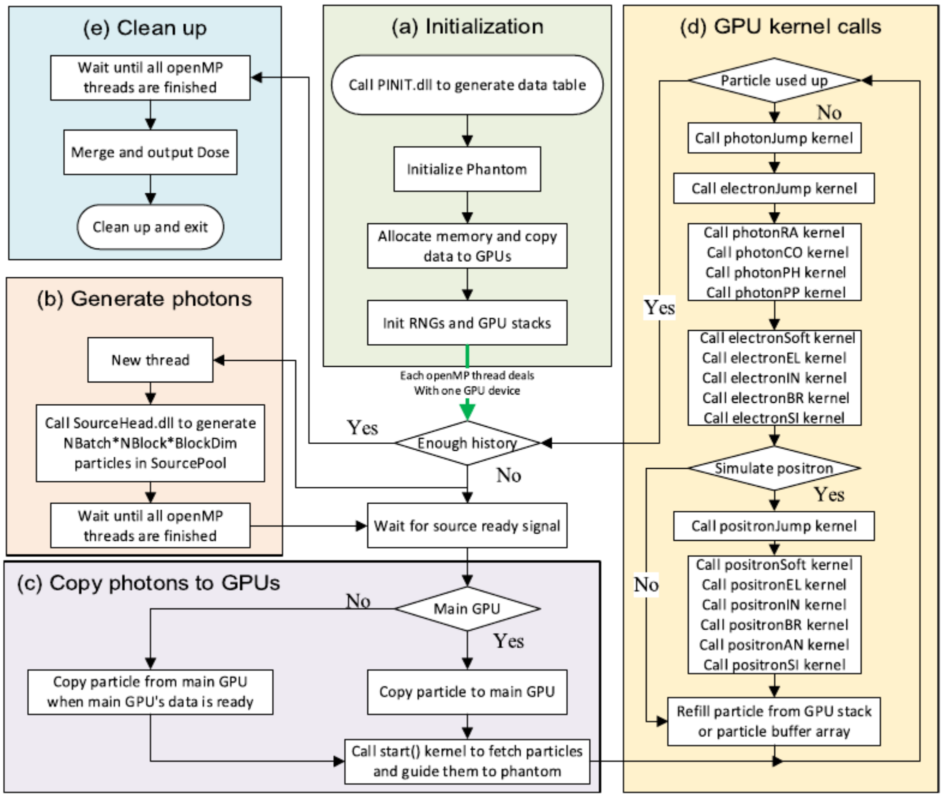


Figura 14: Flux del gPENELOPE extret de [28].

No obstant, l'adapten a un cas específic. Entre altres aproximacions, no admet geometries quàdriques i tracten els voxelitzats heterogenis com si foren uniformes per a aconseguir un recorregut lliure mitjà uniforme. Aquestes aproximacions poden ser vàlides per a fotons d'altes energies en un camp magnètic, com els que tracten a aquest article, però no ho és per al cas que estem estudiant, ja que no oferiria cap millora respecte al TG-43.

El maquinari emprat per a comparar els temps d'execució són, una CPU Intel Xeon E5 2630 v3 i una GPU NVIDIA Tesla K80. Tot i que la millora que diuen aconseguir és un factor 10, la targeta gràfica és entre 7 i 8 vegades més cara que aquest processador. Per aquest motiu creguem que si volem un codi completament general que no empre aquestes aproximacions en les simulacions i que mantinga totes les seues funcionalitats, el paral·lelisme amb GPUs no és la millor opció durant la simulació.

4.2 Paral·lelització de la simulació amb MPI

Un altra opció, és crear Nf fils ("threads"), emprant una llibreria com openMP [17]. En una mateixa simulació cada fil simularà una història, aconseguint simular Nf històries concurrentment. No obstant, degut al gran nombre de variables globals emprades al codi i que aquestes són accedides contínuament, es deurien gastar pràcticament totes les variables com a privades, pel que sembla més recomanable emprar directament una paral·lelització amb processos independents amb una llibreria del tipus openMPI [18]. A més, la comunicació entre processos únicament té lloc a la inicialització i al final de totes les simulacions parcials, i ens facilitarà l'ús en memòria distribuïda. Per tant, si s'executen Np processos, cada un simularà N/Np històries, en un sistema homogeni.

Una qüestió a considerar en aquesta divisió és l'efecte sobre el post-tractament. Aquest involucra un gran volum de dades ja que, com hem vist abans, un DICOM complet pot contenir entre desenes i centenes de milions de vòxels. El problema és que la dimensió del resultat és independent del nombre d'històries simulades, ja que és la dimensió total o parcial del DICOM, simule 10 històries o 10^{10}. Per tant, si subdividim la simulació en porcions més menudes, reduirem el temps que tarda la simulació en acabar però augmentarem el temps de post-processat, ja que hem d'unir tots els resultats parcials, fins al punt d'arribar a ser el nou coll de botella en quant a temps de còmput, en simulacions relativament ràpides. Tindrem en consideració aquest fet a l'adaptar les simulacions als temps requerits en aplicacions clíniques.

Tot i que es podrien llançar Np simulacions independents sense emprar MPI, simplement emprant un script amb "bash" que llance Np simulacions, els resultats parcials de cada simulació es deurien escriure en disc per a, una vegada acaben les simulacions, llegir, ajuntar i propagar les incerteses dels resultats. En casos clínics açò pot representar desenes o centenars de GB, depenent de l'escàner, tipus de tractament i nombre de divisions de la simulació principal (Np). A més d'evitar les escriptures i lectures innecessàries en disc, emprar openMPI ens permetrà que no tots els processos repetisquen els càlculs necessaris per a la inicialització, optimitzar el propi post-processat de les dades, que el codi siga fàcilment utilitzable en "clusters" o en diferents processos en una mateixa màquina i que quede tot unificat en un únic programa. Finalment, però no menys important, també ens permetrà fer un repartiment de la càrrega en temps d'execució per a optimitzar el codi en sistemes heterogenis, que són la majoria dels que disposem per a aquest tipus de càlculs.

4.2.1 Adaptació a MPI

Per adaptar el PENELOPEC a MPI, s'ha elegit crear un "wrapper" amb les funcions de lectura i escriptura per a que únicament el procés 0 realitze les lectures i ho envie a la resta de processos mitjançant les directives de comunicació del MPI. La definició i detalls de les funcions es mostren a l'apèndix B.

```
#define scanf(...) PENELOPE_MPI::mpi_scanf(__VA_ARGS__)
#define fscanf(...) PENELOPE_MPI::mpi_fscanf(__VA_ARGS__)
#define printf(...) PENELOPE_MPI::mpi_printf(__VA_ARGS__)
#define fgets(str, n, stream) PENELOPE_MPI::mpi_fgets(str, n, stream
   )
#define fopen(filename, mode) PENELOPE_MPI::mpi_fopen(filename, mode
   )
#define fclose(file) PENELOPE_MPI::mpi_fclose(file)
#define getpar() PENELOPE_MPI::mpi_getpar()
```

A part d'aquestes funcions, es divideix el nombre d'històries entre els processos i es suma el número de procés a les llavors del generador de nombres aleatoris per tal d'assegurar que totes són distintes i, per tant, els resultats siguen estadísticament independents. També s'han adaptat els "tallys" per a recollir els resultats mitjançant directives "MPI_Reduce", com vegem al segment de codi següent que reuneix l'energia dipositada a cada vòxel i la suma dels quadrats de l'energia depositada per cada història en cada vòxel,

```

int rank;
int p;

//take process rank and total process number
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

int total_voxels = (xvoxmin-xvoxmax)*(yvoxmin-yvoxmax)*(zvoxmin-
   zvoxmax);
if(total_voxels == geovoxMod::nvox)
{
  //all voxels are selected
  if(rank == 0)
```

```
   {
      MPI_Reduce( MPI_IN_PLACE, edep , total_voxels, MPI_DOUBLE,
          MPI_SUM, 0, MPI_COMM_WORLD);
      MPI_Reduce( MPI_IN_PLACE, edep2, total_voxels, MPI_DOUBLE,
          MPI_SUM, 0, MPI_COMM_WORLD);
   }
   else
   {
      MPI_Reduce( edep , edep , total_voxels, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
      MPI_Reduce( edep2, edep2, total_voxels, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
   }
}
else
{
   //Only some voxels are selected
   int size = xvoxmax - xvoxmin;
   for(int z = zvoxmin-1; z < zvoxmax; z++)
   {
      int indz = geovoxMod::nx*geovoxMod::ny*z;
      for(int y = yvoxmin-1; y < yvoxmax; y++)
      {
         int indy = geovoxMod::nx*y;
         int inipos = indz+indy+xvoxmin-1;
         if(rank == 0)
         {
            MPI_Reduce( MPI_IN_PLACE, edep , size, MPI_DOUBLE, MPI_SUM
                , 0, MPI_COMM_WORLD);
            MPI_Reduce( MPI_IN_PLACE, edep2, size, MPI_DOUBLE, MPI_SUM
                , 0, MPI_COMM_WORLD);
         }
         else
         {
            MPI_Reduce( edep , edep , size, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
            MPI_Reduce( edep2, edep2, size, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
         }
      }
   }
}
```

```
double total_nhist = 0;
MPI_Reduce(&n   , &total_nhist,  1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);

if(rank != 0){return;} //Only rank 0 process print global output
.
.
.
```

Com hem mencionat a la secció 2.2, si emprem espais de fase, no apareixen artefactes estadístics a l'utilitzar el mateix espai de fases en simulacions independents. No obstant, al subdividir una simulació, aquestes no són independents i, per tant, no poden emprar el mateix espai de fase. Per aquest motiu, la funció *mpi_ splitPhaseSpaceFile* de l'apèndix B llegeix i divideix l'espai de fases entre tots els processos. Aquest procés és llarg, ja que els espais de fases solen ser voluminosos, però, com que normalment s'emprarà el mateix espai de fase per a diferents simulacions, si tots els processos detecten una divisió prèvia del mateix espai de fase reutilitzarà aquesta. Per tant, per a casos clínics, aquests espais de fase es poden dividir prèviament per a optimitzar el temps.

A més, s'ha creat una nova funció $PENELOPE\_MPI :: mpi\_getpar()$ per a substituir a l'original $getpar()$. Aquesta funció llegeix una partícula de l'espai de fases i actualitza una serie de comptadors. El motiu és que $getpar()$ llegeix un espai de fases en format ASCII i la funció *mpi_ splitPhaseSpaceFile* crea els nous fitxers d'espai de fase en format binari per a optimitzar la lectura. Per tant, cal una nova funció per a extraure les dades de les partícules.

4.2.2 Balanceig de la càrrega

Com que, normalment, els "clusters" emprats per a realitzar les simulacions no són exclusius per a aquesta ni homogenis (són sistemes heterogenis), seria convenient emprar un sistema per a balancejar la càrrega segons la velocitat de cada procés. Amb aquest propòsit, cada cert temps predefinit per l'usuari, el procés amb "rank" 0 demanarà a la resta de processos que se li informe del nombre d'històries simulades fins ara junt a un interval de temps. Amb aquesta informació, el procés 0 recalcula el nombre d'històries assignada a cada procés MPI en funció de les velocitats relatives. Es reajusta així la càrrega de còmput en temps d'execució. Aquesta possibilitat no la tenim si dividim la simulació en múltiples processos sense comunicació entre ells.

En cas de que disposem d'un sistema homogeni o no volem emprar aquesta característica es pot desactivar.

En la inicialització, l'usuari especificarà diferents límits per a parar les comunicacions entre processos. Aquests són: El nombre màxim de comunicacions que es poden realitzar i el percentatge d'històries simulades a partir del qual, una vegada superat, es tallen les comunicacions, ja que considerem que la velocitat de les simulacions ja s'ha estabilitzat. A part, l'usuari ha d'especificar l'interval de temps entre comunicacions.

Després de la inicialització, cada procés MPI crearà dos fils emprant openMP [18]. Mentre un realitza la simulació normalment, l'altre servirà per a les comunicacions entre processos, com vegem al codi següent,

```
#ifdef _PENELOPE_WITH_MPI_
#ifdef _USE_OMP_
#ifdef _USE_LB_

printf("*** Load Balancer activated ***\n");
const char* s = getenv("OMP_NUM_THREADS");
int defaultThreads;
if(s != NULL)
{
   defaultThreads = atoi(s);
}
else
{
   printf(" No OMP_NUM_THREADS especified\n");
   defaultThreads = 0;
}
ctrsimMod::finishedSim = false;

omp_set_num_threads(2); //Set num threads to 2
#pragma omp parallel sections default(shared)
{
#pragma omp section
{
printf("*** Communitacion thread created.\n");
//Create a communication thread
PENELOPE_MPI::mpi_balance();
printf("*** Communitacion thread finished.\n");
}
```

```
#pragma omp section
{

#endif
#endif
#endif


ctrsimMod::nhist = 0.0;
while(!Bucle_History)       // Each iteration simulates a new
    history
{
.
.
.
```

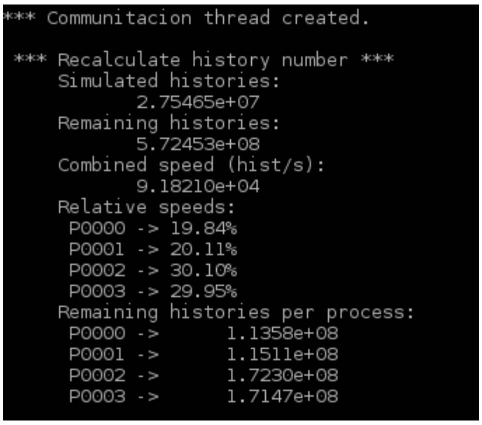
on la funció *PENELOPE_MPI* :: *mpi_balance*() la podem trobar a l'apèndix B.Depenent del procés, el fil de comunicacions actuarà de distinta forma:

- El procés 0 esperarà el temps especificat per l'usuari i aleshores enviarà, mitjançant la directiva *MPI_Bcast*, una petició d'actualitzar l'assignació d'històries. Una vegada enviada, esperarà a que tots els processos envien el nombre d'històries simulades i el temps transcorregut des de l'última actualització. Al rebre la informació de tots els processos, calcularà la velocitat relativa a la suma de les velocitats de totes les simulacions. Per a cada procés, a partir d'aquesta, recalcula el nombre d'històries restants que ha de simular. Tot seguit envia aquesta informació a cada procés i actualitza el seu propi límit d'històries a simular.

  Aquest procés es repeteix fins que acaba de simular totes les històries, realitza el màxim nombre d'iteracions de comunicació o supera el percentatge d'històries predefinit per l'usuari. En aquest punt enviarà un senyal per a finalitzar les comunicacions i acabarà l'execució del fil comunicador.

- Els processos amb "rank" distints de 0 esperaran el mateix temps definit per l'usuari i quedaran bloquejats en la directiva *MPI_Bcast* fins que el procés 0 envie la petició corresponent. Si la petició que arriba és d'actualització de l'assignació d'històries enviarà al procés 0 la informació mencionada al punt anterior. A continuació esperarà la resposta per a, finalment, actualitzar el seu límit d'històries a simular. D'altra banda, si la petició és de finalització de les comunicacions, acabarà el bucle de comunicacions i l'execució del fil pertinent.

Si algun dels processos acaba de simular i les comunicacions continuen activades, enviarà una velocitat de 0 històries/segon. Amb aquest mètode evitarà que se li assignen més històries. A la figura 15 vegem un exemple de l'eixida mostrada pel fil comunicador.



```
*** Communitacion thread created.

 *** Recalculate history number ***
     Simulated histories:
            2.75465e+07
     Remaining histories:
            5.72453e+08
     Combined speed (hist/s):
            9.18210e+04
     Relative speeds:
      P0000 -> 19.84%
      P0001 -> 20.11%
      P0002 -> 30.10%
      P0003 -> 29.95%
     Remaining histories per process:
      P0000 ->      1.1358e+08
      P0001 ->      1.1511e+08
      P0002 ->      1.7230e+08
      P0003 ->      1.7147e+08
```

Figura 15: Eixida del fil de comunicacions durant la redistribució de la càrrega.

4.3 Paral·lelització del post-processat

La primera etapa del post-processat consisteix en recollir totes les dades dels diferent processos. Hem vist a la secció anterior que s'empra la directiva MPI_Reduce per a aquest propòsit, reduint així el cost a ordre $O(log_2(Np))$.

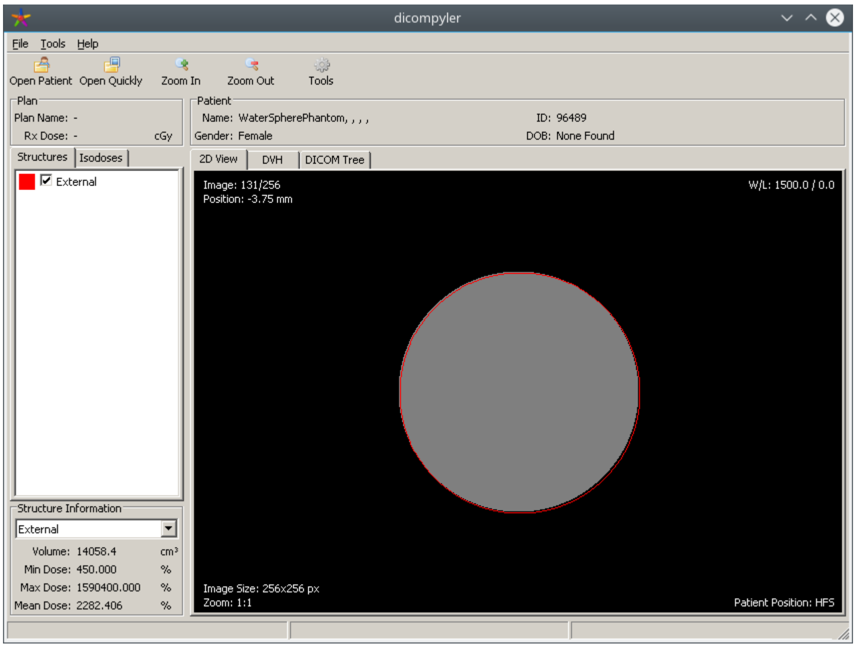
En segon lloc, una vegada el resultat final es trobe al procés amb "rank" 0, es paral·lelitza el processat de cada pla del DICOM. Per a fer-ho, primer emprem la directiva *#pragma omp parallel*, on creem els comptadors parcials per a cada fil en execució i es creen variables privades per a valors que seran accedits contínuament durant el post-processat dels plans. A continuació, mitjançant la directiva *#pragma omp* for, es reparteixen els plans en l'eix Z a processar entre tots els fils en execució. Finalment, es sumen els comptadors parcials de tots els fils per a obtindre el resultat final. Ho fem mitjançant la directiva *#pragma omp atomic* per a garantir que l'escriptura i lectura als comptadors globals es realitzen de forma correcta. Els details de l'implementació es poden veure a l'apèndix D.

5 Tests

A aquesta secció es mostraran els tests realitzats per a comprovar si és viable una futura aplicació en l'àmbit clínic. En primer lloc, es prendrà un DICOM artificial que consisteix en una única esfera d'aigua dins d'una caixa d'aire. A més, conté un contorn que envolta l'esfera i una única llavor situada al centre d'aquesta. En un segon test, estudiarem un tractament clínic real de braquiteràpia de pròstata.

5.1 DICOM artificial

Com podem veure a la figura 16, aquest DICOM està format per una esfera d'aigua dins d'un bloc d'aire. Les dimensions, en vòxels, del DICOM són $256x256x256$ pel que és relativament menut.



Plan Name: -
Rx Dose: - cGy
Name: WaterSpherePhantom, , , ,
ID: 96489
Gender: Female
DOB: None Found
Image: 131/256
Position: -3.75 mm
W/L: 1500.0 / 0.0
Image Size: 256x256 px
Zoom: 1:1
Patient Position: HFS
External
Volume: 14058.4 cm³
Min Dose: 450.000 %
Max Dose: 1590400.000 %
Mean Dose: 2282.406 %

Figura 16: DICOM artificial visualitzat amb el visor "dicompyler" [4].

Realitzarem dos tipus de simulacions amb aquest DICOM. Al primer cas, simularem la geometria de la llavor vista a la figura 5. Després, realitzarem la mateixa simulació emprant l'espai de fase que crearem per a dita llavor. D'aquesta forma compararem la velocitat d'ambdós tipus de mètodes i elegirem el més adequat per al cas clínic.

5.1.1 Creació de l'espai de fases

La geometria de la llavor consta d'una càpsula de titani de $4.5mm$ d'altura i $0.05mm$ de gruix. Conté un cilindre de plata de $3.0mm$ de llargària on es deposita l'isòtop I^{125} (és una combinació de $AgBr$ i AgI amb un ràtio molecular de 2.5 : 1). Els detalls de la font es poden trobar a [19] i la vegem a la figura 5. L'espectre de decaïment del I^{125} s'ha extret de la base de dades de la IAEA [13].

En primer lloc simulem la llavor envoltada per un material absorbent perfecte per a crear un espai de fases que emprarem en totes les simulacions posteriors. Per tal d'obtindre suficient estadística per a satisfer l'error relatiu recomanat al TG-186 [23] (5%) es simulen 10^9 partícules per a generar l'espai de fases, el qual ocupa un volum de $58Gb$.

Finalment emprarem aquest espai de fases per a realitzar la simulació del tractament clínic. Un dels problemes en aquest tipus de simulacions on el volum dels vòxels és relativament menut ($1.74418{\cdot}10^{-02}x1.72413{\cdot}10^{-02}x1.00000{\cdot}10^{-01}\ cm$) és que la disminució de la incertesa és lenta. Açò es deu a que, per a reduir la incertesa estadística d'un vòxel, un gran nombre de partícules han de depositar energia en aquest, ja que, com hem vist a la secció 2.1.5, aquesta disminueix amb $\sqrt{N}$, sent N el nombre d'històries que depositen energia a aquest vòxel.

Per a solucionar aquest problema existeixen dues vies. La primera opció és simular més partícules, el que involucra un espai de fases molt més gran. Una segona opció, i la que elegim en aquest tractament, és la d'emprar tècniques de reducció de variància, més específicament el "spliting" de partícules. Aquesta consisteix en simular cada partícula inicial ns vegades disminuint el pes estadístic de cada partícula clonada. Cadascuna tindrà un pes estadístic $1/ns$. Tot i que inicialment les partícules són idèntiques, cada clon seguirà una traça aleatòria i distinta de la resta. Com a conseqüència, cada història depositarà energia en un major nombre de vòxels, disminuint així la incertesa global. Aquest mètode no provoca biaix a les dades sempre que es tracten correctament els pesos estadístics.

S'ha elegit un factor de "spliting" de 10 (cada partícula inicial es clona 10 vegades), amb el qual arribem a la incertesa relativa requerida. Augmentar aquest factor incrementa el cost computacional de la simulació, per tant s'ha de trobar el compromís entre reducció d'incertesa i temps de còmput.

5.1.2 Elecció dels paràmetres

Per a elegir els paràmetres de les simulacions, empraren el mètode anàleg al que veurem a la secció següent en l'exemple de cas clínic. Donat que el mètode és equivalent, aquest apartat s'explicarà en el cas clínic, ja que, al ser un cas d'estudi real, resulta més interessant.

5.1.3 Resultats en memòria compartida

A aquest apartat, estudiarem les simulacions amb i sense emprar espai de fases en memòria compartida. El maquinari emprat en aquests tests és un processador Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz, el qual disposa de 4 cors i 8 "threads". Les característiques completes del processador les podem trobar a [1]. En primer lloc vegem el "speed up" per a la simulació sense espai de fase en funció del nombre de processos MPI. La velocitat amb un únic procés és de $4.06144 \cdot 10^3\, hist/s$.

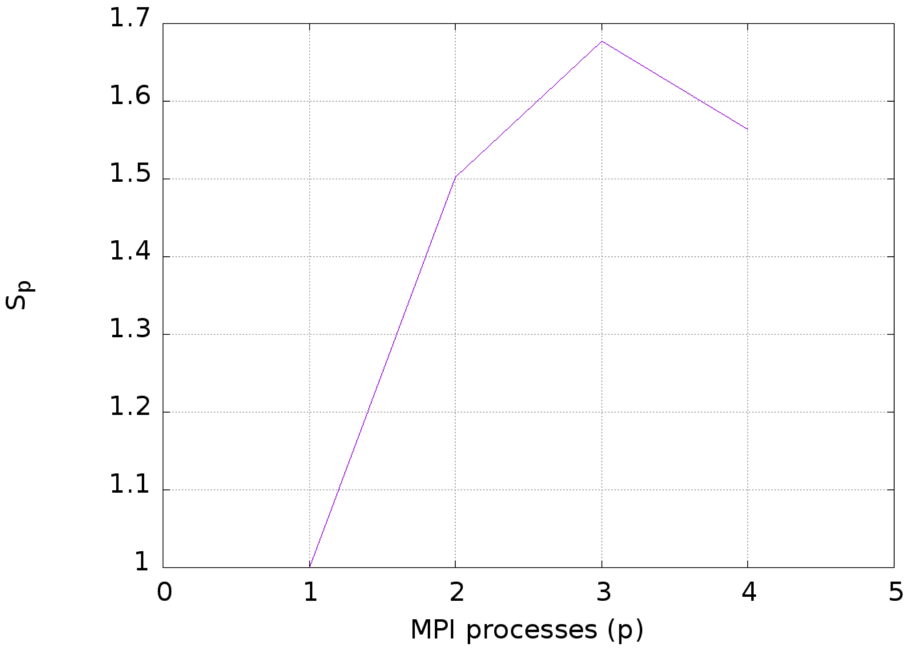


Figura 17: A l'eix Y es representa $\frac{T_1}{T_p}$ on T_1 és el temps invertit en realitzar la simulació amb un únic procés i T_p el temps necessari per a acabar la simulació emprant p processos. A l'eix X s'indica el nombre de processos. No s'inclou el temps d'inicialització ni post-processat. Dades corresponents a la simulació sense espai de fases.

A la figura 18 mostrem el temps que tarda el procés amb rank 0 en obtindre els resultats sumats de tots els processos des de que acaba la simulació. Vegem a la gràfica que el temps d'espera per a que acaben les simulacions de tots els processos és comparable al temps d'unir les dades. No obstant, aquest temps és inferior a un segon i, per tant, és menyspreable.

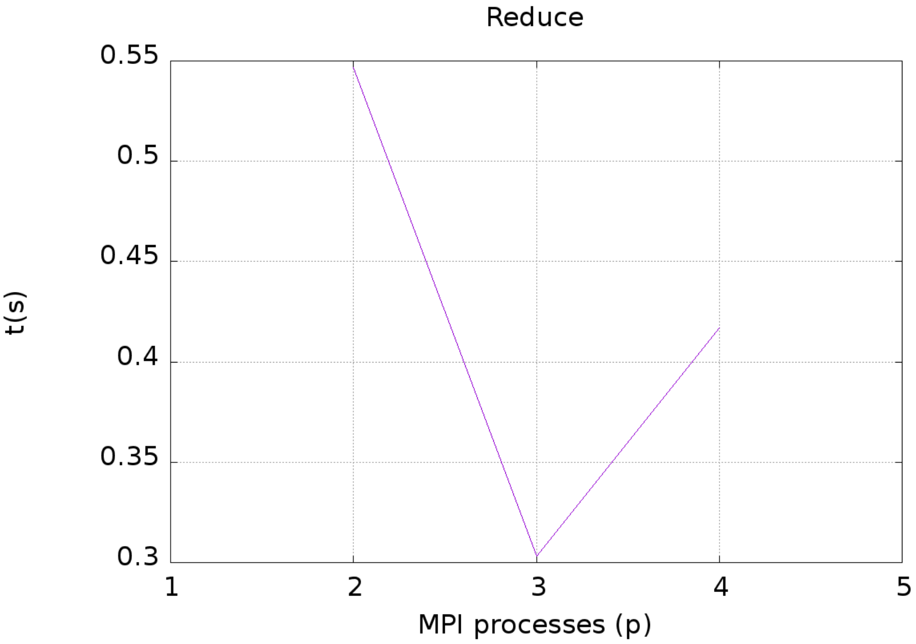


Figura 18: Temps transcorregut en sincronitzar els processos i sumar les dades dels diferent p processos. Dades corresponents a la simulació sense espai de fases.

Anàlogament obtenim els resultats per a la simulació emprant espai de fases. La velocitat, en històries per segon, d'un únic procés és de $7.43368 \cdot 10^5\, hist/s$.

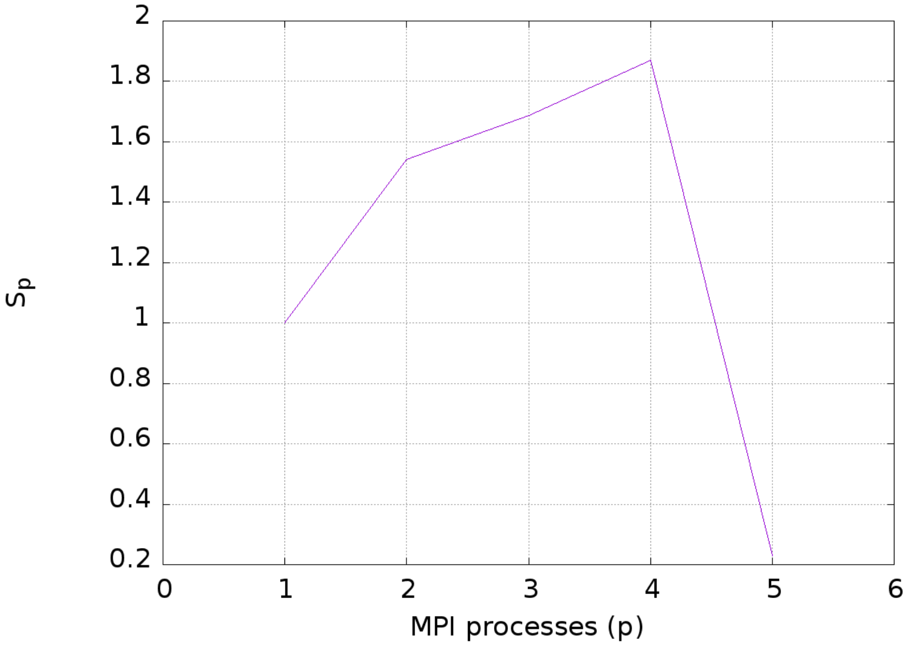


Figura 19: A l'eix Y es representa $\frac{T_1}{T_p}$ on T_1 és el temps invertit en realitzar la simulació amb un únic procés i T_p el temps necessari per a acabar la simulació emprant p processos. A l'eix X s'indica el nombre de processos. No s'inclou el temps d'inicialització ni post-processat. Dades corresponents a la simulació amb espai de fases.

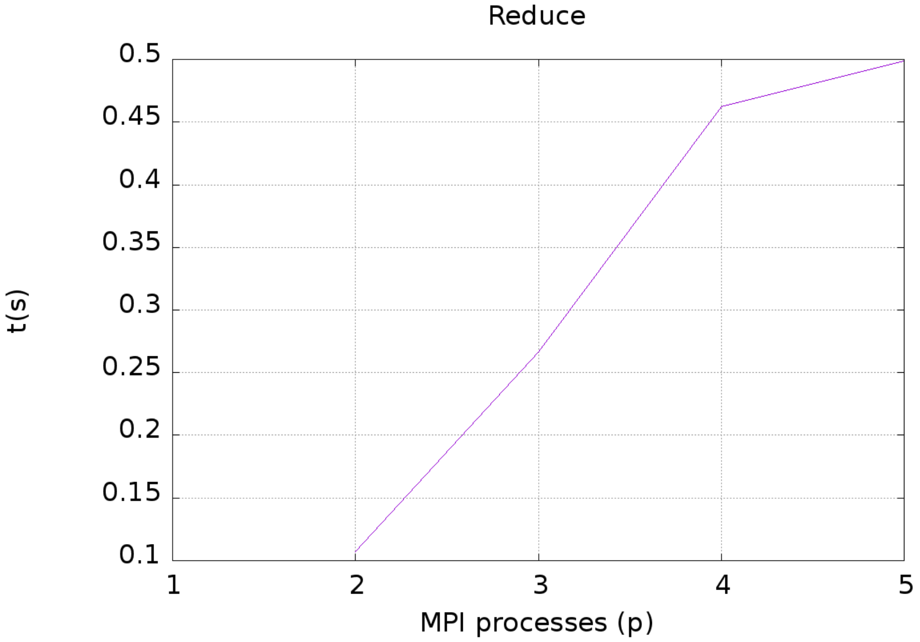


Figura 20: Temps transcorregut en sincronitzar els processos i sumar les dades dels diferent p processos. Dades corresponents a la simulació amb espai de fases.

Pel que respecta al temps d'inicialització, en tots els casos està entre $40 - 50\,s$ pel que tampoc te major rellevància. A la gràfica següent vegem el quocient de velocitats de simular amb i sense espai de fases, v_w i v_{wo} respectivament.

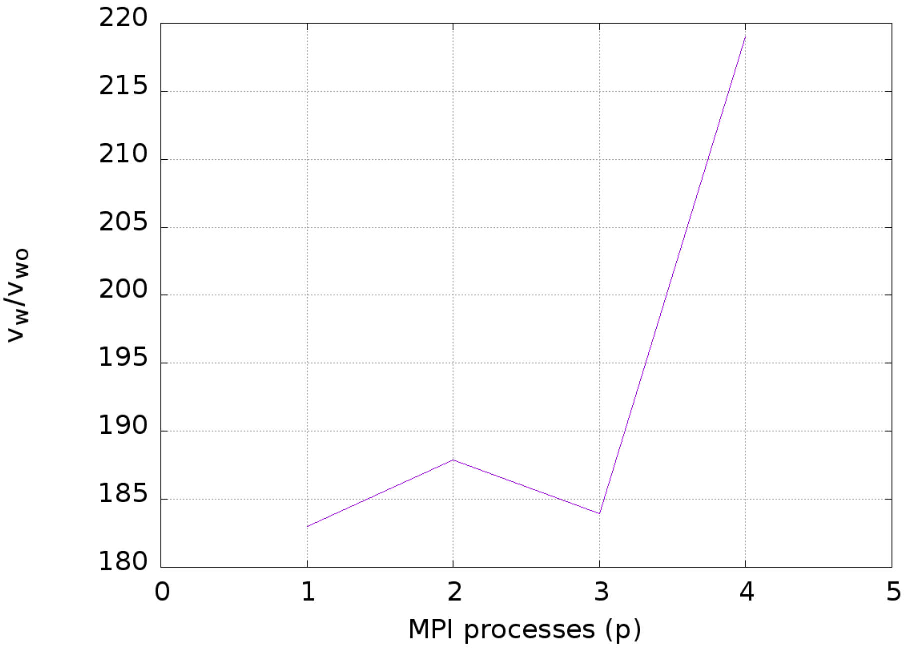


Figura 21: Quocient entre les velocitats de simulació amb espai de fase (v_w) i sense (v_{wo}).

Vegem clarament que emprar espais de fase incrementa molt la velocitat de simulació, per aquest motiu els utilitzarem al cas clínic. A part, hem vist que no s'aconsegueix un "speed up" pròxim al nombre de CPUs a un mateix node. Més avant comprovarem l'escalabilitat en memòria distribuïda per a determinar si és més convenient emprar un gran nombre de nodes en memòria distribuïda o processadors més potents amb menys nodes.

5.1.4 Cost teòric de suma dels resultats

A la secció següent realitzarem proves amb 4 nodes en memòria distribuïda. No obstant, donat que no tenim accés a un gran nombre de nodes, realitzarem un càlcul teòric del cost de sumar els resultats en el cas d'emprar un "cluster" amb memòria distribuïda. Durant la simulació no hi ha comunicació entre els processos o és menyspreable (si emprem el distribuïdor de càrrega). Per tant, els temps de comunicació sols afectaran a la inicialització i post-processat.

Pel que fa a la inicialització, si el que es vol és simular tractaments i estudis clínics, els fitxers de materials, les geometries de les llavors radioactives, espais de fase i demés fitxers de configuració seran comuns per a molts tractaments. Per tant, en una versió futura optimitzada per a aquests tipus de tractaments, aquestes dades estaran precarregades al programa o, en el cas d'espai de fase, dividit amb anterioritat. Així es redueix les comunicacions durant la inicialització a únicament l'enviament del DICOM processat, ja que, aquest sí canviarà en cada tractament i pacient.

D'altra banda, si les simulacions realitzades són amb finalitat investigadora, el temps de simulació serà, en la majoria dels cassos, ordres de magnitud superiors a la inicialització, fins al punt de ser el temps invertit en aquesta menyspreable.

Per tant, ens centrarem en el cost del post-processat, ja que aquest serà inevitable en qualsevol tipus de simulació.

Per simplicitat, suposarem que el nombre de nodes és potència de 2. Mostrem a continuació un exemple d'un algoritme simple de "reduce" per a sumar la dosi depositada de tots els vòxels en les simulacions dels p processos,

```
$s = RecDoub(a, nvox, p, p_r)$
|  $per\ a\ i = 1, 2, ..., \log_2(p)$
|      $si\ p_r\ MOD\ 2^i = 0$
|          $per\ a\ j = 1, 2, ..., nvox$
|              $a(j)_{p_r} = a(j)_{p_r} + a(j)_{p_r + 2^{i-1}}$
|          $fi\ per\ a$
|      $fi\ si$
|  $fi\ per\ a$
|  $s = a_0$
```

on a és la matriu a sumar, $nvox$ el nombre d'elements de la matriu a i p_r l'identificador de procés. Donat que hem de sumar dues matrius amb $nvox$ elements i precisió doble, una amb la dosi depositada i un altra amb la suma dels quadrats de la dosi depositada per cada història a cada vòxel, el cost computacional (flops) serà,

$$\text{Cost} = 2 \cdot 2(1 + nvox)\log_2(p) + 1 \approx 4 \cdot nvox \cdot \log_2(p),$$

on hem emprat que $nvox >> 1$. D'altra banda, suposant una aproximació lineal on τ és el temps necessari per a enviar un "double" i β és el temps de latència, el cost en comunicacions serà,

$$\text{Cost comunicació} = 2(nvox \cdot \tau + \beta)\log_2(p) \approx 2 \cdot nvox \cdot \tau \cdot \log_2(p),$$

on suposem que $nvox \cdot \tau >> \beta$ i que tots els nodes estan intercomunicats entre ells. Si agafem com exemple el nombre de vòxels del cas clínic que anem a emprar, la imatge d'aquest està formada per 31367232 vòxels. Per tant, el cost total de comunicacions és,

$$\text{Cost comunicació} \approx 62734464\tau\log_2(p).$$

A la pàgina [14] podem veure que ofereixen interconnexions "Ethernet" de 10, 25, 40, 50 i 100 Gb/s. Assumint que la dimensió d'un "double" és 64 bits, el valor de τ per a aquestes connexions serà de $6.4 \cdot 10^{-9}$, $2.56 \cdot 10^{-9}$, $1.6 \cdot 10^{-9}$, $1.28 \cdot 10^{-9}$ i $6.4 \cdot 10^{-10}$ respectivament. A la figura 22 vegem una estimació dels temps de comunicació del post-processat,

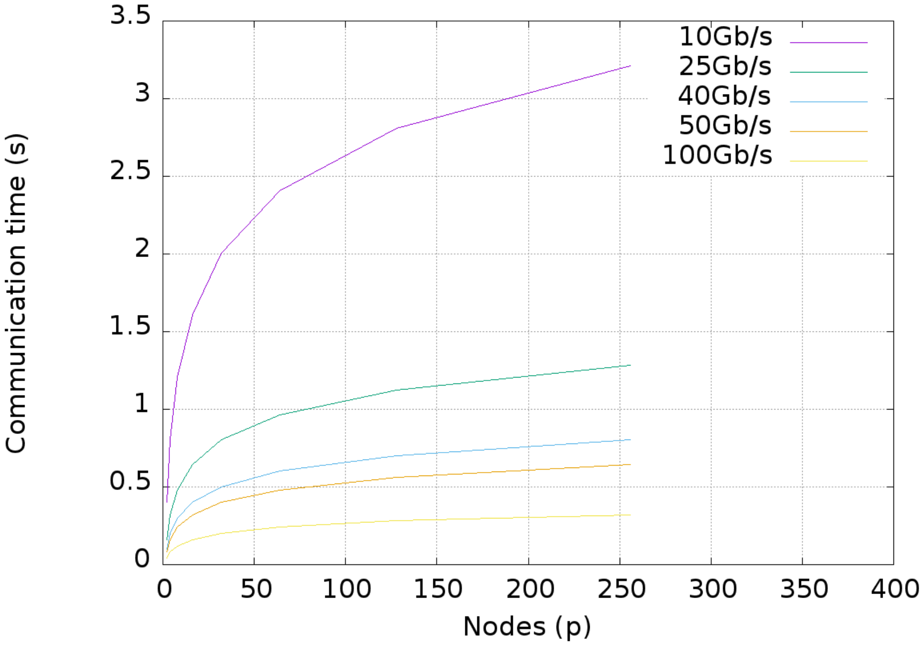


Figura 22: Estimació del temps de comunicacions durant el post-processat per al cas clínic d'exemple depenent de la velocitat d'interconnexió.

Al cas més desfavorable, vegem que l'ordre de temps és de pocs segons, el que és menyspreable front al temps de simulació. El temps de còmput de la suma de les matrius dependrà del tipus de processador, no obstant, ja hem vist a la figura 20 que, fins a 5 processos, aquest temps és menyspreable inclús incloent el temps transcorregut entre la finalització de la simulació del node 0 i la resta. Com que el cost escala logarítmicament amb el nombre de processos, podem extrapolar que aquest serà menyspreable fins a emprar, com a mínim desenes de nodes.

Cal tenir en compte que s'ha emprat una modelització simple de les comunicacions. A més, hem suposat que tots els nodes estan interconnectats, el que, en general, no serà cert. Caldria fer un estudi rigorós amb un gran nombre de nodes. Lamentablement no disposem d'aquesta infraestructura.

5.1.5 Escalabilitat en memòria distribuïda

Per a mesurar l'escalabilitat en memòria distribuïda, hem emprat la màquina "quadcluster" de la UPV, la qual consta de 4 nodes. Donades les limitacions de temps d'execució i d'espai en disc, no hem utilitzat espais de fases per a realitzar les proves, sinó que hem simulat la geometria de la font completa. Per a mesurar l'escalabilitat, s'han simulat $2 \cdot 10^6$ històries per cada node emprat. Els resultats els vegem a la gràfica següent,

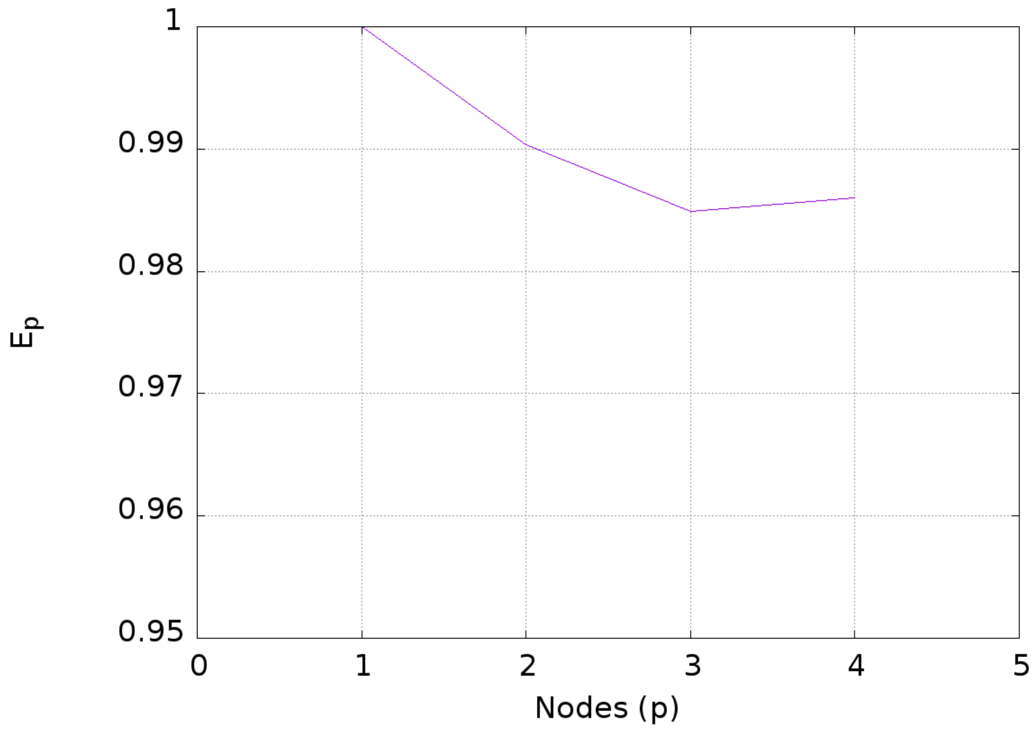


Figura 23: Escalabilitat (E_p) en funció del nombre de nodes (p) en memòria distribuïda.

On vegem que les fluctuacions en la velocitat de simulació són de l'ordre del temps de suma dels resultats dels diferents nodes. A continuació mostrem els temps transcorreguts des de que el procés amb “rank” 0 acaba la simulació fins que conté els resultats sumats de tots els nodes,

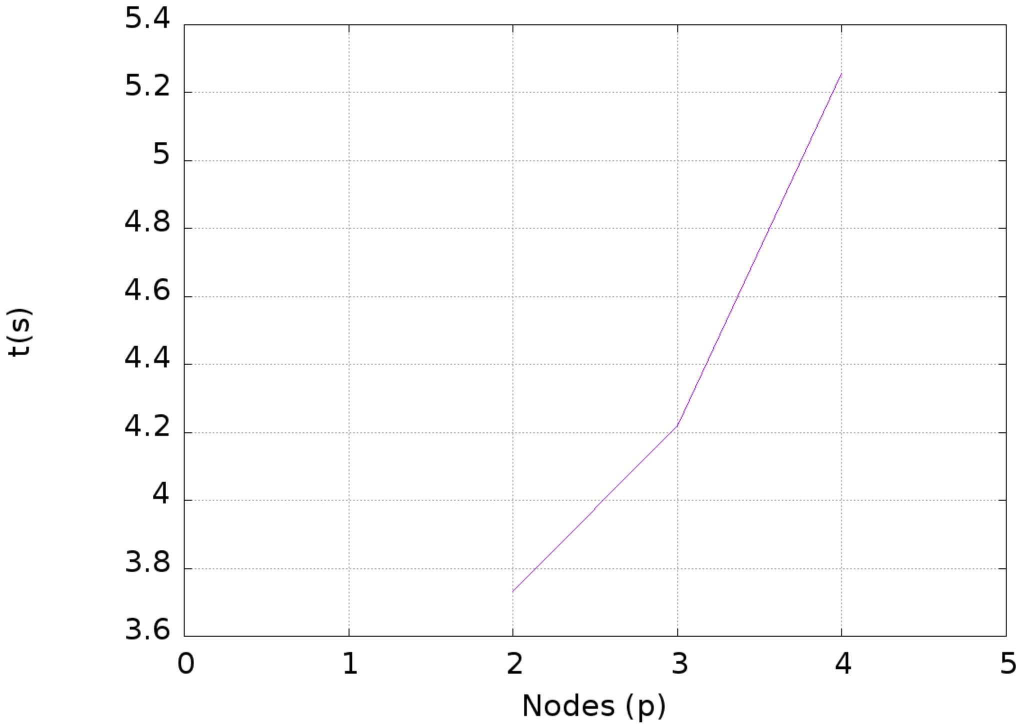


Figura 24: Temps invertit en reunir els resultats en funció del nombre de nodes (p) en memòria distribuïda.

Vegem per tant que el programa escala pràcticament linealment per a un nombre de nodes reduït. A més, els temps de reunir les dades més els desfasaments entre nodes (si emprem el balanceig de càrrega, com veurem més avant) són menyspreables. Tenint en compte el càlcul de la figura 22, aquesta escalabilitat es mantindrà per a desenes de nodes.

Concloem, que s’obté un major rendiment emprant un gran nombre de nodes poc potents intercomunicats entre ells, que amb menys nodes amb més CPUs.

5.2 Exemple de cas clínic

Per a testejar la viabilitat del codi final, hem elegit un cas real de tractament de càncer de pròstata amb braquiterapia proveït per l’Hospital Universitari i Politècnic La Fe. El tractament consta de 71 llavors radioactives del model 6711-OncoSeed introduïdes a través de 15 catèters. En aquest pla, la pròstata és l'objectiu del tractament clínic, el recte i uretra són els òrgans de risc.

5.2.1 Elecció dels paràmetres de simulació

En la configuració de la simulació sobre la imatge d'US simularem $6 \cdot 10^8$ històries de l'espai de fases produït en la secció anterior. Aquest valor, junt a un factor de "spliting" de 10 en l'espai de fases, permet arribar a un error relatiu global d'entre el 4 i 4.5%, suficient segons les recomanacions del TG-186.

El DICOM del tractament conté 3 contorns: el "target" corresponent a la pròstata, la "urethra" que rodeja la uretra i, finalment el "rectum" corresponent al recte. A més, conté 71 focus d'emissió de partícules, corresponent a les 71 llavors emprades al tractament.

En quant als materials emprats, utilitzarem la composició especificada en la figura 7 per a la pròstata i per al teixit extern als contorns ("Mean male soft tissue"). Per a la uretra s'ha emprat el material "urea" modelitzat en les taules de materials del PENELOPE [22]. Finalment, donat que no coneguem una modelització senzilla per al recte i aquesta dependria fortament de la dieta del pacient, emprarem aigua en aquest contorn. En cada material s'han d'especificar les energies d'absorció per a cada tipus de partícula i els paràmetres emprats per a agrupar les col·lisions "soft" dels electrons i positrons.

En primer lloc, l'energia d'absorció dels positrons és irrellevant, ja que l'única forma de produir-se és a partir de la desintegració d'un fotó amb una energia de, com a mínim, dues vegades la massa de l'electró (o positró). Si acompleix aquesta condició, el fotó pot interaccionar amb un àtom del material i desintegrar-se produint un parell electró-positró. Donat que la massa de l'electró és aproximadament $511keV$ i els fotons que es van a simular no superen els $36keV$ aquesta producció mai tindrà lloc.

Estudiem ara el cas dels electrons. Mostrem a continuació els rangs que recorren els electrons en els materials emprats en la simulació a partir de les taules del NIST [16].

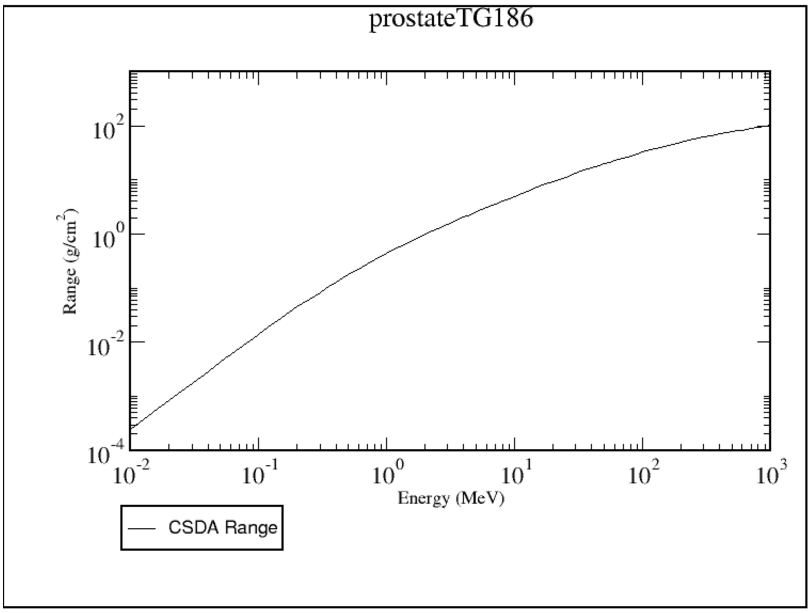


Figura 25: Rang dels electrons en pròstata extret del NIST [16].

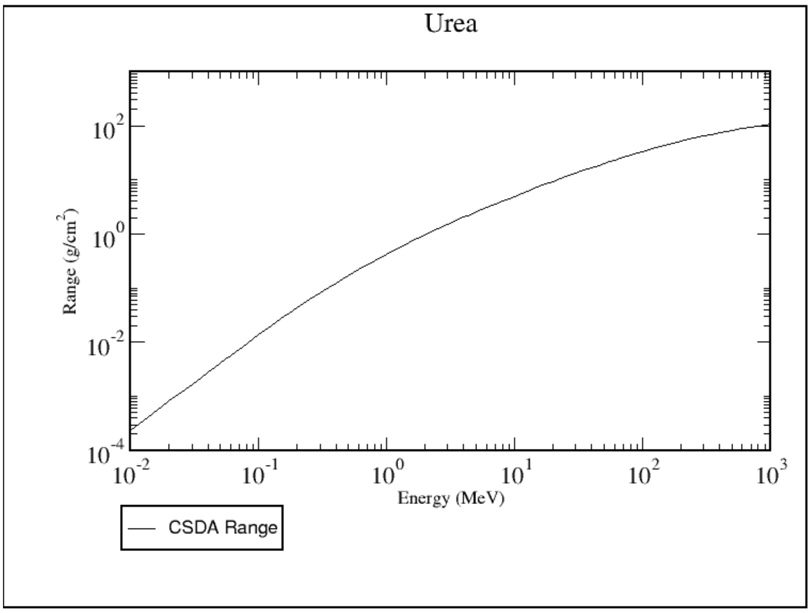


Figura 26: Rang dels electrons en urea extret del NIST [16].

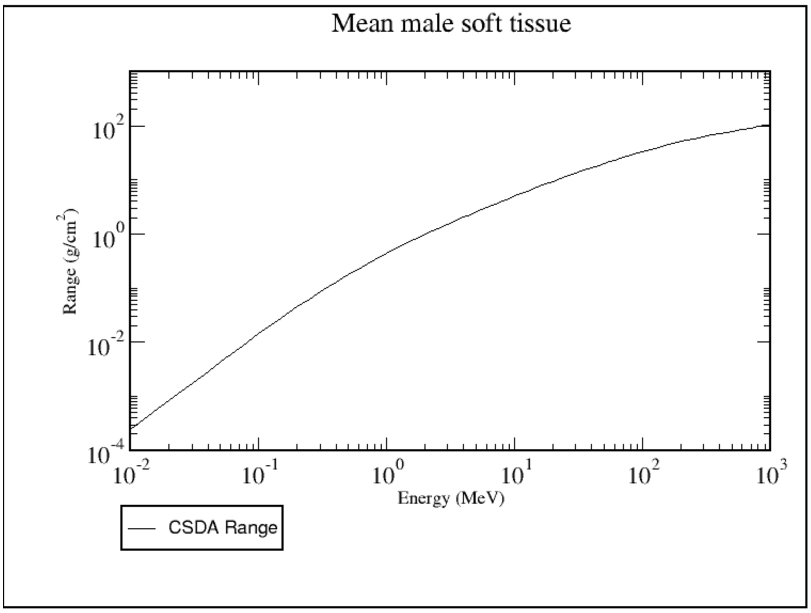


Figura 27: Rang dels electrons al material “Mean male soft tissue” extret del NIST [16].

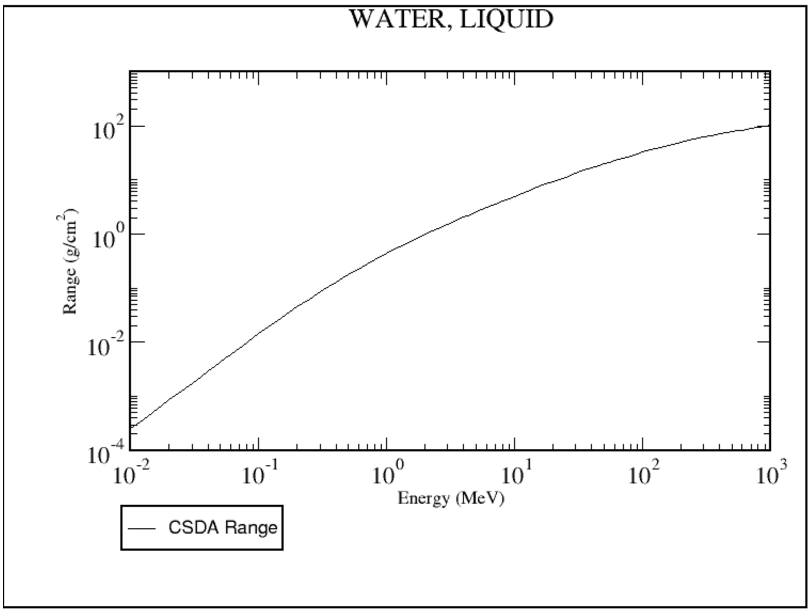


Figura 28: Rang dels electrons en aigua extret del NIST [16].

Com vegem, els rangs a tots els materials són molt semblants i són de l'ordre de $10^{-3} cm$. D'altra banda, la dimensió de cada vòxel és $1.74418 \cdot 10^{-02} x 1.72413 \cdot 10^{-02} x 1.00000 \cdot 10^{-01}\ cm$. Per tant, el rang dels electrons en aquests materials és unes 10 vegades menor que la dimensió del vòxel. Aquest fet ens permet absorbir tots els electrons, una vegada són produïts en les interaccions, especificant una energia d'absorció major que l'energia de qualsevol partícula inicial. Açò provocarà que l'electró siga absorbit al vòxel on es produeix i, per tant, la seva energia es depositarà en aquest. Simular el transport de l'electró únicament augmentaria el cost computacional sense canviar el resultat final, ja que aquests rarament recorreran més d'un vòxel.

Finalment, estudiem l'energia d'absorció dels fotons. Per a decidir aquest paràmetre emprem la llei d'atenuació exponencial (equació 22) i els coeficients d'absorció tabulats al NIST [16]. Fixarem l'energia d'absorció dels fotons a $2keV$, on la probabilitat de que un fotó travesse la distància mínima entre dos parets oposades d'un mateix vòxel sense ser absorbit és del $4.6798 \cdot 10^{-03}\%$.

Pel que fa als paràmetres del tractament, la dosi receptada pels metges és de $160Gy$, el que significa que, idealment, tots els vòxels dins del contorn de la pròstata deurien rebre aquesta dosi. L'activitat aparent (la que es mesura a partir del nombre de desintegracions incloent la càpsula) és de $20.934 \cdot 10^6\ Bq$ (desintegracions/s), i la seva semivida és de 59.4 dies.

5.2.2 "Speed up" de la simulació en memòria compartida

Donat que ja hem vist que l'escalabilitat del codi és pràcticament lineal, farem una prova en memòria compartida. Estudiarem quants processos llançar per node emprant el mateix maquinari. Realitzem la simulació per a diferent nombre de processos a un mateix node i representem l'increment de velocitat. La velocitat de la simulació amb un procés és de $5.37360 \cdot 10^4$ histories/s.

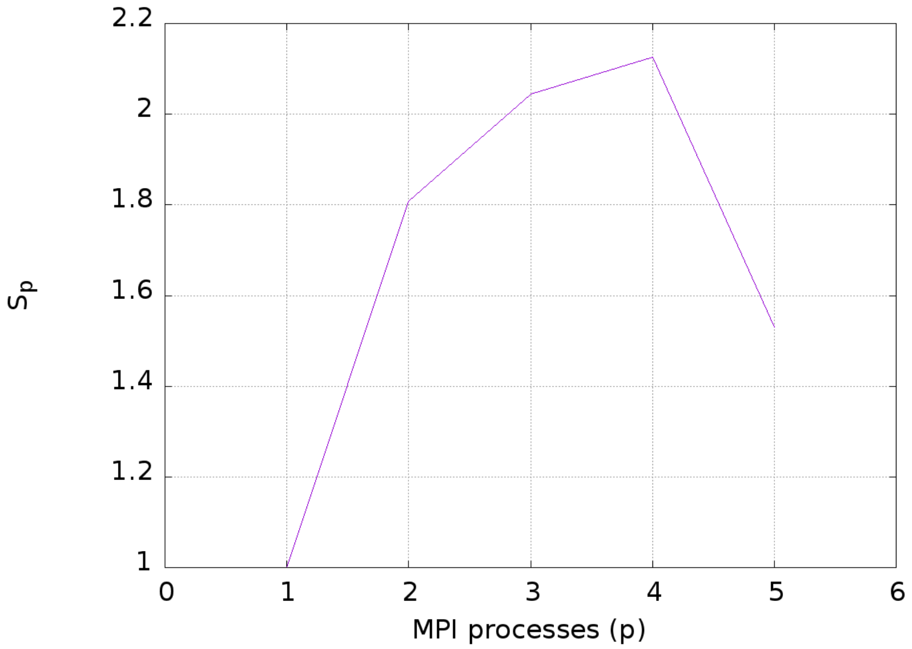


Figura 29: A l'eix Y es representa $\frac{T_1}{T_p}$ on T_1 és el temps invertit en realitzar la simulació amb un únic procés i T_p el temps necessari per a acabar la simulació emprant p processos. A l'eix X s'indica el nombre de processos. No s'inclou el temps d'inicialització ni post-processat. Dades corresponents a la simulació del cas clínic amb espai de fases.

Al cas més favorable, 4 processos, el temps invertit en la simulació és de $5.25141 \cdot 10^3\ s$. Per a que el temps de simulació baixe a l'ordre de pocs minuts, tenint en compte els resultats sobre memòria distribuïda de la secció anterior, caldrà emprar millors processadors i/o entre 10 i 20 nodes interconnectats, depenent de la velocitat requerida. Suposem que el temps de reunir els resultats finals de tots els nodes és menyspreable, tal i com hem vist als tests anteriors i estimat a la figura 22.

5.2.3 "Speed up" del post-processat

Una vegada reunits els resultats dels diferents processos, cal extraure les corbes d'isodosi i les gràfiques acumulatives de dosi. Per a aconseguir-ho, emprarem distints fils. No obstant, prèviament, per a aconseguir un major rendiment durant la simulació, es possible que s'haja lligat cada procés MPI a un "socket" i CPU determinades mitjançant directives del tipus "–map-by socket' i "–map-by core". Una vegada acaba la simulació i sumats els resultats dels diferents processos, cal eliminar aquesta restricció al "rank" 0 per a permetre emprar totes les CPUs durant el post-processat. La part del codi que s'encarrega d'aquesta tasca la vegem a continuació, en la qual hem emprat la llibreria *sched.h*,

---

```
.
.
.
//next tallys only must be processed by rank 0
//with no communications

#ifdef _PENELOPE_WITH_MPI_

if(mode == -1) //final report
  {

     int rank = 0;
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);

     //Calculate the number of simulated showers
     double total_nhist = 0.0;
     MPI_Reduce(&n   , &total_nhist,  1, MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD);
     n = total_nhist;

     MPI_Barrier(MPI_COMM_WORLD);
     MPI_Finalize();

     if(rank != 0){ return;} //Only proces 0 report next tallys

#ifdef _USE_OMP_

     //Unset bindigs to use all processes

     int nCPU = sysconf(_SC_NPROCESSORS_ONLN);
     printf("No. of processors : %d\n", nCPU);

     cpu_set_t cset;
     CPU_ZERO(&cset);
     sched_getaffinity(0, sizeof(cset), &cset);

     printf("Rank 0 binded to:\n");
     bool avaibleCores = false;
     for(int i = 0; i < nCPU; i++)
```

```
{
   //Check if CPU is member of set
   if(CPU_ISSET(i,&cset))
   {
      printf(" CPU %d\n",i);
   }
   else
   {
      //Add CPU to set
      CPU_SET(i,&cset);
      avaibleCores = true;
   }
}

if(avaibleCores)
{
   printf("Rebinding to all avaible CPUs for OMP post-
       processing\n");

   sched_setaffinity(0, sizeof(cset), &cset);

   CPU_ZERO(&cset);
   sched_getaffinity(0, sizeof(cset), &cset);

   printf("\nRank 0 rebinded to\n");
   for(int i = 0; i < nCPU; i++)
   {
      //Check if CPU is member of set
      if(CPU_ISSET(i,&cset))
      {
         printf(" CPU %d\n",i);
      }
      else
      {
         printf("  WARNING: Not binded to CPU %d\n", i);
      }
   }
}
else
{
   printf("All avaible cores are already binded.\n");
}
```

```
#endif

   }
#endif

IDCreport(mode,n,cputim,uncdone);
.
.
.
```

A la figura 30 vegem un exemple de l'eixida del codi anterior.

```
No. of processors : 8
Rank 0 binded to:
  CPU 0
  CPU 4
Rebinding to all avaible CPUs for OMP post-processing

Rank 0 rebinded to
  CPU 0
  CPU 1
  CPU 2
  CPU 3
  CPU 4
  CPU 5
  CPU 6
  CPU 7


 Post-processing IDCreport with 3 threads
```

Figura 30: Eixida de la reassignació de CPUs.

A continuació mostrem la gràfica amb l'increment de velocitat del post-processat depenent del nombre de "threads" emprats,

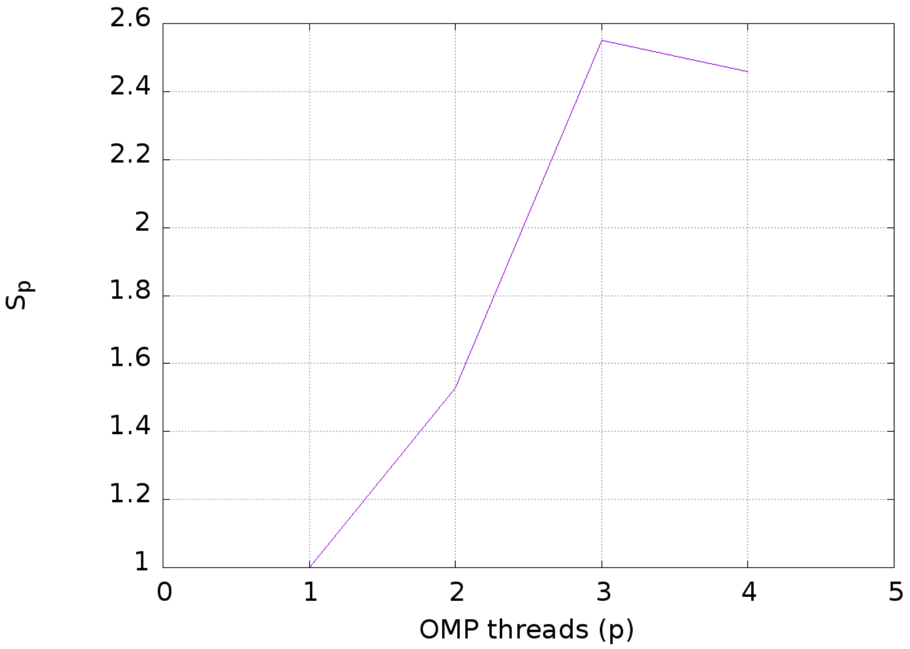


Figura 31: T_p/T_1 en funció del nombre de fils emprats durant el post-processat.

on el temps requerit per a un únic fil ha sigut de $2.14153 \cdot 10^2$ segons. Aquest temps, tot i que està un ordre de magnitud per baix del de simulació, pot arribar a ser el coll de botella si emprem suficients nodes per a reduir el temps de simulació o si tractem DICOMs de major dimensió.

5.2.4 Balanceig de la càrrega

Per a fer un test sobre la funcionalitat de balanceig de la càrrega, s'ha executat la instrucció "yes" sobre dos de les CPUs del processador. Les instruccions utilitzades han sigut les següents,

```
taskset 0x1 yes &> /dev/null &
taskset 0x2 yes &> /dev/null &
```

D'aquesta forma s'aconsegueix que dos de les CPUs siguen més lentes, simulant dos tipus de processadors distints. A continuació realitzarem una simulació amb 4 processos MPI. A la gràfica següent vegem les velocitats (històries/s) de cada procés,

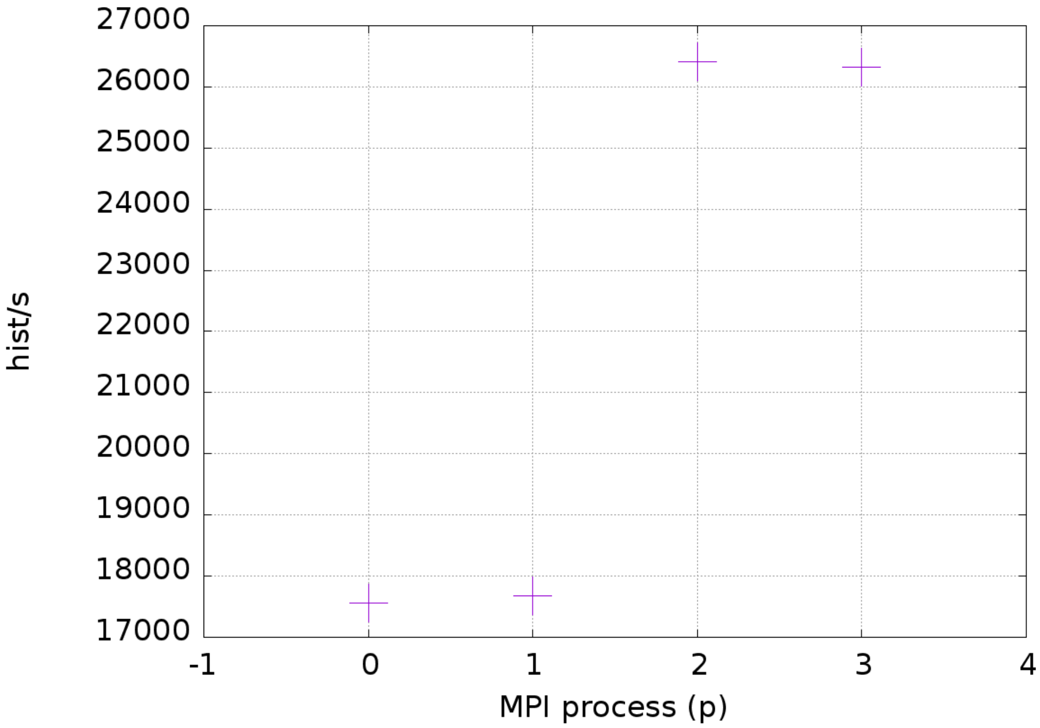


Figura 32: Velocitats, en històries per segon, de la simulació de cada procés MPI.

La simulació s'ha executat dues vegades, amb i sense balanceig de càrrega. A les gràfiques 33 i 34 vegem representat els temps invertits en les simulacions de cada procés així com el nombre d'històries simulades.

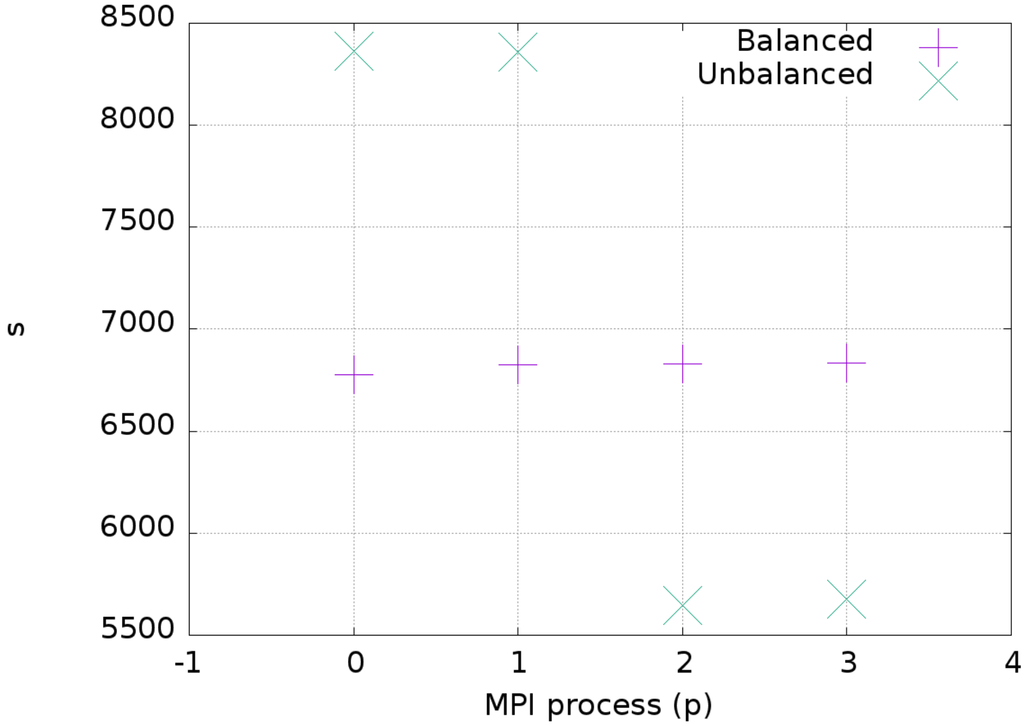


Figura 33: Temps de simulació per a cada procés MPI.

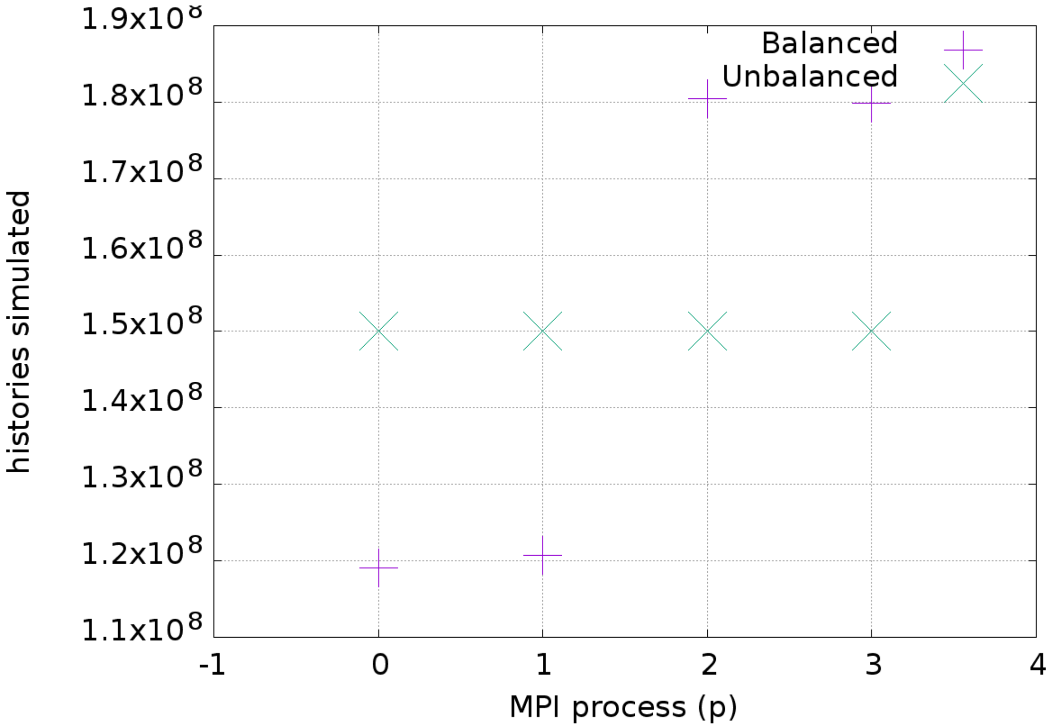


Figura 34: Històries simulades per cada procés MPI.

Com és obvi, el temps de la simulació global ve determinat per la velocitat més lenta. Les velocitats de simulació global amb i sense balanceig de càrrega són $8.85376 \cdot 10^4$ i $7.17551 \cdot 10^4$ històries/s respectivament. En aquest cas d'exemple el factor guanyat és un 23.39%, però aquest increment en la velocitat dependrà del nombre de processos i de les velocitats relatives entre ells.

6 Línies futures

El treball futur que queda pendent és:

- Fer un estudi experimental de l'escalabilitat en un major nombre de nodes en memòria distribuïda.
- Acabar de traduir i verificar els (pocs) "tallys" que queden en FORTRAN.
- Una vegada el codi complet estiga traduït i verificat, estudiar si és possible reestructurar les variables i el codi per a millorar els accessos a memòria.

- Estudiar la possibilitat d'emprar coprocessadors per a la paral·lelització de les simulacions.

- Emprar el codi en una major varietat de tractaments i ajustar els resultats calculats a les necessitats clíniques. A més, elegir uns paràmetres estàndard per a cada tipus de tractament de forma que es simplifique l'ús de les simulacions en aquests.

- Comprovar si es viable, tant en cost monetari com en temps de càlcul, emprar servidors "cloud" per a realitzar estudis clínics i evitar així tindre maquinari als hospitals. Per a açò, aprofitant la funcionalitat MPI, s'empraria la plataforma per a desplegar "clusters" en el "cloud" $EC3$ [6].

- Fer que el resultat final de la simulació es guarde en un fitxer amb format DICOM que guarde les distribucions de dosi del tractament. D'aquesta forma, podran ser visualitzats amb el visor de DICOMs que empren els corresponents metges o radiofísics.

- Optimitzar el post-processat per a que s'execute en més nodes i comprovar si es possible accelerar-lo respecte emprar sols memòria compartida.

7 Conclusions

Al llarg d'aquest treball s'ha aconseguit els següents objectius:

- Traduir i verificar la totalitat del nucli del PENELOPE de FORTRAN a C++ permetent realitzar simulacions de forma completament general. No obstant, falten alguns "tallys" per traduir i verificar.

- S'ha estudiat l'estàndard d'imatge mèdica DICOM i com extraure tota la informació necessària per a simular tractaments clínics de forma general. A més, s'ha automatitzat el procés de processat d'aquest format d'imatge per a escàners d'US i CT, típicament emprats en les planificacions de tractaments de radioteràpia.

- Hem conclòs que la paral·lització de simulacions de forma general mitjançant GPUs, a no ser que s'empren aproximacions que volem evitar, no és viable.

- A falta de més proves, concloem que, segons les proves realitzades i els càlculs per als temps de comunicacions, és possible aconseguir l'incertesa recomanada pel TG-186 amb un temps raonable realitzant simulacions precises en casos clínics. Per a açò necessitarem emprar un conjunt de nodes en memòria distribuïda on el nombre d'aquests dependrà de l'exigència de temps de cada tractament.

- Pel que fa a les aplicacions d'investigació, s'ha adaptat el PENELOPE per a ser emprat en sistemes distribuïts heterogenis, els quals representen la majoria dels que disposem en investigació. A més permet distribuir la càrrega automàticament, millorant així el rendiment en aquest tipus de recursos.

Tot i que s'ha fet un avanç important en aquest treball, encara falta un llarg recorregut per a aconseguir que les simulacions precises arriben a l'àmbit clínic. Aquest queda com a treball futur a realitzar.

A Apèndix: Funció llegir DICOM

```
void loadDicom(const char* dirName, int& nmatvox, double& mem)
{
  //Load and process dicom file

  const int nvoxmax=int(2.14748e9);     // ~max int*4
  const double eps=1.0E-10;
  const double minvoxSide=eps*1.0E4;


#ifdef _PENELOPE_WITH_MPI_

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if(rank == 0) //Only rank 0 read an processes the DICOM
    {

#endif

  DIR *auxDir;
  int i=0;
  char folderName[500];
  // check if is relative or absolute path
  if(dirName[0] != '/')
    {
      //relative path
      const char* pwd = getenv("PWD");
      strcpy(folderName, pwd);
      strcat(folderName,"/");
      strcat(folderName,dirName);
    }
  else
    {
      //absolute path
      strcpy(folderName, dirName);
    }

  //Init variables
  geovoxMod::nx = 0;
```

```
geovoxMod::ny = 0;
geovoxMod::nz = 0;

auxDir=opendir(folderName);
printf("\n");
if(auxDir==NULL) //check directory existence
  {
    printf("Error in 'loadDicom': folder not found (%s)\n",
       folderName);
    NERROR = 1000; exit(NERROR);
  }
printf("Reading folder '%s'\n", folderName);
//search for dicom files with extension '.dcm'

int totalFiles = 0;                               //Stores total
    files number. (not only dicoms)
while(readdir(auxDir)!=NULL)   //Count number of files
  {
    totalFiles++;
  }
closedir(auxDir);

// use dirent var to check files in the specified folder
dirent *data;
char AuxChar[500];                             //Stores filename of
     loaded dicom
char Dicomrtss[500];                           //Stores filename of
     dicom with contours data
char Dicomrtplan[500];                         //Stores filename of
     dicom with seeds data
int num_zPlanes=totalFiles;
double* ZPlanes = new double[num_zPlanes];        //Stores z
    position of each dicom file
int* nFilePixels = new int[num_zPlanes];          //Stores number
    of pixes in each file
char** filenamesDicom = new char*[totalFiles];   //Store image
    dicom filenames
char bottomDicomFilename[300];                   //Store bottom
    plane dicom filename
for(int k = 0; k < num_zPlanes; k++)
  {
    filenamesDicom[k] = new char[300];
  }
```

```
bool firstImageDicom = true;
double zmin = 1e20;                              //Stores minimum
    z origin found

int NumDicoms=0;            //Stores number of '.dcm' files in the
    folder


auxDir=opendir(folderName);
for(int k = 0; k < totalFiles; k++)      //Read one file data in
    each iteration and save filename
  {
    data = readdir(auxDir);
    char filename[200];
    sprintf(filename,"%s",data->d_name); //Save filename (d_name)
        in AuxChar and the number of chars
    if(filename[0] == '.')       //skip hide files (.filename)
 {
   continue;
 }

    //append path to filename
    strcpy(AuxChar,folderName);
    strcat(AuxChar,"/");
    strcat(AuxChar,filename);

    //store filename
    strcpy(filenamesDicom[k], AuxChar);

    dicom::dicomfile* dfaux;
    dfaux = dicom::open_dicomfile(AuxChar); //open dicom
    if(dfaux!=NULL)
 {
   //Read dicom modality
   dicom::dataelement* de;
   de = dfaux->get_dataelement(0x00080060);
   if(de -> is_valid()) //check modality dataelement
     {
      char dicomModality[500]; //store dicom image modality
      sprintf(dicomModality,"%s",(de -> to_string()).c_str());
      if(imageModality(dicomModality))
    {
     if(firstImageDicom)
```

```
  {
    //store dicom image modality
    firstImageDicom = false;
    strcpy(dicomPenelope::imageModality,dicomModality);
  }
else
  {
    if(strcmp(dicomPenelope::imageModality,dicomModality) !=
        0)
 {
   printf("loadDicom:warning: multiple image modality detected
       : %s %s\n", dicomPenelope::imageModality, dicomModality
       );
   printf("                 '%s' modality will be ignored.\n",
        dicomModality);
   delete dfaux;
   continue;
 }
  }
//Read Z position of the first plane of this dicom
de = dfaux->get_dataelement(0x00200032);
if(de -> is_valid()) //check element existence
  {
    double* AuxPos;
    int AuxNumComp;
    //Take image position
    de -> to_double_values_a(&AuxPos, &AuxNumComp);
    //Save it
    ZPlanes[NumDicoms] = AuxPos[2];

    if(ZPlanes[NumDicoms] < zmin) //Check first image dicom
 {
   zmin = ZPlanes[NumDicoms];
   strcpy(bottomDicomFilename,filenamesDicom[k]);
 }

    //read width, height, pixel number, number of frames and
        bytes per pixel
    int width, height;
    int precision, signedness, ncomponents, bytes_per_pixel;
    int nframes; //Planes in the image
    dfaux->get_image_info(&width, &height,
```

```
            &precision, &signedness, &ncomponents, &
                bytes_per_pixel,
            &nframes);

       nFilePixels[NumDicoms] = width*height*nframes;
       geovoxMod::nz += nframes;

       free(AuxPos);
       NumDicoms++; //Increment dicom number

       //Check vector size
       if(NumDicoms==num_zPlanes)
    {
      //needs larger array
      num_zPlanes+=200;
      redimArray(ZPlanes  , NumDicoms, num_zPlanes);
      redimArray(nFilePixels, NumDicoms, num_zPlanes);
    }

     }
}
   else if(strcmp(dicomModality,"RTSTRUCT") == 0)
{
  //This modality store contours data. Check the existence of
      element "ROI Contour Sequence"
  dicom::dataset* ds = dfaux -> get_dataelement(0x30060039)->
      dataset_at(0);
  if(ds!=NULL)
    {
      sprintf(Dicomrtss,"%s",AuxChar);
    }
}
   else if(strcmp(dicomModality,"RTPLAN") == 0)
{
  //This modality store seeds data. Check the existente of
      element "Application Setup Sequence"
  dicom::dataset* ds = dfaux -> get_dataelement(0x300a0230)->
      dataset_at(0);
  if(ds!=NULL)
    {
      sprintf(Dicomrtplan,"%s",AuxChar);
    }
}
```

```
    }
 }
   else
 {
  printf("%s is not a DICOM, skiping.\n", AuxChar);
 }
   delete dfaux;
  }

if(NumDicoms==0)
  {
   printf("No files with '.dcm' extension founded\n");
   NERROR = 1000; exit(NERROR);
  }

printf("\n************ %d DICOMs to load\n",NumDicoms);

//Try to read contours data dicom

dicom::dicomfile* dfrtss;
dfrtss = dicom::open_dicomfile(Dicomrtss);
//clean contours variables
int k=0;
while(k<dicomPenelope::nContours)
  {
   for(int n = 0; i < dicomPenelope::nContoursPlanes[k]; n++)
 {
  delete [] dicomPenelope::contoursPoints[k][n];
 }
   delete [] dicomPenelope::contoursPoints[k];
   delete [] dicomPenelope::nPlanePoints[k];
   dicomPenelope::nContoursPlanes[k]=0;
   dicomPenelope::nVoxContour[k]=0;
   k++;
  }
dicomPenelope::nContours=0;
if(dfrtss)
  {
   printf("\n*** Loading DICOMs contours\n");
   bool remainingContours=true; //Control if there are more
       contours to load
   int dicomContour = -1;
   while(remainingContours)
```

```
{
  dicomContour++;
  //Take contour number "dicomContour" of "Structure Set Roi
      Sequence"
  dicom::dataset* ds=dfrtss->get_dataelement(0x30060020)->
      dataset_at(dicomContour);
  if(ds!=NULL)
    {
      //Take contour name
      char AuxChar[200];
      sprintf(AuxChar,"%s",(ds->get_dataelement(0x30060026)->
          to_string()).c_str());

      stringToLow(AuxChar);
      //Find contour
      int ncontour = -1;
      for(int i = 0; i < dicomPenelope::nContoursInit; i++)
   {
     if(strcmp(AuxChar,dicomPenelope::contourName[i]) == 0)
       {
         //Contour found
         ncontour = i;
         break;
       }
   }

      if(ncontour == -1)
   {
     printf("loadDicom:warning: contour '%s' not specified in
         configuration file. Will be ignored.\n",AuxChar);
     continue;
   }

      //Read member number "dicomContour" of "Roi Contour Sequence
          "
      ds=dfrtss->get_dataelement(0x30060039)->dataset_at(
          dicomContour);
      if(ds!=NULL)
   {
     //read points of "Contour Sequence", ordered by planes.
     //first, count number of planes
     while(1)
       {
```

```
    dicom::dataset* ds2 = ds->get_dataelement(0x30060040)->
        dataset_at(dicomPenelope::nContoursPlanes[ncontour]);
    if(ds2!=NULL)
 {
   //Increment number of planes
   dicomPenelope::nContoursPlanes[ncontour]++;
 }
    else
 {
   //No more planes for this contour
   break;
 }
  }

//Create a array with the number of planes
if(dicomPenelope::nContoursPlanes[ncontour]>0)
  {
    dicomPenelope::contoursPoints[ncontour] = new double*[
        dicomPenelope::nContoursPlanes[ncontour]];
    dicomPenelope::nPlanePoints[ncontour] = new int[
        dicomPenelope::nContoursPlanes[ncontour]];
  }
int contPlanes=0;
while(contPlanes<dicomPenelope::nContoursPlanes[ncontour])
  {
    double* auxPoints;
    int AuxNumPoints=0;
    //read element number "contPlanes" of "Contour Sequence"
    dicom::dataset* ds2 = ds->get_dataelement(0x30060040)->
        dataset_at(contPlanes);
    if(ds2!=NULL)
 {
   //read contour points and convert to cm
   ds2->get_dataelement(0x30060050)->to_double_values_a(&
       auxPoints, &AuxNumPoints);
   int ContadorPasarcm=0;
   while(ContadorPasarcm<AuxNumPoints)
     {
       auxPoints[ContadorPasarcm]=auxPoints[ContadorPasarcm
           ]*0.1; //Convert to cm
       ContadorPasarcm++;
     }
   //create array and store points
```

```
        dicomPenelope::nPlanePoints[ncontour][contPlanes]=
            AuxNumPoints;
        dicomPenelope::contoursPoints[ncontour][contPlanes] = new
            double[AuxNumPoints];
        int contadorCopiarPunts=0;
        while(contadorCopiarPunts<AuxNumPoints)
          {
            dicomPenelope::contoursPoints[ncontour][contPlanes][
                contadorCopiarPunts]=auxPoints[contadorCopiarPunts
                ];
            contadorCopiarPunts++;
          }
        contPlanes++;
      }
         else
      {
        //if not exists, break bucle
        break;
      }
       }

     printf("Contour %s loaded\n",dicomPenelope::contourName[
         ncontour]);
     dicomPenelope::nContours++; //Augmentem el nombre de contorns
   }

    }
  else
    {
      remainingContours=false;
    }
 }
   //if the modality is "US", the number of materials will be
       number of contours+1
   if(strcmp(dicomPenelope::imageModality, "US") == 0)
 {
   dicomPenelope::NumMatsCal=dicomPenelope::nContours+1;
 }
  }
else
  {
    printf("\n*** Missing contours DICOM file (rtss.dcm)\n");
  }
```

```
delete dfrtss;

if(dicomPenelope::nContours != dicomPenelope::nContoursInit)
  {
    printf("loadDicom:Error: Only %d contours of the %d especified
        in configuration file found.\n", dicomPenelope::nContours,
        dicomPenelope::nContoursInit);
    NERROR=1000; exit(NERROR);
  }

//Save material number
nmatvox = dicomPenelope::NumMatsCal;

/////////////////////////////////
// Check dicom with seeds data //
/////////////////////////////////

//Clear var seeds
k=0;
while(k<dicomPenelope::numSeedTypes)
  {
    delete [] dicomPenelope::seedPos[k];
    dicomPenelope::seedsNumber[k]=0;
    k++;
  }
dicomPenelope::numSeedTypes=0;

//Try to open dicom with seeds data
dicom::dicomfile* dfrtplan;
dfrtplan = dicom::open_dicomfile(Dicomrtplan);
if(dfrtplan)
  {
    printf("\n*** Loading seeds planning\n");

    dicomPenelope::numSeedTypes=0;
    bool remainingSeedTypes=true; //control when all seed types was
         loaded
    while(remainingSeedTypes)
 {
   //read number "dicomPenelope::numSeedTypes" of element "
       Application Setup Sequence"
   dicom::dataset* ds=dfrtss->get_dataelement(0x300a0230)->
       dataset_at(dicomPenelope::numSeedTypes);
```

```
//check existence of seed type number "dicomPenelope::
    numSeedTypes"
if(ds!=NULL)
  {
    int channelNumber=0;
    dicomPenelope::seedsNumber[dicomPenelope::numSeedTypes]=0;
    //read channel number (cateters) from "Channel Sequence"
    bool remainingChannels=true;
    while(remainingChannels)
 {
   //read member number "channelNumber" from "Channel Sequence"
   dicom::dataset* ds2 = ds->get_dataelement(0x300a0280)->
       dataset_at(channelNumber);
   if(ds2!=NULL) //check existence of channel "channelNumber"
     {
      channelNumber++;
      double previousTime=-1; //store "Cumulative Time Weight".
          Is the timestamp when seed arrive to one checkpoint.
          Checkpoints with same "Cumulative Time Weight" as the
          previus checkpoint, happens because the seed doesn't
          wait at this point, pass on.
      //Calculate the number of seeds in this channel
      bool remainingCheckPoints=true;
      int contCheckPoints = 0;
      while(remainingCheckPoints)
    {
     //read member number "contCheckPoints" from "Brachy Control
          Point Sequence"
     dicom::dataset* ds3 = ds2->get_dataelement(0x300a02d0)->
         dataset_at(contCheckPoints);
     if(ds3!=NULL)
       {
         if(previousTime==-1) //This is the first checkpoint
      {
        previousTime = ds3 -> get_dataelement(0x300a02d6)->
            to_double();
      }
         else
      {
        double actualTime = ds3 -> get_dataelement(0x300a02d6)->
            to_double();
        if(previousTime!=actualTime)
          {
```

```
          //If times are different, a seed remain in this
             position for a while. Then, in this position we
             have a seed.
          dicomPenelope::seedsNumber[dicomPenelope::
             numSeedTypes]++;
          previousTime=actualTime;
         }
      }
        contCheckPoints++;
       }
     else
       {
        remainingCheckPoints=false;
       }
   }
    }
  else
    {
     remainingChannels=false;
    }
}

   //Now we know the number of positions. Next, create arrays
      to store data and read each position

   dicomPenelope::seedPos[dicomPenelope::numSeedTypes] = new
      double[dicomPenelope::seedsNumber[dicomPenelope::
      numSeedTypes]*3]; //3 components per seed position (x,y,z
      )

   int seedCont = 0;
   int seedChannelCont = 0;
   while(seedChannelCont<channelNumber)
{

  //read member number "seedChannelCont" from "Channel Sequence
     "
  dicom::dataset* ds2 = ds->get_dataelement(0x300a0280)->
     dataset_at(seedChannelCont);
  if(ds2!=NULL)
    {
     double previousTime=-1; //store "Cumulative Time Weight".
        Is the timestamp when seed arrive to one checkpoint.
```

```
        Checkpoints with same "Cumulative Time Weight" as the
        previus checkpoint, happens because the seed doesn't
        wait at this point, pass on.

    bool remainingCheckPoints=true;
    int contCheckPoints=0;
    while(remainingCheckPoints)
  {
   //read member number "contCheckPoints" from "Brachy Control
        Point Sequence"
   dicom::dataset* ds3 = ds2->get_dataelement(0x300a02d0)->
       dataset_at(contCheckPoints);
   if(ds3!=NULL)
     {
       if(previousTime==-1) //This is the first checkpoint
    {
      previousTime = ds3 -> get_dataelement(0x300a02d6)->
          to_double();
    }
       else
    {
      double actualTime = ds3 -> get_dataelement(0x300a02d6)->
          to_double();
      if(previousTime!=actualTime)
        {
          //If times are different, a seed remain in this
              position for a while. Then, in this position we
              have a seed.
          double* AuxPosicio;
          int AuxNumComponents;
          ds3 -> get_dataelement(0x300a02d4)->
              to_double_values_a(&AuxPosicio, &AuxNumComponents
              );

          dicomPenelope::seedPos[dicomPenelope::numSeedTypes][
              seedCont]=AuxPosicio[0]*0.1; //convert to cm
          seedCont++;
          dicomPenelope::seedPos[dicomPenelope::numSeedTypes][
              seedCont]=AuxPosicio[1]*0.1; //convert to cm
          seedCont++;
          dicomPenelope::seedPos[dicomPenelope::numSeedTypes][
              seedCont]=AuxPosicio[2]*0.1; //convert to cm
          seedCont++;
```

```
                free(AuxPosicio);
                previousTime=actualTime;
               }
           }
              contCheckPoints++;
            }
          else
            {
              remainingCheckPoints=false;
            }
        }
         }
       seedChannelCont++;
     }

       printf("%d seeds loaded of type %d\n",dicomPenelope::
           seedsNumber[dicomPenelope::numSeedTypes],dicomPenelope::
           numSeedTypes+1);

       dicomPenelope::numSeedTypes++;
     }
    else
     {
       remainingSeedTypes=false;
     }

 }
  }
else
  {
    printf("\n*** Missing seeds planing DICOM file (rtplan.dcm)\n")
        ;
  }
delete dfrtplan;

//Seeds readed.

////////////////////////
//  Read image data ////
////////////////////////

printf("\n");
```

```
//Read data from fisrt dicom (with zmin)
dicom::dicomfile* df;
df = dicom::open_dicomfile(bottomDicomFilename); //open dicom

if(df == NULL)
  {
    printf("loadDicom:Error: can't re-read fist dicom.");
    NERROR = 1000; exit(NERROR);
  }
else
  {
    printf("************ Reading dicom voxels properties\n\n");
    //Extract number of pixels in this dicom image file
    //read width, height, pixel number, number of frames and bytes
        per pixel
    int width, height;
    int precision, signedness, ncomponents, bytes_per_pixel;
    int nframes; //Planes in the image
    df->get_image_info(&width, &height,
        &precision, &signedness, &ncomponents, &bytes_per_pixel,
        &nframes);
    std::string PixelSpacing =
 df->get_dataelement("PixelSpacing")->to_string();
    //read z dimension of each image plane
    std::string SliceThickness =
 df->get_dataelement("SliceThickness")->to_string();

    geovoxMod::nx = width;
    geovoxMod::ny = height;
    geovoxMod::nxy = geovoxMod::nx*geovoxMod::ny;

    ///////////////////////////////////////////////////
    //take the origin of the lower plane of the image//
    double* AuxOrigenDicom;
    int auxInt;
    df->get_dataelement(0x00200032)->to_double_values_a(&
        AuxOrigenDicom, &auxInt);
    dicomPenelope::dicomOrigin[0]=AuxOrigenDicom[0]*0.1; //
        transform to cm
    dicomPenelope::dicomOrigin[1]=AuxOrigenDicom[1]*0.1; //
        transform to cm
```

```
   dicomPenelope::dicomOrigin[2]=AuxOrigenDicom[2]*0.1; //
      transform to cm

   printf("Dicom origin x,y,z(cm):\n");
   printf(" %10.5e %10.5e %10.5e \n",dicomPenelope::dicomOrigin
      [0],dicomPenelope::dicomOrigin[1],dicomPenelope::
      dicomOrigin[2]);

   free(AuxOrigenDicom);

   /////////////////////////////////////////////////
   //read pixel size
   float DimYPixel;
   float DimXPixel;
   sscanf(PixelSpacing.c_str(),"%f%*c%f",&DimYPixel,&DimXPixel);

   dicomPenelope::DimVoxelDicom[0]=DimXPixel/10.0; //transform to
      cm
   dicomPenelope::DimVoxelDicom[1]=DimYPixel/10.0; //transform to
      cm
   dicomPenelope::DimVoxelDicom[2]=atof(SliceThickness.c_str())
      /10.0; //transform to cm
   dicomPenelope::dimDicom[0] = width*dicomPenelope::DimVoxelDicom
      [0];
   dicomPenelope::dimDicom[1] = height*dicomPenelope::
      DimVoxelDicom[1];
   dicomPenelope::dimDicom[2] = geovoxMod::nz*dicomPenelope::
      DimVoxelDicom[2];

   geovoxMod::dx = dicomPenelope::DimVoxelDicom[0];
   geovoxMod::dy = dicomPenelope::DimVoxelDicom[1];
   geovoxMod::dz = dicomPenelope::DimVoxelDicom[2];
   printf("Voxel dimensions in x,y,z (cm):\n");
   printf(" %12.5E %12.5E %12.5E\n",geovoxMod::dx,geovoxMod::dy,
      geovoxMod::dz);

   double tmpaux=(geovoxMod::dx < geovoxMod::dy ? geovoxMod::dx :
      geovoxMod::dy);
   double tmp2aux=(tmpaux < geovoxMod::dz ? tmpaux : geovoxMod::dz
      );
   if (tmp2aux < minvoxSide)
{
```

```
  printf("loadDicom:ERROR: voxel side too small, tracking
      algorithm\n");
  printf(" requires voxel sides to be larger than (cm):%12.5E\n",
      minvoxSide);
  NERROR = 230; exit(NERROR);
}

   geovoxMod::volvox = geovoxMod::dx*geovoxMod::dy*geovoxMod::dz;
   printf("Voxels volume (cm^3):\n");
   printf("%12.5E\n",geovoxMod::volvox);

   geovoxMod::idx = 1.0E0/geovoxMod::dx;
   geovoxMod::idy = 1.0E0/geovoxMod::dy;
   geovoxMod::idz = 1.0E0/geovoxMod::dz;

   geovoxMod::vbb[0] = geovoxMod::dx*geovoxMod::nx;
   geovoxMod::vbb[1] = geovoxMod::dy*geovoxMod::ny;
   geovoxMod::vbb[2] = geovoxMod::dz*geovoxMod::nz;

   printf("Size of voxels bounding box, VBB (cm):\n");
   printf(" %12.5E %12.5E %12.5E\n",geovoxMod::vbb[0],geovoxMod::
       vbb[1],geovoxMod::vbb[2]);

   tmpaux=(geovoxMod::vbb[0] > geovoxMod::vbb[1] ? geovoxMod::vbb
       [0] : geovoxMod::vbb[1]);
   tmp2aux=(tmpaux > geovoxMod::vbb[2] ? tmpaux : geovoxMod::vbb
       [2]);

   if (tmp2aux > 1.0E5)
{
  printf("readvox:ERROR: VBB too large.\n");
  NERROR = 231; exit(NERROR);
}

   //Apply a translation to contours points to move origin to
       (0,0,0)

   int j=0;
   while(j<dicomPenelope::nContours)
{
  int i=0;
  while(i<dicomPenelope::nContoursPlanes[j])
    {
```

```
     int k=0;
     while(k<dicomPenelope::nPlanePoints[j][i])
   {
    dicomPenelope::contoursPoints[j][i][k]-=dicomPenelope::
        dicomOrigin[0];
    k++;
    dicomPenelope::contoursPoints[j][i][k]-=dicomPenelope::
        dicomOrigin[1];
    dicomPenelope::contoursPoints[j][i][k]=-dicomPenelope::
        contoursPoints[j][i][k]+dicomPenelope::dimDicom[1]; //
        apply a reflexion
    k++;
    dicomPenelope::contoursPoints[j][i][k]-=dicomPenelope::
        dicomOrigin[2];
    k++;
   }
     i++;
    }
  j++;
}

   //Apply a translation to seeds positions to move origin to
       (0,0,0)

   j=0;
   printf(" *Transform seed positions* \n");
   while(j<dicomPenelope::numSeedTypes)
{
  printf(" Type %d\n",j);
  int i=0;
  int nseed = 0;
  while(i<dicomPenelope::seedsNumber[j]*3) // 3 components (x,y,z)
    {
     printf("   Seed %d: %10.5e %10.5e %10.5e to ",nseed,
         dicomPenelope::seedPos[j][nseed], dicomPenelope::seedPos[
         j][nseed+1], dicomPenelope::seedPos[j][nseed+2]);
     dicomPenelope::seedPos[j][i]-=dicomPenelope::dicomOrigin[0];
     i++;
     dicomPenelope::seedPos[j][i]-=dicomPenelope::dicomOrigin[1];
     dicomPenelope::seedPos[j][i]=-dicomPenelope::seedPos[j][i]+
         dicomPenelope::dimDicom[1]; //reflexion
     i++;
     dicomPenelope::seedPos[j][i]-=dicomPenelope::dicomOrigin[2];
```

```
      i++;

      printf("%10.5e %10.5e %10.5e \n",dicomPenelope::seedPos[j][
          nseed], dicomPenelope::seedPos[j][nseed+1], dicomPenelope
          ::seedPos[j][nseed+2]);
      nseed++;
    }
   j++;
 }

  }

delete df;


geovoxMod::nvox = geovoxMod::nxy*geovoxMod::nz;
dicomPenelope::dicomNPixels[0] = geovoxMod::nx;
dicomPenelope::dicomNPixels[1] = geovoxMod::ny;
dicomPenelope::dicomNPixels[2] = geovoxMod::nz;

if(geovoxMod::nvox > nvoxmax)
  {
    printf("loadDicom:ERROR: No. voxels exceeds %d\n", nvoxmax);
    NERROR = 226; exit(NERROR);
  }
// Allocate arrays:
mem = (2*dataTypesMod::sizeOfInt+dataTypesMod::sizeOfReal)*float(
    geovoxMod::nvox);
printf("Memory required to allocate voxel arrays (MB):\n");
printf(" %12.5E\n", mem*1.0E-6);

geovoxMod::matvox        = new (std::nothrow) int [geovoxMod::nvox
    ];
geovoxMod::densvox       = new (std::nothrow) float [geovoxMod::
    nvox];
dicomPenelope::dicomImage = new (std::nothrow) int [geovoxMod::nvox
    ];

if (geovoxMod::matvox == NULL || geovoxMod::densvox == NULL)
  {
    printf("loadDicom:ERROR: not enough memory.\n");
    NERROR = 227; exit(NERROR);
  }
```

```
geovoxMod::memvox = geovoxMod::memvox+mem;
                                 // Store for later use by voxels
    dose report


printf("\n");

//Order dicom Z values
order(NumDicoms,ZPlanes,nFilePixels);

closedir(auxDir);
int lodadedDicoms = 0;

for(int k = 0; k < totalFiles; k++)       //Read one file data in
    each iteration and save filename
  {
    sprintf(AuxChar,"%s",filenamesDicom[k]); //extract filename

    if(AuxChar[0] == '.')         //skip hide files (.filename)
 {
   continue;
 }
    int imagePos = -1; //Save image position
    char Modality[500]; //store dicom image modality
    int firstPixelPos = 0; //stores first pixel to be filled

    int j=0;
    //check if it's a image dicom
    dicom::dicomfile* dfNum;
    dicom::dataelement* deNum;
    dfNum = dicom::open_dicomfile(AuxChar);
    if(dfNum!=NULL)
 {
   deNum = dfNum->get_dataelement(0x00080060);
   if(deNum -> is_valid())
     {
       sprintf(Modality,"%s",(deNum -> to_string()).c_str());
       if(strcmp(Modality, dicomPenelope::imageModality) == 0)
    {
      //Check image position
      deNum = dfNum -> get_dataelement(0x00200032);
      if(deNum -> is_valid())
        {
```

```
     //read Z position of this file
     double* AuxPos;
     int AuxNumComp;
     deNum -> to_double_values_a(&AuxPos, &AuxNumComp);

     imagePos = searchArray(NumDicoms, ZPlanes, AuxPos[2]);
     free(AuxPos);
     if(imagePos==-1)
       {
         delete dfNum;
         continue;
       }
    }
      else
    {
     //this dicom has not image data
     delete dfNum;
     continue;
    }

     }
   else
     {
      //This dicom modality is not accepted
      delete dfNum;
      continue;
     }
 }
    else
 {
   //Can't read modality from this dicom
   delete dfNum;
   continue;
 }
  }
else
  {
    //Is not a dicom
    delete dfNum;
    continue;
  }
delete dfNum;
```

```
//Calculate first pixel position for this dicom
for(int k = 0; k < imagePos; k++)
  {
    firstPixelPos += nFilePixels[k];
  }

//open dicom
df=dicom::open_dicomfile(AuxChar);
if(df)
  {
    //read width, height, pixel number, number of frames and
        bytes per pixel
    int width, height;
    int precision, signedness, ncomponents, bytes_per_pixel;
    int nframes; //Planes in the image
    df->get_image_info(&width, &height,
        &precision, &signedness, &ncomponents, &bytes_per_pixel,
        &nframes);

    //read pixel format RGB, MONOCHROME2, MONOCHROME1 etc
    std::string PhotometricInterpretation =
 df->get_dataelement("PhotometricInterpretation")->to_string();
    //read of pixel data is signed (1) or unsigned (0)
    int PixelRep =
 df->get_dataelement("PixelRepresentation")->to_int();

    //RescaleIntercept and RescaleSlope, are used to transform
        each pixel value with the formula: RescaleSlope*
        pixelValue+RescaleIntercept
    int RescaleIntercept =
 df->get_dataelement("RescaleIntercept")->to_int();
    int RescaleSlope =
 df->get_dataelement("RescaleSlope")->to_int();
    if(RescaleSlope==0){RescaleSlope=1;}

    //
        ////////////////////////////////////////////////////////////////////////

    int SamplesPerPixel =
 df->get_dataelement("SamplesPerPixel")->to_int();
    //read number of reserved bits in the buffer for each "
        sample" of each pixel
    //int AllocatedBits =
```

```
   //  df->get_dataelement("BitsAllocated")->to_int();
   //read how many reserved bits are used to store data
   int StoredBits =
df->get_dataelement("BitsStored")->to_int();
   //HighBit tell us where is the last bit. We need this
       information to check if pixels was stored in big or
       little endian
   int HighBit =
df->get_dataelement("HighBit")->to_int();
   std::string studydesc =
df->get_dataelement("StudyDescription")->to_string();

   //Check if this image uses monochromatic format ("
       SamplesPerPixel" = 1)

   if(SamplesPerPixel!=1)
{
  printf("Error 'load_dicom': The image format is not grayscale
      \n");
  delete df;
  NERROR = 1000; exit(NERROR);
}

   //Create a buffer to store pixel data
   char* pixeldata= new char[width*height*bytes_per_pixel];
   int pixeldata_length;   //store number of pixels
   //load pixels
   df->get_pixeldata_a(&pixeldata, &pixeldata_length);

   //take position (0,0,0) at top left corner.
   if (pixeldata)
{
  //Check dimensions
  if(dicomPenelope::dicomNPixels[0]!=width || dicomPenelope::
      dicomNPixels[1]!=height)
    {
      printf("Error 'load_dicom': DICOMs image size mismatched\n
          ");
      delete df;
      delete pixeldata;
      NERROR = 1000; exit(NERROR);
    }
```

```
int i=0;    // x axis (width)
j=0;            // y axis (height)
int k=0;    // z axis (frames)
int Nbyte=0;
while(k<nframes)
  {
    int zIndex = k*height*width;
    j=0;
    while(j<height)
 {
   int zyIndex = zIndex+(height-1-j)*width; //Image origin is
       on top left corner
   i=0;
   while(i<width)
     {
       std::string Aux;
       Aux.clear();
       int cont=0;
       while(cont<bytes_per_pixel)
    {
      Aux[cont]=pixeldata[Nbyte];
      cont++;
      Nbyte++;
    }

       //convert pixel bytes to integer

       if(PixelRep==0)  //unsigned
    {
      unsigned int auxint1=0;
      if(HighBit==StoredBits-1) //read left to right
        {
          cont=bytes_per_pixel-1;
          while(cont>=0)
       {
         auxint1 = (auxint1 << 8) +Aux[cont];
         cont--;
       }
        }
      else if(HighBit==0) //read right to left
        {
          cont=0;
          while(cont<bytes_per_pixel)
```

```
         {
           auxint1 = (auxint1 << 8) +Aux[cont];
           cont++;
         }
          }
        //apply calibration line on pixel value
        dicomPenelope::dicomImage[firstPixelPos + zyIndex + i] =
             RescaleSlope*auxint1+RescaleIntercept;
      }
         else if(PixelRep==1) //signed
      {
        int auxint1=0;
        if(HighBit==StoredBits-1) //read left to right
          {
            cont=bytes_per_pixel-1;
            while(cont>=0)
         {
           auxint1 = (auxint1 << 8) +Aux[cont];
           cont--;
         }
          }
        else if(HighBit==0) //read right to left
          {
            cont=0;
            while(cont<bytes_per_pixel)
         {
           auxint1 = (auxint1 << 8) +Aux[cont];
           cont++;
         }
          }
        //apply calibration line on pixel value
        dicomPenelope::dicomImage[firstPixelPos + zyIndex + i] =
             RescaleSlope*auxint1+RescaleIntercept;
      }
         i++;
       }
     j++;
   }
      k++;
    }

}
   else
```

```
   {
     printf("Error in loadDicom\n");
     delete pixeldata;
     NERROR = 1000; exit(NERROR);
   }
    }
  else
    {
      printf("Error in 'loadDicom': can't open dicom: %s\n",
          AuxChar);
      NERROR = 1000; exit(NERROR);
    }
  delete df;

  lodadedDicoms++;
 }

//////////////////////////
// Density assign ////////
//////////////////////////

//Pixel transformation depends on modality
if(strcmp(dicomPenelope::imageModality,"CT") == 0)
  {
    for(int i = 0; i < geovoxMod::nvox; i++)
 {
   //Calculate electronic density relative to water with polynomial
        calibration of scanner
   float DensElec=0;
   int cont=0;
   while(cont<dicomPenelope::nCoefficients)
     {
       DensElec+=dicomPenelope::coefficients[cont]*pow(
           dicomPenelope::dicomImage[i],cont);
       cont++;
     }
   //with the TG_186 fit, obtain voxel density
   geovoxMod::densvox[i] = -0.1746+1.176*DensElec;
   if(geovoxMod::densvox[i] < 0)
     {
       //Check for negative densities
       geovoxMod::densvox[i] = 0.0;
     }
```

```
 }
  }
else if(strcmp(dicomPenelope::imageModality,"US") == 0)
  {
    for(int i = 0; i < geovoxMod::nvox; i++)
 {
   //Assing density according to contour material. By default
       assign 1.0 (water)
   geovoxMod::densvox[i] = 1.0;
   geovoxMod::matvox[i] = dicomPenelope::defaultMaterial;
 }
  }

///////////////////////////
//material assign ////////
///////////////////////////

if(strcmp(dicomPenelope::imageModality,"CT") == 0)
  {

    for(int i = 0; i < geovoxMod::nvox; i++)
 {
   int cont=0;
   bool assignedMaterial=false;
   while(cont<dicomPenelope::NumMatsCal)
     {
       if(geovoxMod::densvox[i] >= dicomPenelope::lowMatLimit[cont]
            && geovoxMod::densvox[i] < dicomPenelope::topMatLimit[
           cont])
    {
      geovoxMod::matvox[i]=dicomPenelope::matID[cont];
      assignedMaterial=true;
      break;
    }
       cont++;
     }
   //assign same material to all voxels aout of density ranges
   if(!assignedMaterial)
     {
       geovoxMod::matvox[i] = dicomPenelope::defaultMaterial;
       dicomPenelope::nPixelsOutRange++;
     }
 }
```

```
  }

if( dicomPenelope::nContours > 0 )
  {

    int* zPlaneContainingVoxel = new int[dicomPenelope::nContours
       +1]; //Store the plane number of each contour containing
       this voxel frame. (+1 to avoid create empty array)
    double PosVoxel[3];
    int nextPixel = 0;
    for(int k = 0; k < geovoxMod::nz; k++)
 {
   PosVoxel[2]=(k+0.5)*dicomPenelope::DimVoxelDicom[2];
   double previousVoxelZpos=(k == 0 ? 0.0 : k-0.5)*dicomPenelope::
      DimVoxelDicom[2];
   //Search contour plane number containing this voxel frame
   for(int j = 0; j < dicomPenelope::nContours; j++)
     {
      zPlaneContainingVoxel[j]=-1;
      for(int i = 0; i < dicomPenelope::nContoursPlanes[j]; i++)
    {
     if(dicomPenelope::contoursPoints[j][i][2]>previousVoxelZpos
         && dicomPenelope::contoursPoints[j][i][2]<=PosVoxel[2])
       {
         zPlaneContainingVoxel[j]=i;
         break;
       }
    }
     }
   for(int j = 0; j < geovoxMod::ny; j++)
     {
      PosVoxel[1]=(j+0.5)*dicomPenelope::DimVoxelDicom[1]; //
          calculate position Y
      for(int i = 0; i < geovoxMod::nx; i++)
    {
     PosVoxel[0]=(i+0.5)*dicomPenelope::DimVoxelDicom[0]; //
         calculate position X
     //Check if this voxel is in a contour
     bool inContour[MAX_CONTOURS]={0};

     //To verify if the voxel is within the contour, we draw a
         straight line with the fixed x component and equal to the
         component x of the voxel with a source with y = -1, to
```

```
    ensure it is out of the contour, and that everything is
    going through The space. When interact with the segments
    that delimit the plane with constant z of the contour, we
    will know when we are in and out of the contour and so if
    the voxel is inside or outside.

//create origin of the line
double Origen_Recta[2]={PosVoxel[0],-1.0}; // {x,y}
for(int z2 = 0; z2 <dicomPenelope::nContours; z2++)
  {
    inContour[z2]=false;

    //check if some plane of this contour contains voxel
    if(zPlaneContainingVoxel[z2]==-1)
 {
   //check next contour
   continue;
 }

    double* Y_Tall = new double[dicomPenelope::nPlanePoints[z2
        ][zPlaneContainingVoxel[z2]]]; //Store y coordinate
        where line cut some contour segment
    int cutsNumber = 0; //stores how many times line cut
        contour segments
    ////////////////////////////////////
    //Search intersections/////////////
    ////////////////////////////////////
    int n=0;
    while(n<dicomPenelope::nPlanePoints[z2][
        zPlaneContainingVoxel[z2]])
 {
   int xNextPoint;
   if(n==dicomPenelope::nPlanePoints[z2][zPlaneContainingVoxel
       [z2]]-3)
     {
       //this is the last point, take point number 0 as next
           point
       xNextPoint = 0;
     }
   else{xNextPoint = n+3;}

   if(((dicomPenelope::contoursPoints[z2][
       zPlaneContainingVoxel[z2]][n]>=Origen_Recta[0] &&
```

```
     dicomPenelope::contoursPoints[z2][zPlaneContainingVoxel
     [z2]][xNextPoint]<=Origen_Recta[0]) || (dicomPenelope::
     contoursPoints[z2][zPlaneContainingVoxel[z2]][n]<=
     Origen_Recta[0] && dicomPenelope::contoursPoints[z2][
     zPlaneContainingVoxel[z2]][xNextPoint]>=Origen_Recta
     [0])) && (dicomPenelope::contoursPoints[z2][
     zPlaneContainingVoxel[z2]][n]!=dicomPenelope::
     contoursPoints[z2][zPlaneContainingVoxel[z2]][
     xNextPoint]))
    {
     //To know if you cut the segment between the 2 points,
         let's look at two things:
     //1-> That the component x of a point be on the right
         side of the line or on it and that the component x
         on the other is on the left or on it.
     //2-> That the two points do not have the same X
         component, since if it fulfills the first one but
         not the second one, the straight line will pass
         through the 2 points and therefore parallel to the
         segment that they form and will not be entered or
         leaving the surface of the contour.
     double pendent = (dicomPenelope::contoursPoints[z2][
         zPlaneContainingVoxel[z2]][xNextPoint+1]-
         dicomPenelope::contoursPoints[z2][
         zPlaneContainingVoxel[z2]][n+1])/(dicomPenelope::
         contoursPoints[z2][zPlaneContainingVoxel[z2]][
         xNextPoint]-dicomPenelope::contoursPoints[z2][
         zPlaneContainingVoxel[z2]][n]); //Pendent del
         segment entre els punts
     Y_Tall[cutsNumber] = Origen_Recta[0]*pendent+
         dicomPenelope::contoursPoints[z2][
         zPlaneContainingVoxel[z2]][n+1]-pendent*
         dicomPenelope::contoursPoints[z2][
         zPlaneContainingVoxel[z2]][n];
     cutsNumber++;
    }
  n=n+3;
}
   //order cuts y coordinates (minor to major)
   order(cutsNumber,Y_Tall);

   n=0;
   bool inner = false;
```

```
    while(n<cutsNumber)
 {
   if(PosVoxel[1]<Y_Tall[n])
     {
      //if point y is before next cut stop bucle
      break;
     }
   if(inner==false){inner=true;}else{inner=false;}
   n++;
 }
    inContour[z2]=inner;
    delete [] Y_Tall;
  }

//now, assign material

int materialPriority=0; //Save actual contour priority
for(int z2 = 0; z2 <dicomPenelope::nContours; z2++)
  {
    if(inContour[z2]==true)
 {
   if(dicomPenelope::priorities[z2]>materialPriority) //check
       if this contour priority is greater than actual
     {
      geovoxMod::matvox[nextPixel] = dicomPenelope::
          contourMatID[z2];
      materialPriority = dicomPenelope::priorities[z2];

      if(strcmp(dicomPenelope::imageModality,"US") == 0)
    {
     //Change density only on US modality
     geovoxMod::densvox[nextPixel] = dicomPenelope::density[
         z2];
    }
     }
 }
  }
if(materialPriority != 0)
  {
    //If is in a contour, increment number of voxels in this
        contour
    for(int z2 = 0; z2 <dicomPenelope::nContours; z2++)
 {
```

```
          if(geovoxMod::matvox[nextPixel]==dicomPenelope::
            contourMatID[z2])
           {
            //printf("AHA!, material %d contorn %d\n",geovoxMod::
               matvox[nextPixel],z2);
            dicomPenelope::nVoxContour[z2]++;
            break;
           }
         }
          }
        nextPixel++;
      }
       }
  }
      delete [] zPlaneContainingVoxel;
    }

  if(dicomPenelope::nPixelsOutRange > 0)
    {
      printf("loadDicom:Warning: %d voxels out of range.\n",
          dicomPenelope::nPixelsOutRange);
    }

  printf("\n");

  //free memory
  delete [] nFilePixels;
  delete [] ZPlanes;
  for(int k = 0; k < num_zPlanes; k++)
    {
      delete [] filenamesDicom[k];
    }
  delete [] filenamesDicom;

#ifdef _PENELOPE_WITH_MPI_

  //Send voxel information to all processes
  //Bcast geovoxMod variables (rank 0).

  int gevoxIntVars[5] = {geovoxMod::nx, geovoxMod::ny, geovoxMod::nz,
       nmatvox, dicomPenelope::numSeedTypes};
  double geovoxDoubleVars[3] = {geovoxMod::dx, geovoxMod::dy,
      geovoxMod::dz};
```

```
MPI_Bcast(gevoxIntVars  , 5, MPI_INT  , 0, MPI_COMM_WORLD);
MPI_Bcast(geovoxDoubleVars, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);

MPI_Bcast(dicomPenelope::seedsNumber, dicomPenelope::numSeedTypes,
   MPI_INT, 0, MPI_COMM_WORLD);

for(int i = 0; i < dicomPenelope::numSeedTypes; i++)
  {
   MPI_Bcast(dicomPenelope::seedPos[i], dicomPenelope::seedsNumber
       [i]*3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  }

//Bcast voxels information (rank 0).
MPI_Bcast(geovoxMod::matvox , geovoxMod::nvox, MPI_INT , 0,
   MPI_COMM_WORLD);
MPI_Bcast(geovoxMod::densvox, geovoxMod::nvox, MPI_FLOAT, 0,
   MPI_COMM_WORLD);

  } //end of rank 0 if
else
  {

   //Recive voxels information (rank > 0)
   //Bcast geovoxMod variables.
   int gevoxIntVars[5];
   double geovoxDoubleVars[3];
   MPI_Bcast(gevoxIntVars  , 5, MPI_INT  , 0, MPI_COMM_WORLD);
   MPI_Bcast(geovoxDoubleVars, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   dicomPenelope::numSeedTypes = gevoxIntVars[4];

   MPI_Bcast(dicomPenelope::seedsNumber, dicomPenelope::
      numSeedTypes, MPI_INT, 0, MPI_COMM_WORLD);

   for(int i = 0; i < dicomPenelope::numSeedTypes; i++)
 {
  dicomPenelope::seedPos[i] = new double[dicomPenelope::
     seedsNumber[i]*3]; //3 components per seed position (x,y,z)
  MPI_Bcast(dicomPenelope::seedPos[i], dicomPenelope::seedsNumber[
     i]*3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
 }
```

```
   nmatvox = gevoxIntVars[3];
   dicomPenelope::NumMatsCal = nmatvox;

   geovoxMod::nx = gevoxIntVars[0];
   geovoxMod::ny = gevoxIntVars[1];
   geovoxMod::nz = gevoxIntVars[2];

   dicomPenelope::dicomNPixels[0] = geovoxMod::nx;
   dicomPenelope::dicomNPixels[1] = geovoxMod::ny;
   dicomPenelope::dicomNPixels[2] = geovoxMod::nz;

   geovoxMod::dx = geovoxDoubleVars[0];
   geovoxMod::dy = geovoxDoubleVars[1];
   geovoxMod::dz = geovoxDoubleVars[2];

   dicomPenelope::DimVoxelDicom[0] = geovoxMod::dx;
   dicomPenelope::DimVoxelDicom[1] = geovoxMod::dy;
   dicomPenelope::DimVoxelDicom[2] = geovoxMod::dz;

   geovoxMod::nxy = geovoxMod::nx*geovoxMod::ny;
   geovoxMod::volvox = geovoxMod::dx*geovoxMod::dy*geovoxMod::dz;
   geovoxMod::idx = 1.0E0/geovoxMod::dx;
   geovoxMod::idy = 1.0E0/geovoxMod::dy;
   geovoxMod::idz = 1.0E0/geovoxMod::dz;

   geovoxMod::vbb[0] = geovoxMod::dx*geovoxMod::nx;
   geovoxMod::vbb[1] = geovoxMod::dy*geovoxMod::ny;
   geovoxMod::vbb[2] = geovoxMod::dz*geovoxMod::nz;

   geovoxMod::nvox = geovoxMod::nxy*geovoxMod::nz;

   //Allocate arrays
   geovoxMod::matvox       = new (std::nothrow) int [geovoxMod::
       nvox];
   geovoxMod::densvox      = new (std::nothrow) float [geovoxMod::
       nvox];

   if (geovoxMod::matvox == NULL || geovoxMod::densvox == NULL)
{
  printf("loadDicom:ERROR: not enough memory.\n");
  NERROR = 227; exit(NERROR);
}
```

```
      geovoxMod::memvox = geovoxMod::memvox+mem;     // Store for
         later use by voxels dose report

      //Bcast voxels information (rank > 0).
      MPI_Bcast(geovoxMod::matvox , geovoxMod::nvox, MPI_INT , 0,
         MPI_COMM_WORLD);
      MPI_Bcast(geovoxMod::densvox, geovoxMod::nvox, MPI_FLOAT, 0,
         MPI_COMM_WORLD);

    }


#endif


}
```

B Apèndix: “wrapper” MPI

```
#ifdef _USE_MPI_ //Defined in cmake if MPI is enabled

#ifndef _PENELOPE_WITH_MPI_
#define _PENELOPE_WITH_MPI_ 1

#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

#ifdef scanf
#undef scanf
#endif

#ifdef fscanf
#undef fscanf
#endif

#define _MPI_COMM_TAG_ 11
#define _MPI_GETPAR_STACK_SIZE_ 2000
```

```
//Define wrapper routines for MPI

extern "C" double realtime(void);

namespace PENELOPE_MPI
{
  int mpi_scanf(const char*, ...);
  int mpi_fscanf(FILE*, const char*, ...);
  int mpi_printf(const char* format, ...);
  char* mpi_fgets(char *str, int n, FILE *stream);
  FILE* mpi_fopen(const char *filename, const char *mode);
  int mpi_fclose(FILE*);
  void mpi_balance();
  void mpi_splitPhaseSpaceFile(char * psfname);
  bool mpi_getpar();
  FILE* mpi_fopenLocal(const char *filename, const char *opt);

  class getparData
  {
  public:
    int intData[7];
    double doubleData[8];
  };

  getparData getParStack[_MPI_GETPAR_STACK_SIZE_];
  int nStackPars = 0;
}

int PENELOPE_MPI::mpi_scanf(const char* format, ...)
{
  va_list par_list;
  va_start(par_list, format);

  //Take proces rank
  int rank;

  char format_copy[1000];
  char processed_format[1000];

  int nElements = 0; //number of elements readed
  char data[1000]; //Stores packaged data
  int strings_length[1000]; //Stores the lengths of data strings
  int nStrings = 0; //number of strings in data
```

```
int dataSize = 1000; //max data size

int position = 0; //position in "data" to store next element

int format_len = strlen(format);

//Check format length
if(format_len >= dataSize)
  {
    printf("\nError in mpi_scanf: format length too long\n");
    exit(1000);
  }

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

strcpy(format_copy,format);

//Check if this is process number 0
if(rank == 0)
  {
    nElements = vscanf(format, par_list);
    if(nElements < 0)
 {
   printf("\nError in mpi_scanf: can't read correctly\n");
   exit(1000);
 }

    if(nElements > 0) //Check if some argument was readed
 {
   //Package the read data
   int cpy_formatlen = format_len;
   int i = 0;
   while(i < nElements)
     {
       //Find next %
       char *pvar = strstr(format_copy,"%");
       if(pvar == NULL)
    {
      printf("\nError in mpi_scanf: Diferent number of readed and
          sended elements \n");
      exit(1000);
    }
```

```
   int len = cpy_formatlen-(pvar-format_copy);
   bool islong = false;
   bool point = false;
   for(int j = 1; j < len; j++) //j=0 is '%'
{
  if(pvar[j] == '*')
    {
     break;
    }

  if(pvar[j] == 'l' && !islong)
    {
     islong = true;
     continue;
    }
  else if(pvar[j] == '.' && !point)
    {
     point = true;
     continue;
    }

  if(pvar[j] == 'd')
    {
     if(islong)
   {
    long int *auxInt = va_arg(par_list,long int*);
    MPI_Pack(auxInt, 1, MPI_LONG, data, dataSize, &position,
        MPI_COMM_WORLD);
   }
     else
   {
    int *auxInt = va_arg(par_list,int*);
    MPI_Pack(auxInt, 1, MPI_INT, data, dataSize, &position,
        MPI_COMM_WORLD);
   }
     i++; //increment founded elements
     break;
    }
  else if(pvar[j] == 'f' || pvar[j] == 'e' || pvar[j] == 'E')
    {
     if(islong)
   {
    double *auxDouble = va_arg(par_list,double*);
```

```
   MPI_Pack(auxDouble, 1, MPI_DOUBLE, data, dataSize, &
       position, MPI_COMM_WORLD);
 }
    else
 {
   float *auxFloat = va_arg(par_list,float*);
   MPI_Pack(auxFloat, 1, MPI_FLOAT, data, dataSize, &position,
        MPI_COMM_WORLD);
 }
    i++; //increment founded elements
    break;
  }
else if(pvar[j] == 'c')
  {
    char *auxChar = va_arg(par_list,char*);

    //Check if is a string or a char
    int auxLen = strlen(auxChar);
    if(auxLen > 1)
 {
   //is a string
   strings_length[nStrings] = auxLen; //Save string length
   MPI_Pack(auxChar, strings_length[nStrings], MPI_CHAR, data,
        dataSize, &position, MPI_COMM_WORLD);
   nStrings++; //increment number of strings
   i++; //increment founded elements
   break;
 }
    else
 {
   //is a char
   MPI_Pack(auxChar, 1, MPI_CHAR, data, dataSize, &position,
       MPI_COMM_WORLD);
   i++; //increment founded elements
   break;
 }
  }
else if(pvar[j] == 's' || pvar[j] == '[')
  {
    char *auxChar = va_arg(par_list,char*);
    strings_length[nStrings] = strlen(auxChar); //Save string
        length
```

```
          MPI_Pack(auxChar, strings_length[nStrings], MPI_CHAR, data
              , dataSize, &position, MPI_COMM_WORLD);
          nStrings++; //increment number of strings
          i++; //increment founded elements
          break;
        }
      else if(pvar[j] != ’0’ && pvar[j] != ’1’ && pvar[j] != ’2’ &&
           pvar[j] != ’3’ && pvar[j] != ’4’ && pvar[j] != ’5’ &&
          pvar[j] != ’6’ && pvar[j] != ’7’ && pvar[j] != ’8’ && pvar
          [j] != ’9’ && pvar[j] != ’ ’)
        {
          //no variable type identificator or number
          printf("\nError in mpi_scanf: invalid format ’%c’ reading
              element %d\n", pvar[j], nElements);
          exit(1000);
        }

      if(islong)
        {
          //no variable type identificator after ’l’
          printf("\nError in mpi_scanf: invalid variable type ID\n")
              ;
          exit(1000);
        }
    }
       cpy_formatlen = len;
       memmove(format_copy, pvar+1, len); //copy end of array
     }
   if(i != nElements)
     {
       printf("\nError in mpi_scanf: Diferent number of readed and
           sended elements \n");
       printf("    Readed: %d\n",nElements);
       printf("    Sended: %d\n",i);
       exit(1000);
     }
 }
    //add the final data possition to strings_length array
    strings_length[nStrings] = position;
    //add the return value of vscanf
    strings_length[nStrings+1] = nElements;
  }
else
```

```
 {
   //Read format
   int cpy_formatlen = format_len;
   while(1)
{
  //Find next %
  char *pvar = strstr(format_copy,"%");
  if(pvar == NULL)
    {
      break;
    }

  int len = cpy_formatlen-(pvar-format_copy);
  bool islong = false;
  bool point = false;
  char num[20];
  int numcont = 0;
  for(int j = 1; j < len; j++) //j=0 is '%'
    {
      if(pvar[j] == '*'){break;}
      if(pvar[j] == 'l' && !islong)
   {
     islong = true;
     continue;
   }

      if(pvar[j] == '.' && !point)
   {
     point = true;
     continue;
   }

      if(pvar[j] == 'd')
   {
     if(islong)
       {
         processed_format[nElements] = 'I';
         position += sizeof(long int);
       }
     else
       {
         processed_format[nElements] = 'i';
         position += sizeof(int);
```

```
    }
  nElements++;
  break;
}
   else if(pvar[j] == ’f’ || pvar[j] == ’e’ || pvar[j] == ’E’)
{
  if(islong)
    {
      processed_format[nElements] = ’d’;
      position += sizeof(double);
    }
  else
    {
      processed_format[nElements] = ’f’;
      position += sizeof(float);
    }
  nElements++;
  break;
}
   else if(pvar[j] == ’c’)
{
  //Check if is a string or a char
  double auxNum = -1.0;
  if(numcont > 0){num[numcont] = ’\0’; auxNum = atof(num);}
  else{auxNum = 1.0;}

  if(auxNum >=2.0)
    {
      //is a string
      processed_format[nElements] = ’s’;
      nStrings++;
      nElements++;
      break;
    }
  else
    {

      //is a char
      processed_format[nElements] = ’c’;
      nElements++;
      break;
    }
}
```

```
      else if(pvar[j] == 's' || pvar[j] == '[')
    {
     processed_format[nElements] = 's';
     nStrings++;
     nElements++;
     break;
    }
      else if(pvar[j] != '0' && pvar[j] != '1' && pvar[j] != '2'
          && pvar[j] != '3' && pvar[j] != '4' && pvar[j] != '5' &&
          pvar[j] != '6' && pvar[j] != '7' && pvar[j] != '8' &&
          pvar[j] != '9' && pvar[j] != ' ')
    {
     //no variable type identificator or number
     printf("\nError in mpi_scanf: invalid format '%c' reading
         element %d\n", pvar[j], nElements);
     exit(1000);
    }

      num[numcont] = pvar[j];
      numcont++;

      if(islong)
    {
     //no variable type identificator after 'l'
     printf("\nError in mpi_scanf: invalid variable type ID\n");
     exit(1000);
    }
     }
   cpy_formatlen = len;
   memmove(format_copy, pvar+1, len); //copy end of array
 }
  }


//Bcast string lengths
//printf("\n Send/Recive %d with %d strings of '%s', %d\n",
    nElements, nStrings, format, rank);
MPI_Bcast(strings_length, nStrings+2, MPI_INT, 0, MPI_COMM_WORLD);
//printf("\n Sended/Recived %d with %d strings of '%s', %d\n",
    nElements, nStrings, format, rank);

//add string lenth to total data size
if(rank != 0)
```

```
  {
    for(int i = 0; i < nStrings; i++)
 {
   position += strings_length[i];
 }
    //Check total data size
    if(strings_length[nStrings] != position)
 {
   printf("\nError in mpi_scanf: data size of root proces (0) does
       not match with proces %d\n", rank);
   printf("    root: %d\n",strings_length[nStrings]);
   printf("%9d: %d\n",rank,position);
   exit(1000);
 }
    else if(strings_length[nStrings+1] != nElements)
 {
   printf("\nError in mpi_scanf: element number of root proces (0)
       does not match with proces %d\n", rank);
   exit(1000);
 }
  }

//Bcast package
if(nElements > 0)
  {
    MPI_Bcast(data, position, MPI_PACKED, 0, MPI_COMM_WORLD);

    if(rank != 0)
 {
   int extractPos = 0;
   int stringCount = 0;
   for(int i = 0; i < nElements; i++)
     {
       if(processed_format[i] == 'I') //long int
    {
      long int *auxInt = va_arg(par_list,long int*);
      MPI_Unpack(data, dataSize, &extractPos, auxInt, 1, MPI_LONG,
          MPI_COMM_WORLD);
    }
       else if(processed_format[i] == 'i') //int
    {
      int *auxInt = va_arg(par_list,int*);
```

```
       MPI_Unpack(data, dataSize, &extractPos, auxInt, 1, MPI_INT,
           MPI_COMM_WORLD);
     }
        else if(processed_format[i] == ’f’) //float
     {
       float *auxFloat = va_arg(par_list,float*);
       MPI_Unpack(data, dataSize, &extractPos, auxFloat, 1,
           MPI_FLOAT, MPI_COMM_WORLD);
     }
        else if(processed_format[i] == ’d’) //double
     {
       double *auxDouble = va_arg(par_list,double*);
       MPI_Unpack(data, dataSize, &extractPos, auxDouble, 1,
           MPI_DOUBLE, MPI_COMM_WORLD);
     }
        else if(processed_format[i] == ’c’) //char
     {
       char *auxChar = va_arg(par_list,char*);
       MPI_Unpack(data, dataSize, &extractPos, auxChar, 1, MPI_CHAR,
            MPI_COMM_WORLD);
     }
        else if(processed_format[i] == ’s’) // c string
     {
       char *auxChar = va_arg(par_list,char*);
       MPI_Unpack(data, dataSize, &extractPos, auxChar,
           strings_length[stringCount], MPI_CHAR, MPI_COMM_WORLD);
       //add end of string character
       auxChar[strings_length[stringCount]] = ’\0’;
       stringCount++;
     }
      }
   }
    }
  va_end(par_list);

  return nElements;
}
int PENELOPE_MPI::mpi_fscanf(FILE* fileIn,const char* format, ...)
{
  va_list par_list;
  va_start(par_list, format);

  //Take proces rank
```

```
int rank;

char format_copy[1000];
char processed_format[1000];

int nElements = 0; //number of elements readed
char data[1000]; //Stores packaged data
int strings_length[1000]; //Stores the lengths of data strings
int nStrings = 0; //number of strings in data
int dataSize = 1000; //max data size

int position = 0; //position in "data" to store next element

int format_len = strlen(format);

//Check format length
if(format_len >= dataSize)
  {
    printf("\nError in mpi_scanf: format length too long\n");
    exit(1000);
  }

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

strcpy(format_copy,format);

//Check if this is process number 0
if(rank == 0)
  {
    nElements = vfscanf(fileIn,format, par_list);
    if(nElements < 0)
 {
   printf("\nError in mpi_fscanf: can't read correctly\n");
   exit(1000);
 }

    if(nElements > 0) //Check if some argument was readed
 {
   //Package the read data
   int cpy_formatlen = format_len;
   int i = 0;
   while(i < nElements)
     {
```

```
   //Find next %
   char *pvar = strstr(format_copy,"%");
   if(pvar == NULL)
{
  printf("\nError in mpi_fscanf: Diferent number of readed and
      sended elements \n");
  exit(1000);
}

   int len = cpy_formatlen-(pvar-format_copy);
   bool islong = false;
   bool point = false;
   for(int j = 1; j < len; j++) //j=0 is '%'
{
  if(pvar[j] == '*')
    {
      break;
    }

  if(pvar[j] == 'l' && !islong)
    {
      islong = true;
      continue;
    }
  else if(pvar[j] == '.' && !point)
    {
      point = true;
      continue;
    }

  if(pvar[j] == 'd')
    {
      if(islong)
   {
     long int *auxInt = va_arg(par_list,long int*);
     MPI_Pack(auxInt, 1, MPI_LONG, data, dataSize, &position,
         MPI_COMM_WORLD);
   }
      else
   {
     int *auxInt = va_arg(par_list,int*);
     MPI_Pack(auxInt, 1, MPI_INT, data, dataSize, &position,
         MPI_COMM_WORLD);
```

```
 }
    i++; //increment founded elements
    break;
  }
else if(pvar[j] == 'f' || pvar[j] == 'e' || pvar[j] == 'E')
  {
    if(islong)
 {
   double *auxDouble = va_arg(par_list,double*);
   MPI_Pack(auxDouble, 1, MPI_DOUBLE, data, dataSize, &
       position, MPI_COMM_WORLD);
 }
    else
 {
   float *auxFloat = va_arg(par_list,float*);
   MPI_Pack(auxFloat, 1, MPI_FLOAT, data, dataSize, &position,
        MPI_COMM_WORLD);
 }
    i++; //increment founded elements
    break;
  }
else if(pvar[j] == 'c')
  {
    char *auxChar = va_arg(par_list,char*);

    //Check if is a string or a char
    int auxLen = strlen(auxChar);
    if(auxLen > 1)
 {
   //is a string
   strings_length[nStrings] = auxLen; //Save string length
   MPI_Pack(auxChar, strings_length[nStrings], MPI_CHAR, data,
        dataSize, &position, MPI_COMM_WORLD);
   nStrings++; //increment number of strings
   i++; //increment founded elements
   break;
 }
    else
 {
   //is a char
   MPI_Pack(auxChar, 1, MPI_CHAR, data, dataSize, &position,
       MPI_COMM_WORLD);
   i++; //increment founded elements
```

```
      break;
    }
     }
   else if(pvar[j] == 's' || pvar[j] == '[')
     {
       char *auxChar = va_arg(par_list,char*);
       strings_length[nStrings] = strlen(auxChar); //Save string
           length
       MPI_Pack(auxChar, strings_length[nStrings], MPI_CHAR, data
           , dataSize, &position, MPI_COMM_WORLD);
       nStrings++; //increment number of strings
       i++; //increment founded elements
       break;
     }
   else if(pvar[j] != '0' && pvar[j] != '1' && pvar[j] != '2' &&
        pvar[j] != '3' && pvar[j] != '4' && pvar[j] != '5' &&
       pvar[j] != '6' && pvar[j] != '7' && pvar[j] != '8' && pvar
       [j] != '9' && pvar[j] != ' ')
     {
       //no variable type identificator or number
       printf("\nError in mpi_fscanf: invalid format '%c' reading
            element %d\n", pvar[j], nElements);
       exit(1000);
     }

   if(islong)
     {
       //no variable type identificator after 'l'
       printf("\nError in mpi_fscanf: invalid variable type ID\n"
           );
       exit(1000);
     }
 }
    cpy_formatlen = len;
    memmove(format_copy, pvar+1, len); //copy end of array
  }
if(i != nElements)
  {
    printf("\nError in mpi_fscanf: Diferent number of readed and
         sended elements \n");
    printf("    Readed: %d\n",nElements);
    printf("    Sended: %d\n",i);
    exit(1000);
```

```
     }
 }
   //add the final data possition to strings_length array
   strings_length[nStrings] = position;
   //add the return value of vfscanf
   strings_length[nStrings+1] = nElements;
  }
else
  {
   //Read format
   int cpy_formatlen = format_len;
   while(1)
 {
  //Find next %
  char *pvar = strstr(format_copy,"%");
  if(pvar == NULL)
    {
      break;
    }

  int len = cpy_formatlen-(pvar-format_copy);
  bool islong = false;
  bool point = false;
  char num[20];
  int numcont = 0;
  for(int j = 1; j < len; j++) //j=0 is '%'
    {
      if(pvar[j] == '*'){break;}
      if(pvar[j] == 'l' && !islong)
   {
     islong = true;
     continue;
   }

      if(pvar[j] == '.' && !point)
   {
     point = true;
     continue;
   }

      if(pvar[j] == 'd')
   {
     if(islong)
```

```
      {
        processed_format[nElements] = 'I';
        position += sizeof(long int);
      }
    else
      {
        processed_format[nElements] = 'i';
        position += sizeof(int);
      }
    nElements++;
    break;
  }
     else if(pvar[j] == 'f' || pvar[j] == 'e' || pvar[j] == 'E')
  {
    if(islong)
      {
        processed_format[nElements] = 'd';
        position += sizeof(double);
      }
    else
      {
        processed_format[nElements] = 'f';
        position += sizeof(float);
      }
    nElements++;
    break;
  }
     else if(pvar[j] == 'c')
  {
    //Check if is a string or a char
    double auxNum = -1.0;
    if(numcont > 0){num[numcont] = '\0'; auxNum = atof(num);}
    else{auxNum = 1.0;}

    if(auxNum >=2.0)
      {
        //is a string
        processed_format[nElements] = 's';
        nStrings++;
        nElements++;
        break;
      }
    else
```

```
       {

         //is a char
         processed_format[nElements] = 'c';
         nElements++;
         break;
       }
    }
      else if(pvar[j] == 's' || pvar[j] == '[')
    {
     processed_format[nElements] = 's';
     nStrings++;
     nElements++;
     break;
    }
      else if(pvar[j] != '0' && pvar[j] != '1' && pvar[j] != '2'
          && pvar[j] != '3' && pvar[j] != '4' && pvar[j] != '5' &&
          pvar[j] != '6' && pvar[j] != '7' && pvar[j] != '8' &&
          pvar[j] != '9' && pvar[j] != ' ')
    {
     //no variable type identificator or number
     printf("\nError in mpi_fscanf: invalid format '%c' reading
         element %d\n", pvar[j], nElements);
     exit(1000);
    }

      num[numcont] = pvar[j];
      numcont++;

      if(islong)
    {
     //no variable type identificator after 'l'
     printf("\nError in mpi_fscanf: invalid variable type ID\n");
     exit(1000);
    }
     }
   cpy_formatlen = len;
   memmove(format_copy, pvar+1, len); //copy end of array
 }
  }


//Bcast string lengths
```

```
//printf("\n Send/Recive %d with %d strings of '%s', %d\n",
    nElements, nStrings, format, rank);
MPI_Bcast(strings_length, nStrings+2, MPI_INT, 0, MPI_COMM_WORLD);
//printf("\n Sended/Recived %d with %d strings of '%s', %d\n",
    nElements, nStrings, format, rank);

//add string lenth to total data size
if(rank != 0)
  {
    for(int i = 0; i < nStrings; i++)
 {
   position += strings_length[i];
 }
    //Check total data size
    if(strings_length[nStrings] != position)
 {
   printf("\nError in mpi_fscanf: data size of root proces (0) does
        not match with proces %d\n", rank);
   printf("    root: %d\n",strings_length[nStrings]);
   printf("%9d: %d\n",rank,position);
   exit(1000);
 }
    else if(strings_length[nStrings+1] != nElements)
 {
   printf("\nError in mpi_fscanf: element number of root proces (0)
        does not match with proces %d\n", rank);
   exit(1000);
 }
  }

//Bcast package
if(nElements > 0)
  {
    MPI_Bcast(data, position, MPI_PACKED, 0, MPI_COMM_WORLD);

    if(rank != 0)
 {
   int extractPos = 0;
   int stringCount = 0;
   for(int i = 0; i < nElements; i++)
     {
       if(processed_format[i] == 'I') //long int
    {
```

```
     long int *auxInt = va_arg(par_list,long int*);
     MPI_Unpack(data, dataSize, &extractPos, auxInt, 1, MPI_LONG,
         MPI_COMM_WORLD);
   }
      else if(processed_format[i] == 'i') //int
   {
     int *auxInt = va_arg(par_list,int*);
     MPI_Unpack(data, dataSize, &extractPos, auxInt, 1, MPI_INT,
         MPI_COMM_WORLD);
   }
      else if(processed_format[i] == 'f') //float
   {
     float *auxFloat = va_arg(par_list,float*);
     MPI_Unpack(data, dataSize, &extractPos, auxFloat, 1,
         MPI_FLOAT, MPI_COMM_WORLD);
   }
      else if(processed_format[i] == 'd') //double
   {
     double *auxDouble = va_arg(par_list,double*);
     MPI_Unpack(data, dataSize, &extractPos, auxDouble, 1,
         MPI_DOUBLE, MPI_COMM_WORLD);
   }
      else if(processed_format[i] == 'c') //char
   {
     char *auxChar = va_arg(par_list,char*);
     MPI_Unpack(data, dataSize, &extractPos, auxChar, 1, MPI_CHAR,
          MPI_COMM_WORLD);
   }
      else if(processed_format[i] == 's') // c string
   {
     char *auxChar = va_arg(par_list,char*);
     MPI_Unpack(data, dataSize, &extractPos, auxChar,
         strings_length[stringCount], MPI_CHAR, MPI_COMM_WORLD);
     //add end of string character
     auxChar[strings_length[stringCount]] = '\0';
     stringCount++;
   }
    }
 }
  }
va_end(par_list);

return nElements;
```

```
}
int PENELOPE_MPI::mpi_printf(const char* format, ...)
{
  va_list par_list;
  va_start(par_list, format);

  static FILE* output = NULL;
  int return_printf = 0;

  if(output == NULL)
    {
      //Take proces rank
      int rank;
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);

      char output_name[40];
      sprintf(output_name,"log_%d", rank);
      output = freopen(output_name, "w", stdout);

      if(output == NULL)
   {
     printf("\nError in mpi_printf: proces %d can't open output file
         'log_%d'\n", rank, rank);
     exit(1000);
   }

    }

  return_printf = vfprintf(output,format,par_list);
  fflush(output);

  va_end(par_list);

  return return_printf;
}
char* PENELOPE_MPI::mpi_fgets(char *str, int n, FILE *stream)
{
  //Take proces rank
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  char* return_fgets = NULL;
  char send_recive = 'a';
```

```
  if(rank == 0)
    {
      return_fgets = fgets(str, n, stream);
      send_recive = (return_fgets == NULL ? '0' : '1');
    }

  //send returned value
  MPI_Bcast(&send_recive, 1, MPI_CHAR, 0, MPI_COMM_WORLD);

  if(send_recive != '0')
    {
      //send read data
      MPI_Bcast(str, n, MPI_CHAR, 0, MPI_COMM_WORLD);
      return str;
    }
  else
    {
      return NULL;
    }
}
FILE* PENELOPE_MPI::mpi_fopen(const char *filename, const char *mode
    )
{
  //Take proces rank
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  //check if it's a input file
  if(strstr(mode,"r") != NULL)
    {
      FILE* auxFile = NULL;
      char send = 'a';
      if(rank == 0)
   {
     auxFile = fopen(filename, mode);
     send = auxFile == NULL ? '0' : '1';
   }

      //Bcast if file was opened successfully.
      MPI_Bcast(&send, 1, MPI_CHAR, 0, MPI_COMM_WORLD);


      if(rank != 0)
   {
```

```
     if(send == ’0’) //File not opened in process 0
       {
         return NULL;
       }
     else //file opened successfully in root process
       {
         //Create and open a dummy local file.
         char filename2[300];
         sprintf(filename2, "dummy_rank%d_%s",rank,filename);
         auxFile = fopen(filename2,"w");
         fclose(auxFile);
         return fopen(filename2,mode);
       }
   }
      else
   {
     return auxFile;
   }
    }
  else
    {
      if(rank == 0)
   {
     return fopen(filename, mode);
   }
      else
   {
     char filename2[300];
     sprintf(filename2, "mpi_rank%d_%s",rank,filename);
     return fopen(filename2, mode);
   }
    }
}
int PENELOPE_MPI::mpi_fclose(FILE* file)
{
  if(file != NULL)
    {
      return fclose(file);
    }
  return 0;
}
void PENELOPE_MPI::mpi_balance()
{
```

```
double previousTime = realtime();
double previousnHist = 0.0;
//Take proces rank
int rank;
int p;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

int nUpdates = 0; //count number of updates

if(rank == 0)
  {
    double* velocity = new double[p]; //Store velocity of each
        process
    MPI_Status status;

    while(nUpdates < communicationMPI::maxUpdates)
 {
   sleep(communicationMPI::updateDelay);
   nUpdates++;
   char req = ’0’;
   if(ctrsimMod::finishedSim)
     {
       req = ’-’; //End of simulation
       //Bcast end of communication.
       MPI_Bcast(&req, 1, MPI_CHAR, 0, MPI_COMM_WORLD);

       printf("    History bucle has been finished .\n Stopping
           simulation communications.\n");
       delete [] velocity;
       return;
     }

   //Bcast a request simulation status.
   MPI_Bcast(&req, 1, MPI_CHAR, 0, MPI_COMM_WORLD);

   if(req == ’0’)
     {
       //Update balance request
       printf("\n *** Recalculate history number ***\n");
       double simulatedHist = 0.0;
       double totalVelocity = 0.0;
       for(int i = 1; i < p; i++)
```

```
{
  double recivBuffer[3];
  //Recive number of simulated showers and elapsed time
  // of each process
  MPI_Recv(recivBuffer, 3, MPI_DOUBLE,
      i, _MPI_COMM_TAG_, MPI_COMM_WORLD, &status);

  simulatedHist += recivBuffer[0];
  //Calculate velocity
  velocity[i] = recivBuffer[1]/recivBuffer[2];
  totalVelocity += velocity[i];
}

   double actualnHist0 = ctrsimMod::nhist;
   double actualTime0 = realtime();
   simulatedHist += actualnHist0; //Simulated showers
   velocity[0] = (actualnHist0-previousnHist)/(actualTime0-
       previousTime);
   totalVelocity += velocity[0];

   double remainingHist = ctrsimMod::nhistmaxtotal-
       simulatedHist; //Remaining showers

   printf("    Remaining histories:\n    %17.5e\n",
       remainingHist);
   printf("    Combined speed (hist/s):\n   %17.5e\n",
       totalVelocity);
   printf("    Relative speeds: \n");

   //Calculate relative velocity
   for(int i = 0; i < p; i++)
{
  velocity[i] /= totalVelocity;
  printf("     P%04d -> %5.2f%%\n",i,velocity[i]*100.0);
}

   //Calculate new histories assign
   printf("    New histories assign: \n");
   ctrsimMod::nhistmax = velocity[0]*remainingHist+actualnHist0
       ;
   printf("     P%04d -> %15.4e\n",0,ctrsimMod::nhistmax);
   for(int i = 1; i < p; i++)
{
```

```
     double sendData[2] = {velocity[i]*remainingHist,simulatedHist
         };

     printf("     P%04d -> %15.4e\n",i,sendData[0]);
     //Send new assign to process
     MPI_Send(sendData, 2, MPI_DOUBLE, i,
         _MPI_COMM_TAG_, MPI_COMM_WORLD);
   }

      if(simulatedHist >= communicationMPI::historyLimit)
   {
     PENELOPE_MPI::mpi_printf("\n  History limit for updates
         reached. Stopping communications\n");
     delete [] velocity;
     return;
   }

      previousnHist = actualnHist0;
      previousTime = actualTime0;


    }

 }
  }
else
  {
    while(nUpdates < communicationMPI::maxUpdates)
 {
   sleep(communicationMPI::updateDelay);
   nUpdates++;
   char req;
   //Recive a request simulation status missage.
   MPI_Bcast(&req, 1, MPI_CHAR, 0, MPI_COMM_WORLD);

   if(req == '0') //Balance charge
    {
      PENELOPE_MPI::mpi_printf("*** Recived a rebalance request
          ***\n");
      double actualTime = realtime();
      double actualnHist = ctrsimMod::nhist;
      double sendData[3] = {actualnHist,actualnHist-previousnHist,
           actualTime-previousTime};
```

```
    if(ctrsimMod::finishedSim)
 {
   //This process has been finished history bucle. His actual
       velocity must be 0 hist/s
   sendData[1] = 0.0;
   PENELOPE_MPI::mpi_printf("  History bucle completed.\n
       Sending velocity of 0 hist/s.\n");
 }
    else
 {
   PENELOPE_MPI::mpi_printf("  %17.0f. simulated showers in
       %16.5e minutes\n", actualnHist-previousnHist, (actualTime-
       previousTime)/60.0);
 }
    previousTime = actualTime;
    previousnHist = actualnHist;


    //Send to rank 0 the number of simulated histories
    MPI_Send(&sendData, 3, MPI_DOUBLE, 0,
        _MPI_COMM_TAG_, MPI_COMM_WORLD);

    //Wait the response
    double recvData[2];
    MPI_Status status;
    MPI_Recv(recvData, 2, MPI_DOUBLE,
        0, _MPI_COMM_TAG_, MPI_COMM_WORLD, &status);

    ctrsimMod::nhistmax = recvData[0]+actualnHist;

    PENELOPE_MPI::mpi_printf("  New number of histories to
        simulate:\n  %17.0f. \n", ctrsimMod::nhistmax);

    if(recvData[1] >= communicationMPI::historyLimit)
 {
   PENELOPE_MPI::mpi_printf("\n  History limit for updates
       reached. Stopping communications\n");
   return;
 }

  }
else if(req == '-')
```

```
      {
        //End of communications
        PENELOPE_MPI::mpi_printf("  Process 0 has completed history
            bucle.\n Stopping simulation communications.\n");
        return;
      }
  }

   }

  PENELOPE_MPI::mpi_printf(" *** Maximum number of updates reached.
      Stopping communications.\n");

  return;
}

void PENELOPE_MPI::mpi_splitPhaseSpaceFile(char * psfname)
{

  using namespace formatVer;

  MPI_Status status;
  int rank;
  int p;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  char filename[100];
  sprintf(filename,"mpi_spf_%s_rank_%d", psfname, rank);

  //Check existence of pre-splited psf
  char exist = 0;

  FILE* check = fopen(filename,"rb");

  if(check != NULL)
    {
      exist = 1;
      fclose(check);
    }

  char globalCheckChar = 0;
```

```
if(rank == 0)
  {
    int globalCheck = (int)exist;
    //Recive file existence check from all processes
    for(int i = 1; i < p; i++)
 {
   char auxChar;
   MPI_Recv(&auxChar, 1, MPI_CHAR, i,
       _MPI_COMM_TAG_, MPI_COMM_WORLD, &status);

   globalCheck += (int)auxChar;
 }

    if(globalCheck == p)
 {
   //All processes have their sub psf
   globalCheckChar = 1;
 }

  }
else
  {
    MPI_Send(&exist, 1, MPI_CHAR, 0,
        _MPI_COMM_TAG_, MPI_COMM_WORLD);
  }

//Bcast global check
MPI_Bcast(&globalCheckChar, 1, MPI_CHAR, 0, MPI_COMM_WORLD);

if((int)globalCheckChar == 1)
  {
    printf("Reusing previous psf split.\n");
    //Rename psf
    strcpy(psfname,filename);
    return;
  }
else
  {
    printf("No valid previous psf split found.\n");
  }

double nPart = 0.0;
```

```
if(rank == 0)
  {
    FILE* ufile = fopen(psfname,"r");
    FILE* newSPF = fopen(filename, "wb");

    if(newSPF == NULL)
 {
   printf("mpi_splitSpaceFile:ERROR: Can't create file %s\n",
       filename);
   exit(1000);
 }

    char  buffer[256];
    char* error;
    int nextProcess = 0;
    bool EndOfFile = false;
    int line = 0;

    while(!EndOfFile)
 {
   //Read next phase space file particle
   while(1)
     {
       error = fgets(buffer, 256, ufile);
       if(strstr(buffer,"\n") == 0)
     {
       fscanf(ufile,"%*[^\n]");
       fgetc(ufile);
     }

       if(feof(ufile)) //End of file
     {
       EndOfFile = true;
       break;
     }

       buffer[strlen(buffer)-1]='\0';
       line++;
       if(error == NULL)
     {
       printf("mpi_splitPhaseSpaceFile:ERROR: unable to read PSF
           line no.:\n");
       printf("%-d\n",line);
```

```
   exit(292);
 }
    if(buffer[0] != '#'){ break;} // A non-comment line was
        found
  }

if(EndOfFile) // End of file, finish file split
  {
    int intData[7];
    double doubleData[8];

    intData[0] = -1; //kpar never will be < 0
    //Pack data
    char data[1000]; //Stores packaged data
    int dataSize = 7*sizeof(int)+8*sizeof(double); //max data
        size
    int position = 0;
    MPI_Pack(doubleData, 8, MPI_DOUBLE, data, dataSize, &
        position, MPI_COMM_WORLD);
    MPI_Pack(intData , 7, MPI_INT , data, dataSize, &position,
        MPI_COMM_WORLD);

    for(int i = 1; i < p; i++) //Send "finish" to all processes
 {
   MPI_Send(data, dataSize, MPI_PACKED, i,
       _MPI_COMM_TAG_, MPI_COMM_WORLD);
 }
    break;
  }

int intData[7];
double doubleData[8];


int Numread=sscanf(buffer,"%d %lf %lf %lf %lf %lf %lf %lf %lf %d
     %d %d %d %d %d",
         &intData[0],&doubleData[0],&doubleData[1],&doubleData
             [2],&doubleData[3],&doubleData[4],
         &doubleData[5],&doubleData[6],&doubleData[7],&intData
             [1],&intData[2],&intData[3],
         &intData[4],&intData[5],&intData[6]);
if(Numread != 15)
  {
```

```
      printf("mpi_splitPhaseSpaceFile:ERROR: Wrong line format,
          specting 15 elements\n");
      printf("                           found instead %d\n",
          Numread);
      exit(1000);
    }

  if(nextProcess == 0)
    {
      //write data
      fwrite(intData , sizeof(int)  , 7, newSPF);
      fwrite(doubleData, sizeof(double), 8, newSPF);
      nPart += 1.0;
    }
  else
    {
      //Pack data
      char data[1000]; //Stores packaged data
      int dataSize = 7*sizeof(int)+8*sizeof(double); //max data
          size
      int position = 0;
      MPI_Pack(doubleData, 8, MPI_DOUBLE, data, dataSize, &
          position, MPI_COMM_WORLD);
      MPI_Pack(intData , 7, MPI_INT  , data, dataSize, &position,
          MPI_COMM_WORLD);

      //Send particle information
      MPI_Send(data, dataSize, MPI_PACKED, nextProcess,
          _MPI_COMM_TAG_, MPI_COMM_WORLD);
    }

  nextProcess++;
  if(nextProcess >= p){nextProcess = 0;}
 }

   fclose(ufile);
   fclose(newSPF);

  }
else
  {

   //Create new spf
```

```
   FILE* newSPF = fopen(filename, "wb");

   if(newSPF == NULL)
{
  printf("mpi_splitSpaceFile:ERROR: Can't create file %s\n",
      filename);
  exit(1000);
}


   while(1)
{

  //Recive particle information

  char data[1000]; //Stores packaged data
  int dataSize = 7*sizeof(int)+8*sizeof(double); //max data size
  MPI_Recv(data, dataSize, MPI_PACKED, 0,
      _MPI_COMM_TAG_, MPI_COMM_WORLD, &status);


  //unpack data
  int intData[7];
  double doubleData[8];
  int extractPos = 0;

  MPI_Unpack(data, dataSize, &extractPos, doubleData, 8,
      MPI_DOUBLE, MPI_COMM_WORLD);
  MPI_Unpack(data, dataSize, &extractPos, intData , 7, MPI_INT ,
      MPI_COMM_WORLD);

  if(intData[0] < 0)
    {
      //End of file reached
      break;
    }

  //write data
  fwrite(intData , sizeof(int)  , 7, newSPF);
  fwrite(doubleData, sizeof(double), 8, newSPF);
  nPart += 1.0;
}
```

```
      fclose(newSPF);

    }

  PENELOPE_MPI::mpi_printf(" psf splited, number of local particules
      :\n %17.0\n", nPart);

  //Rename psf
  strcpy(psfname,filename);
  return;
}

bool PENELOPE_MPI::mpi_getpar()
{

//
    *******************************************************************

//*   Reads a new particle from the PSF.                        *
//*                                                             *
//*   Output:                                                   *
//*     -> returns .false. if EOF has been reached, else .true. *
//*     -> particle state in /srcpsf/                           *
//
    *******************************************************************

  using namespace srcpsf;
  using namespace srcps1;
  using namespace srcps2;
  using namespace formatVer;

  bool  Return_getpar;

  if(PENELOPE_MPI::nStackPars == 0)
    {
      for(PENELOPE_MPI::nStackPars = 0; PENELOPE_MPI::nStackPars <
          _MPI_GETPAR_STACK_SIZE_; PENELOPE_MPI::nStackPars++)
   {
     //read data
     fread(PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars].
         intData , sizeof(int) , 7, ufile);
     fread(PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars].
         doubleData, sizeof(double), 8, ufile);
```

```
    if(feof(ufile))
      {
        break;
      }
  }
   }

//check end of file
if(PENELOPE_MPI::nStackPars == 0)
  {
    Return_getpar=false;
    return Return_getpar;
  }


nline=nline+1;            // Lines read
npart = npart+1;                 // Particles read

kpars = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
    intData[0];
es    = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
    doubleData[0];
xs    = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
    doubleData[1];
ys    = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
    doubleData[2];
zs    = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
    doubleData[3];
us    = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
    doubleData[4];
vs    = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
    doubleData[5];
ws    = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
    doubleData[6];
wghts = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
    doubleData[7];

if(format2008)
  {
    dns     = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
        intData[1];
    ilbs[0] = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars
        -1].intData[2];
```

```
      ilbs[1] = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars
          -1].intData[3];
      ilbs[2] = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars
          -1].intData[4];
      ilbs[3] = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars
          -1].intData[5];
      ilbs[4] = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars
          -1].intData[6];
    }
  else
    {
      dns    = PENELOPE_MPI::getParStack[PENELOPE_MPI::nStackPars-1].
          intData[6];
      ilbs[0] = 1;            // Assume all particles are primaries
      ilbs[1] = 0;
      ilbs[2] = 0;
      ilbs[3] = 0;
    }

  PENELOPE_MPI::nStackPars--;
  Return_getpar = true;
  return Return_getpar;

}

FILE* PENELOPE_MPI::mpi_fopenLocal(const char *filename, const char
    *opt)
{
  return fopen(filename,opt);
}

//Define new scanf

//To avoid warnings, we must check if the number of scanf arguments
    are 1 ore more

//#define scanf_1(format) PENELOPE_MPI::scanf(format)
//#define fscanf_1(file, format) PENELOPE_MPI::fscanf(file, format)

//#define scanf_More(format, ...) PENELOPE_MPI::scanf(format,
    __VA_ARGS__)
//#define fscanf_More(file, format, ...) PENELOPE_MPI::fscanf(file,
    format, __VA_ARGS__)
```

```
//#define SELECT_SCANF(format, CHECK, ...) (CHECK == 1 ? scanf_1 :
    scanf_More )
//#define scanf(...) SELECT_SCANF(__VA_ARGS__, 1, "dummy")

//#define SELECT_FSCANF(file, format,CHECK, ...) ( CHECK == 1 ?
    fscanf_1 : fscanf_More )
//#define fscanf(...) SELECT_FSCANF(__VA_ARGS__, fscanf, "dummy") (
    __VA_ARGS__)


// ALTRA HISTORIA

//#define scanf_Wrapper(format, ...) PENELOPE_MPI::scanf(format,##
    __VA_ARGS__)
//#define fscanf_Wrapper(file, format, ...) PENELOPE_MPI::fscanf(
    file, format,##__VA_ARGS__)

//#define VA_ARGS(...) , ##__VA_ARGS__

//#define scanf(format, ...) scanf(format VA_ARGS(__VA_ARGS__))
//#define fscanf(file, format, ...) fscanf(file, format VA_ARGS(
    __VA_ARGS__))

#define scanf(...) PENELOPE_MPI::mpi_scanf(__VA_ARGS__)
#define fscanf(...) PENELOPE_MPI::mpi_fscanf(__VA_ARGS__)
#define printf(...) PENELOPE_MPI::mpi_printf(__VA_ARGS__)
#define fgets(str, n, stream) PENELOPE_MPI::mpi_fgets(str, n, stream
    )
#define fopen(filename, mode) PENELOPE_MPI::mpi_fopen(filename, mode
    )
#define fclose(file) PENELOPE_MPI::mpi_fclose(file)
#define getpar() PENELOPE_MPI::mpi_getpar()

#endif
#endif
```

C Apèndix: JUMP per a electrons en CUDA

Mostra del codi emprat per a llançar la rutina JUMP d'electrons a la GPU utilitzant CUDA.

```
#define CUDA_SAFE_CALL( call ) {                                  \
   cudaError_t err = call;                                          \
   if( cudaSuccess != err ) {                                       \
      fprintf(stderr,"CUDA: %s\n", cudaGetErrorString(err));
          \
      fprintf(stderr,"CUDA: error occurred in cuda routine. Exiting
          ...\n"); \
      exit(err);
          \
} }
```

Codi en el “host”, al final s’ha obviat codi comú per a la versió amb i sense CUDA.

```

//
    *********************************************************************

//                 SUBROUTINE JUMP FOR ELECTRONS
//
    *********************************************************************

void JUMPelectron(double *DSMAX, double *DS)
{
//  Calculation of the free path from the starting point to the
    position
//  of the next event and of the probabilities of occurrence of
    different
//  events.

//  Arguments:
//    DSMAX .... maximum allowed step length (input),
//    DS ....... segment length (output).

//  Output, through module PENELOPE_mod:
//    E0STEP ... energy at the beginning of the segment,
//    DESOFT ... energy loss due to soft interactions along the step,
//    SSOFT .... stopping power due to soft interactions,
//               = DESOFT/step_length.

using namespace nmspace_PENELOPE_mod;
using namespace nmspace_TRACK_mod;

```

```
using namespace nmspace_CEGRID;
using namespace nmspace_CEIMFP;
using namespace nmspace_CPIMFP;
using namespace nmspace_CJUMP0;
using namespace nmspace_CJUMP1;
using namespace nmspace_JUMPelectrons_Device;

//  ************ Electrons (KPAR=0).

// Copy host memory to device

int GPU_electrons = 0; //save the number of electrons send to GPU

for(int i = 0; i < TRACKNParticles[0]; i++)
{
   if( (CJUMP1[0][i].MHINGE == 1 && TRACKParam[0][i].E < CJUMP1[0][i
       ].ELAST1) || TRACKParam[0][i].E < CJUMP1[0][i].ELAST2 )
   {
      h_E[GPU_electrons] = TRACKParam[0][i].E;
      h_MAT[GPU_electrons] = TRACKBody[0][i].MAT;
      GPU_electrons++;
   }
}


int nStreams = GPU_electrons/streamSize;
int mem_size_double = sizeof(double)*streamSize;
int mem_size_int = sizeof(int)*streamSize;

int offset = GPU_electrons-streamSize*nStreams;
int mem_double_offset = sizeof(double)*offset;
int mem_int_offset = sizeof(int)*offset;


int nblocks = (streamSize/blockSize)+1;



//send cuda streams
//CUDA_USAGE();
for(int i = 0; i < nStreams; i++)
{
cudaStreamCreate(&stream[i]);
```

```
int init_pos = i*streamSize;
cudaMemcpyAsync( &d_E[init_pos], &h_E[init_pos], mem_size_double,
    cudaMemcpyHostToDevice, stream[i] );
cudaMemcpyAsync( &d_MAT[init_pos], &h_MAT[init_pos], mem_size_int,
    cudaMemcpyHostToDevice, stream[i] );

JUMPelectrons_kernel<<< nblocks, blockSize, 0, stream[i] >>>(
    streamSize, d_XEL, d_E, d_XE, d_KE, d_XEK, d_P1, d_P2, d_P3, d_P4
    , d_P7, d_W1, d_W2, d_T1, d_T2, DLEMP1, DLFC, d_SEHEL, d_SEHIN,
    d_SEHBR, d_SEISI, d_W1E, d_W2E, d_T1E, d_T2E, NEGP, d_MAT,
    init_pos);

cudaMemcpyAsync( &h_XEL[init_pos], &d_XEL[init_pos], mem_size_double
    , cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_XE[init_pos], &d_XE[init_pos], mem_size_double,
    cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_XEK[init_pos], &d_XEK[init_pos], mem_size_double
    , cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_P1[init_pos], &d_P1[init_pos], mem_size_double,
    cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_P2[init_pos], &d_P2[init_pos], mem_size_double,
    cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_P3[init_pos], &d_P3[init_pos], mem_size_double,
    cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_P4[init_pos], &d_P4[init_pos], mem_size_double,
    cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_P7[init_pos], &d_P7[init_pos], mem_size_double,
    cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_W1[init_pos], &d_W1[init_pos], mem_size_double,
    cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_W2[init_pos], &d_W2[init_pos], mem_size_double,
    cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_T1[init_pos], &d_T1[init_pos], mem_size_double,
    cudaMemcpyDeviceToHost, stream[i] );
cudaMemcpyAsync( &h_T2[init_pos], &d_T2[init_pos], mem_size_double,
    cudaMemcpyDeviceToHost, stream[i] );

cudaMemcpyAsync( &h_KE[init_pos], &d_KE[init_pos], mem_size_int,
    cudaMemcpyDeviceToHost, stream[i] );
}
//send offset stream
if(offset > 0)
{
```

```
cudaStreamCreate(&offset_stream);

int init_pos = nStreams*streamSize;
cudaMemcpyAsync( &d_E[init_pos], &h_E[init_pos], mem_double_offset,
    cudaMemcpyHostToDevice, offset_stream );
cudaMemcpyAsync( &d_MAT[init_pos], &h_MAT[init_pos], mem_int_offset,
     cudaMemcpyHostToDevice, offset_stream );

JUMPelectrons_kernel<<< nblocks, blockSize, 0, offset_stream >>>(
    offset, d_XEL, d_E, d_XE, d_KE, d_XEK, d_P1, d_P2, d_P3, d_P4,
    d_P7, d_W1, d_W2, d_T1, d_T2, DLEMP1, DLFC, d_SEHEL, d_SEHIN,
    d_SEHBR, d_SEISI, d_W1E, d_W2E, d_T1E, d_T2E, NEGP, d_MAT,
    init_pos);



cudaMemcpyAsync( &h_XEL[init_pos], &d_XEL[init_pos],
    mem_double_offset, cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_XE[init_pos], &d_XE[init_pos], mem_double_offset
    , cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_XEK[init_pos], &d_XEK[init_pos],
    mem_double_offset, cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_P1[init_pos], &d_P1[init_pos], mem_double_offset
    , cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_P2[init_pos], &d_P2[init_pos], mem_double_offset
    , cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_P3[init_pos], &d_P3[init_pos], mem_double_offset
    , cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_P4[init_pos], &d_P4[init_pos], mem_double_offset
    , cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_P7[init_pos], &d_P7[init_pos], mem_double_offset
    , cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_W1[init_pos], &d_W1[init_pos], mem_double_offset
    , cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_W2[init_pos], &d_W2[init_pos], mem_double_offset
    , cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_T1[init_pos], &d_T1[init_pos], mem_double_offset
    , cudaMemcpyDeviceToHost, offset_stream );
cudaMemcpyAsync( &h_T2[init_pos], &d_T2[init_pos], mem_double_offset
    , cudaMemcpyDeviceToHost, offset_stream );

cudaMemcpyAsync( &h_KE[init_pos], &d_KE[init_pos], mem_int_offset,
    cudaMemcpyDeviceToHost, offset_stream );
```

```
//remove CUDA stream
cudaStreamDestroy(offset_stream);
}

//remove CUDA streams
for(int i=0; i < nStreams; i++)
{
cudaStreamDestroy(stream[i]);
}

//Synchronize
cudaDeviceSynchronize();

//Codi igual a la versió sense CUDA obviat~
.
.
.
//~Final del codi obviat

return;
}
```

“Kernel” executat a la GPU.

```
__global__ void JUMPelectrons_kernel( int part_number, double *d_XEL
    , double *d_E, double *d_XE, int *d_KE, double *d_XEK, double *
    d_P1, double *d_P2, double *d_P3, double *d_P4, double *d_P7,
    double *d_W1, double *d_W2, double *d_T1, double *d_T2, double
    d_DLEMP1, double d_DLFC, double *d_SEHEL, double *d_SEHIN, double
     *d_SEHBR, double *d_SEISI, double *d_W1E, double *d_W2E, double
    *d_T1E, double *d_T2E, int d_NEGP, int *d_MAT, int init_pos) {

   /* Obtain the global matrix index accessed by the thread executing
        this kernel */
   int i = blockIdx.x * blockDim.x + threadIdx.x; //Row index


   if(i<part_number)
   {
      int position = init_pos+i;
      d_XEL[position] = log(d_E[position]);
      d_XE[position] = 1.0 + (d_XEL[position] - d_DLEMP1)*d_DLFC;
      d_KE[position] = (int)d_XE[position];
      d_XEK[position] = d_XE[position] - (double)d_KE[position];
```

```
      //EIMFP//

      d_P1[position] = exp(d_SEHEL[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1]+(d_SEHEL[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]]-d_SEHEL[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1])*d_XEK[position]);
      d_P2[position] = exp(d_SEHIN[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1]+(d_SEHIN[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]]-d_SEHIN[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1])*d_XEK[position]);
      d_P3[position] = exp(d_SEHBR[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1]+(d_SEHBR[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]]-d_SEHBR[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1])*d_XEK[position]);
      d_P4[position] = exp(d_SEISI[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1]+(d_SEISI[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]]-d_SEISI[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1])*d_XEK[position]);
      d_P7[position] = 0.0;

      d_W1[position] = exp(d_W1E[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1]+(d_W1E[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]]-d_W1E[(d_MAT[position]-1)*d_NEGP + d_KE[position
          ]-1])*d_XEK[position]);
      d_W2[position] = exp(d_W2E[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1]+(d_W2E[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]]-d_W2E[(d_MAT[position]-1)*d_NEGP + d_KE[position
          ]-1])*d_XEK[position]);

      d_T1[position] = exp(d_T1E[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1]+(d_T1E[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]]-d_T1E[(d_MAT[position]-1)*d_NEGP + d_KE[position
          ]-1])*d_XEK[position]);
      d_T2[position] = exp(d_T2E[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]-1]+(d_T2E[(d_MAT[position]-1)*d_NEGP + d_KE[
          position]]-d_T2E[(d_MAT[position]-1)*d_NEGP + d_KE[position
          ]-1])*d_XEK[position]);

   }

}
```

D Apèndix: post-processat

```
void IDCreport(int mode, double n, double cputim, int &uncdone)
{
//
   *******************************************************************

//*    Input:                                                  *
//*      mode:   -1 if called at end-of-simulation, 0 for dump only,*
//*              1 for tally report only, 2 for both.          *
//*      n:   no. of histories simulated                       *
//*      cputim:  elapsed CPU time                             *
//*    Output:                                                 *
//*      uncdone:  1 if not actived, 0 else                    *
//*                not computed if mode=0                      *
//
   *******************************************************************

  using namespace IDCmod;

  FILE* out;
  const unsigned int rebin = 2; //new voxel size (rebin x rebin)
  const double rebin2 = pow(rebin,2);
  const double invRebin2 = 1.0/rebin2;

  uncdone = 1;
  if(!active){ return;}
  uncdone = 0;

  if(mode != -1){ return;} //Only print on final report


  double total_nhist = n;
  double initProcessTime = realtime();

  out = PENELOPE_MPI::mpi_fopenLocal("tallyIsoDoseCurves.dat", "w");
  if(out == NULL)
  {
    printf(" *********************************************\n");
    printf(" IDCreport:ERROR: cannot open output data file\n");
    printf(" *********************************************\n");
    return;
```

```
  }

  //Create arrays for contour data
  double contourBinWidth = 0.2;
  int contourBins = (int)(percMax/contourBinWidth+1.0);
  int** contourDat = new int*[dicomPenelope::nContours];
  for(int i = 0; i < dicomPenelope::nContours; i++)
    {
      contourDat[i] = new int[contourBins+1];
      for(int j = 0; j < contourBins+1; j++)
   {
     contourDat[i][j] = 0;
   }
    }

  //Find max edep
  double avesig = 0.0;
  int nchan = 0;
  double tmpdbl=0.0;
  for(int unsigned kk=0; kk < geovoxMod::nvox; kk++)
    {
      if(VDDmod::edep[kk] > tmpdbl){tmpdbl=VDDmod::edep[kk];}
    }
  const double maxq = 0.5*tmpdbl;                       // 1/2 of
      the max score


  //Create directory to store output
  system("rm -r isoCurves &> /dev/null");
  system("mkdir isoCurves &> /dev/null");

  //Transform dose to total Gy
  const double invn = 1.0/total_nhist;
  const double ev2perCent = 100.0*evg2Gy*nDes/prescribDose;
  const double curvesGap = (percMax-percMin)/(double)nCurves;

#ifdef _USE_OMP_

  const char* s = getenv("OMP_NUM_THREADS");
  int defaultThreads = atoi(s);

  printf("\n\n Post-processing IDCreport with %d threads\n",
      defaultThreads);
```

```
  omp_set_num_threads(defaultThreads);

  int zvoxmin = VDDmod::zvoxmin;
  int zvoxmax = VDDmod::zvoxmax;

  int k=0;

#pragma omp parallel firstprivate(invn,ev2perCent,curvesGap,rebin,
    rebin2,invRebin2,maxq,contourBinWidth,contourBins) default(shared
    )
  {
    int numThread = omp_get_thread_num();

    int nContours = dicomPenelope::nContours;
    //Create local arrays for contour data
    int** localContourDat = new int*[nContours];
    int* contourMatID = new int[nContours];

    for(int i = 0; i < nContours; i++)
      {
   contourMatID[i] = dicomPenelope::contourMatID[i];
   localContourDat[i] = new int[contourBins+1];
   for(int j = 0; j < contourBins+1; j++)
     {
       localContourDat[i][j] = 0;
     }
      }

    int dicomVox[3] = {dicomPenelope::dicomNPixels[0],dicomPenelope::
        dicomNPixels[1],dicomPenelope::dicomNPixels[2]};
    double dimVox[3] = {dicomPenelope::DimVoxelDicom[0],dicomPenelope
        ::DimVoxelDicom[1],dicomPenelope::DimVoxelDicom[2]};

    int nx = geovoxMod::nx;
    int ny = geovoxMod::ny;
    int nxy = geovoxMod::nxy;

    int xvoxmin = VDDmod::xvoxmin;
    int yvoxmin = VDDmod::yvoxmin;
    int xvoxmax = VDDmod::xvoxmax;
    int yvoxmax = VDDmod::yvoxmax;
```

```
#endif

#pragma omp for
    for(k=zvoxmin; k <= zvoxmax; k++)
      {
   FILE *gnuplotPipe = popen("gnuplot -persist", "w"); // Create
       gnuplot pipe
   if(gnuplotPipe == NULL)
     {
      printf("IDCreport:ERROR: cant open gnuplot pipe.\n");
      NERROR = 1000; exit(NERROR);
     }

   fprintf(gnuplotPipe,"reset\n");
   fprintf(gnuplotPipe,"set terminal png size 1200,%d\n", (int)round
       (1200.0*(double)dicomPenelope::dimDicom[1]/(double)
       dicomPenelope::dimDicom[0]));
   fprintf(gnuplotPipe,"set output 'isoCurves/z%6.2fcm.png'\n",(k
       -0.5)*dimVox[2]);
   fprintf(gnuplotPipe,"set contour base\n");
   fprintf(gnuplotPipe,"set cntrparam levels %d\n", nCurves);
   fprintf(gnuplotPipe,"set cntrparam levels increment %6.2f,%6.2f\n"
       , percMin, curvesGap);
   fprintf(gnuplotPipe,"set palette model RGB defined (0 'gray',1 '
       black')\n");
   //fprintf(gnuplotPipe,"set pal gray negative\n");
   fprintf(gnuplotPipe,"set pm3d map\n");
   fprintf(gnuplotPipe,"unset colorbox\n");
   fprintf(gnuplotPipe,"set multiplot\n");
   fprintf(gnuplotPipe,"splot '-' using 1:2:3 with image title ''\n")
       ; //Plot DICOM plane

   //print dicom plane to gnuplot

   int zindex = k*dicomVox[0]*dicomVox[1];

   for(int yy = 0; yy < dicomVox[1]; yy++)
     {
      int yindex = yy*dicomVox[0];
      for(int xx = 0; xx < dicomVox[0]; xx++)
        {
     int totalindex = zindex+yindex+xx;
```

```
   fprintf(gnuplotPipe," %f %f %d \n", (xx+0.5)*dimVox[0], (yy
      +0.5)*dimVox[1], dicomPenelope::dicomImage[totalindex]);

     }
    fprintf(gnuplotPipe," \n");
  }


fprintf(gnuplotPipe,"e\n"); //end of dicom data

fprintf(gnuplotPipe,"splot ’-’ w l lt -1 lw 1.5 nosurface\n"); //
    isodose contours

int zIndex = nxy*(k-1);
for(unsigned int j=yvoxmin; j <= yvoxmax; j+=rebin)
  {
    int yIndex = zIndex+nx*(j-1);
    for(unsigned int i=xvoxmin; i <= xvoxmax; i+=rebin)
     {
   double q = 0.0;
   double sigma = 0.0;
   for(unsigned int jj = 0; jj < rebin; jj++)
    {
      if(jj+j > ny){break;}

      int yyIndex = yIndex+nx*jj;
      for(unsigned int ii = 0; ii < rebin; ii++)
        {
     if(ii+i > nx){break;}

     int vox = yyIndex+i-1+ii;   // Absolute voxel index
     double fact = 0.0;
     double thismassvox = fmassvox(vox);
     if(thismassvox > 0.0){ fact = 1.0/thismassvox;} // Voxel
         mass may be null if partial vol
     double qAux = VDDmod::edep[vox]*invn;
     double tmpdblAux = (VDDmod::edep2[vox]*invn-pow(q,2))*invn;

     if(tmpdblAux < 0.0){ tmpdblAux=0.0;}
     sigma += tmpdblAux*pow(fact,2);

     qAux = qAux*fact;
```

```
        if(VDDmod::edep[vox] > maxq && fact > 0.0)
          {
            avesig = avesig+sigma/pow(qAux,2);
            nchan++;
          }

        double percentAux = qAux*ev2perCent;
        //Calculate acumulative data
        int contourIn = -1;
        for(int k2 = 0; k2 < nContours; k2++)
          {
#ifdef _USE_OMP_
            if(contourMatID[k2] == geovoxMod::matvox[vox])
#else
            if(dicomPenelope::contourMatID[k2] == geovoxMod::matvox[
               vox])
#endif
             {
           contourIn = k2;
           break;
             }
          }

        if(contourIn != -1) //this voxels is in some contour
          {
            //Calculate bin position
            int bin = percentAux/contourBinWidth;
            if(bin < contourBins)
             {
#ifdef _USE_OMP_
           localContourDat[contourIn][bin]++;
#else
           contourDat[contourIn][bin]++;
#endif
             }
            else
             {
#ifdef _USE_OMP_
           localContourDat[contourIn][contourBins]++;
#else
           contourDat[contourIn][contourBins]++;
#endif
             }
```

```
          }

        q += qAux;
           }
       }

     q = q/(rebin2);
     sigma = invRebin2*sqrt(sigma);

     //convert to % of prescribed dose
     q *= ev2perCent;
     sigma *= ev2perCent;

     int vox = yIndex+i-1;   // Absolute voxel index

     VDDmod::edep[vox] = q;
     VDDmod::edep2[vox] = sigma;

     fprintf(gnuplotPipe," %f %f %lE \n", (double)(i-1.0+rebin*0.5)*
         dicomPenelope::DimVoxelDicom[0], (double)(j-1.0+rebin*0.5)*
         dicomPenelope::DimVoxelDicom[1], VDDmod::edep[vox]);

        }
      fprintf(gnuplotPipe," \n");
    }
   fprintf(gnuplotPipe,"e\n");

   fprintf(gnuplotPipe,"unset multiplot\n");

   fclose(gnuplotPipe);
     }

#ifdef _USE_OMP_

    //Reduce results

    int nthreads = omp_get_num_threads();

    for(int i = 0; i < nContours; i++)
     {
   for(int j = 0; j < contourBins+1; j++)
    {
           #pragma omp atomic
```

```
      contourDat[i][j] += localContourDat[i][j];
    }
     }


    for(int i = 0; i < nContours; i++)
     {
   delete [] localContourDat[i];
     }
    delete [] localContourDat;
    delete [] contourMatID;
  }
#endif

  fprintf(out, "
     #>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n
     ");
  fprintf(out, "# [SECTION REPORT ISODOSE CURVES]\n");
  fprintf(out, "#\n");
  for(int i = 0; i < dicomPenelope::nContours; i++)
    {
     fprintf(out, "# contour %4d -> %s nVoxels: %10d Volume: %15.4e
         cm^3\n", i, dicomPenelope::contourName[i],dicomPenelope::
         nVoxContour[i],dicomPenelope::nVoxContour[i]*geovoxMod::dx*
         geovoxMod::dy*geovoxMod::dz);
    }

  fprintf(out, "#\n");
  fprintf(out, "# prescribed dose: %14.4f Gy\n", prescribDose);
  fprintf(out, "#\n");
  fprintf(out, "# contour | Dose(%%) | Volume(%%)\n");

  FILE *gnuplotPipe = popen("gnuplot -persist", "w"); // Create
     gnuplot pipe

     fprintf(gnuplotPipe,"set term png enhance size 1600,1000 \n");
     fprintf(gnuplotPipe,"set out 'cumulative.png' \n");
     fprintf(gnuplotPipe,"Font=',25' \n");
     fprintf(gnuplotPipe,"set key font Font \n");
     fprintf(gnuplotPipe,"set xtics font Font \n");
     fprintf(gnuplotPipe,"set ytics font Font \n");
     fprintf(gnuplotPipe,"set xlabel font Font \n");
     fprintf(gnuplotPipe,"set ylabel font Font \n");
```

```
    fprintf(gnuplotPipe,"set title font Font \n");
    fprintf(gnuplotPipe,"set lmargin 23 \n");
    fprintf(gnuplotPipe,"set ylabel offset -9 \n");
    fprintf(gnuplotPipe,"set xlabel offset 0,-2 \n");
    fprintf(gnuplotPipe,"set xtics offset 0,-1 \n");
    fprintf(gnuplotPipe,"set grid \n");
    fprintf(gnuplotPipe,"set xrange [0:250] \n");
    fprintf(gnuplotPipe,"set yrange [0:100] \n");
    fprintf(gnuplotPipe,"set xlabel ’Dose(%%)’ \n");
    fprintf(gnuplotPipe,"set ylabel ’Volume(%%)’ \n");
    fprintf(gnuplotPipe,"plot ’-’ using 1:2 ls 1 t ’%s’ smooth
       csplines", dicomPenelope::contourName[0]);

for(int i = 1; i < dicomPenelope::nContours; i++)
  {
    fprintf(gnuplotPipe,", ’-’ using 1:2 ls %d t ’%s’ smooth
       csplines",i+1,dicomPenelope::contourName[i]);
  }

fprintf(gnuplotPipe,"\n");




for(int i = 0; i < dicomPenelope::nContours; i++)
  {
    for(int j = contourBins-1; j >= 0; j--)
 {
   contourDat[i][j] += contourDat[i][j+1];

   fprintf(out, " %4d     %4.2f      %4.2f\n", i, contourBinWidth*(j
      +0.5), (double)contourDat[i][j]/(double)dicomPenelope::
      nVoxContour[i]*100.0);
   fprintf(gnuplotPipe,"%f %f\n", contourBinWidth*(j+0.5), (double)
      contourDat[i][j]/(double)dicomPenelope::nVoxContour[i
      ]*100.0);
 }
    fprintf(gnuplotPipe,"e\n");
  }

fclose(gnuplotPipe);

//Free memory
```

```
  for(int i = 0; i < dicomPenelope::nContours; i++)
    {
      delete [] contourDat[i];
    }
  delete [] contourDat;

  //Calculate uncertainty
  double uncert = 200.0*sqrt(avesig/nchan);

  fprintf(out," \n");
  fprintf(out,"# Performance report\n");
#ifdef _PENELOPE_WITH_MPI_
  fprintf(out,"# Voxel dose distribution process time (s):\n");
  fprintf(out,"# %12.5E\n", VDDmod::mpi_reduce_time);
#endif
  fprintf(out,"# Tally process time (s):\n");
  fprintf(out,"# %12.5E\n", realtime()-initProcessTime);
  fprintf(out,"# Random seeds:\n");
  fprintf(out,"# %10d\n",*RSEED::seed1);
  fprintf(out,"# %10d\n",*RSEED::seed2);
  fprintf(out,"# No. of histories simulated [N]:\n");
  fprintf(out,"# %17.0f.\n",total_nhist);
  fprintf(out,"# CPU time [t] (s):\n");
  fprintf(out,"# %12.5E\n",cputim);
  if (cputim > 0.0)
    {
      fprintf(out,"# Speed (histories/s):\n");
      fprintf(out,"# %12.5E\n",total_nhist/cputim);
    }
  fprintf(out,"# Average uncertainty (above 1/2 max score) in %% [
      uncert]:\n");
  fprintf(out,"# %12.5E\n",uncert);
  double eff = n*pow(uncert,2);
  if (eff > 0.0)
    {
      fprintf(out,"# Intrinsic efficiency [N*uncert^2]^-1:\n");
      fprintf(out,"# %12.5E\n",1.0/eff);
    }
  eff = cputim*pow(uncert,2);
  if (eff > 0.0)
    {
      fprintf(out,"# Absolute efficiency [t*uncert^2]^-1:\n");
```

```
      fprintf(out,"# %12.5E\n",1.0/eff);
    }
  fclose(out);



  return;
}
```

Referències

[1] http://ark.intel.com/products/80807/Intel-Core-i7-4790K-Processor-8M-Cache-up-to-4_40-GHz. Accessed: 2017-09-01.

[2] Cuda. https://developer.nvidia.com/cuda-downloads. Accessed: 2017-07-7.

[3] DICOM. http://dicom.nema.org/standard.html. Accessed: 2017-07-2.

[4] dicompyler. http://www.dicompyler.com/. Accessed: 2017-08-29.

[5] dicomsdl. https://code.google.com/archive/p/dicomsdl/. Accessed: 2017-07-9.

[6] EC3. http://servproject.i3m.upv.es/ec3/. Accessed: 2017-09-01.

[7] Egsnrc: software tool to model radiation transport. https://github.com/nrc-cnrc/EGSnrc. Accessed: 2017-07-2.

[8] Epha. https://epha.org. Accessed: 2017-07-2.

[9] Estar. https://physics.nist.gov/PhysRefData/Star/Text/ESTAR.html. Accessed: 2017-07-2.

[10] gcc. https://gcc.gnu.org/. Accessed: 2017-07-7.

[11] gnuplot. http://www.gnuplot.info/. Accessed: 2017-07-9.

[12] gprof. https://sourceware.org/binutils/docs/gprof/. Accessed: 2017-07-7.

[13] IAEA. https://www-nds.iaea.org/relnsd/vcharthtml/VChartHTML.html. Accessed: 2017-08-23.

[14] Mellanox. http://www.mellanox.com/products/interconnect/ethernet-overview.php. Accessed: 2017-08-30.

[15] National cancer institute. https://seer.cancer.gov. Accessed: 2017-07-2.

[16] Nist. https://physics.nist.gov. Accessed: 2017-08-24.

[17] Omp. http://www.openmp.org/. Accessed: 2017-07-7.

[18] openmpi. `https://www.open-mpi.org/`. Accessed: 2017-07-9.

[19] Dolan, J., Li, Z., Williamson, J.F. Monte Carlo and experimental dosimetry of an I-125 brachytherapy seed. *Med. Phys.*, 33:4675–4684, 2006.

[20] F.Almeida, D.Giménez, J.M.Mantas, A.Vidal. Introducción a la Programación Paralela. *Paraninfo*, 2008.

[21] F. H. Attix. Introduction to radiological physics and radiation dosimetry. *Wiley, New York*, 1986.

[22] F.Salvat. Penelope. a code system for monte carlo simulation of electron and photon transport. *OECD Nuclear Energy Agengy, France*, 2014.

[23] F.Salvat, Åsa Carlsson Tedgren, Jean-François Carrier, Stephen D. Davis, Firas Mourtada, Mark J. Rivard, Rowan M. Thomson, Frank Verhaegen, Todd A. Wareing, Jeffrey F. Williamson. Report of the task group 186 on model-based dose calculation methods in brachytherapy beyond the tg-43 formalism: Current status and recommendations for clinical implementation. *Med. Phys.*, 2012.

[24] GEANT4 Collaboration (Agostinelli. S. et al.). Geant4: A simulation toolkit. *Nucl.Instrum.Meth. A506 250-303 SLAC-PUB-9350, FERMILAB-PUB-03-339*, 2003.

[25] Los Alamos Scientific Laboratory. Group X-6. Mcnp : a general monte carlo code for neutron and photon transport. los alamos, n.m. *Dept. of Energy, Los Alamos Scientific Laboratory*, 1979.

[26] M. J. Rivard, B. M. Coursey, L. A. DeWerd, W. F. Hanson, M. S. Huq, G. S. Ibbott, M. G. Mitch, R. Nath, and J. F. Williamson. Update of aapm task group no. 43 report: A revised aapm pro- tocol for brachytherapy dose calculations. *Med. Phys.*, 2004.

[27] Vicent Giménez-Alventosa, Paula C. G. Antunes, Javier Vijande, Facundo Ballester, José Pérez-Calatayud, Pedro Andreo. Collision-kerma conversion between dose-to-tissue and dose-to-water by photon energy-fluence corrections in low-energy brachytherapy. *Physics in Medicine and Biology*, 2016.

[28] Yuhe Wang, Thomas R. Mazur, Olga Green, Yanle Hu, Hua Li, Vivian Rodriguez,H. Omar Wooten, Deshan Yang, Tianyu Zhao, Sasa Mutic, and H. Harold Li. A gpu-accelerated monte carlo dose calculation platform and its application toward validating an mri-guided radiation therapy beam model. *Med. Phys.*, 2016.