Arey Ferrero Ramos

OPTIMITZACIÓ I PARAL·LELITZAT D'UN ALGORISME PER REALITZAR UN TEST DE *NESTEDNESS* D'UNA MATRIU BINARIA

TREBALL DE FI DE GRAU

Dirigit pel Sr. Carles Aliagas Castell

Doble grau d'enginyeria informàtica i biotecnologia

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2024

Resum.

Durant la meva estada en pràctiques en el grup de recerca SEES:lab, vaig desenvolupar un algorisme en **Python** per tal d'esbrinar si la microbiota intestinal d'un conjunt d'espècies de vertebrat s'organitza segons un **patró *nested***. Tot i que els resultats obtinguts al testejar el programa van demostrar que el seu funcionament era correcte, el seu temps d'execució pel volum de dades més gran del que es disposava es va especular que seria d'unes **280 hores**. Tenint en compte l'interès científic que poden tenir els resultats d'aquest estudi es va decidir portar a terme un projecte d'optimització del codi de l'algorisme.

El projecte s'ha organitzat en dos blocs. El primer ha consistit en una etapa d'**optimització seqüencial**. En aquesta fase s'han aplicat un conjunt de millores en el codi Python, s'ha traduït l'algorisme al llenguatge de programació **C** i s'han realitzat un altre conjunt de millores en el codi traduït. Després, s'ha portat a terme una segona etapa d'**optimització paral·lela**. Aquí s'han utilitzat les eines **OpenMP** i **MPI** per paral·lelitzar el codi C, i s'han fet proves d'execució en el meu ordinador portàtil, en les màquines del zoo **Orca** i **Pop** i en la **Sala dels Mac**.

Després de totes les optimitzacions aplicades, s'ha obtingut un codi programat en C i paral·lelitzat amb OpenMP i MPI que, en executar-se en la màquina Orca, triga **3,223 segons** en obtenir els resultats de l'estudi de *nestedness* pel volum de dades més gran.

Resumen.

Durante mi estancia en prácticas en el grupo de investigación SEES:lab, desarrollé un algoritmo en **Python** con el objetivo de averiguar si el microbiota intestinal de un conjunto de especies de vertebrado se organiza según un **patrón *nested***. Aunque los resultados obtenidos al testear el programa probaron que su funcionamiento es correcto, su tiempo de ejecución para el volumen de datos más grande del que se disponía se especuló que sería de unas **280 horas**. Teniendo en cuenta el interés científico que pueden tener los resultados de este estudio se decidió llevar a cabo un proyecto de optimización del código del algoritmo.

El proyecto se ha organizado en dos bloques. El primero ha consistido en una etapa de **optimización secuencial**. En esta fase se han aplicado un conjunto de mejoras en el código Python, se ha traducido el algoritmo al lenguaje de programación **C** i se ha realizado otro conjunto de mejoras en el código traducido. Después, se ha llevado a cabo una segunda etapa de **optimización paralela**. Aquí se han utilizado las herramientas **OpenMP** y **MPI** para paralelizar el código C y se han hecho pruebas de ejecución en mi ordenador portátil, en las máquinas del zoo **Orca** y **Pop**, y en la **Sala de los Mac**.

Después de todas las optimizaciones aplicadas, se ha obtenido un código programado en C y paralelizado con OpenMP y MPI que, al ejecutarse en la máquina Orca, tarda **3,223 segundos** en proporcionar los resultados del estudio de *nestedness* para el volumen de datos más grande.

Abstract.

During my internship in the SEES:lab research group, I developed an algorithm in **Python** with the aim of finding out if the intestinal microbiota of a set of vertebrate species is organized according to a **nested pattern**. Although the results obtained when testing the program proved that it works correctly, its execution time for the largest volume of data that was available was speculated to be about **280 hours**. Considering the scientific interest that the results of the study may have, it was decided to carry out a project to optimize the code of the algorithm.

The project has been organized in two blocks. The first consisted of a **sequential optimization** stage. In this phase, a set of improvements have been applied to the Python code, the algorithm has been translated into the **C** programming language, and another set of improvements has been made to the translated code. Afterwards, a second stage of **parallel optimization** has been carried out. Here the **OpenMP** and **MPI** tools have been used to parallelize the C code and execution tests have been carried out on my laptop, on the zoo machines **Orca** and **Pop**, and in the **Mac Room**.

After all the optimizations applied, a code programmed in C and parallelized with OpenMP and MPI has been obtained and, when it is executed on the Orca machine, it takes **3.223 seconds** to provide the results of the nestedness study for the largest data volume.

Índex

Índex de taules

Índex de gràfics

Índex de figures

Índex d'equacions

1 Introducció

1.1 Estudi de *nestedness*

La microbiota intestinal és una comunitat microbiana allotjada al tracte gastrointestinal, que s'encarrega de modular molts processos fisiològics imprescindibles per garantir la salut dels éssers humans [1]. Degut a la seva importància, s'ha convertit en la comunitat microbiana més estudiada [2]. Això ha permès nombrosos descobriments, com que una elevada diversitat d'espècies bacterianes en la microbiota intestinal es correspon amb una millor salut de l'hoste [3] o l'existència de patrons comuns en la composició de les microbiotes saludables [4]. No obstant, encara és necessària més recerca en certs àmbits com alhora de determinar la composició d'una microbiota saludable [2] o de desenvolupar un mecanisme de classificació per les microbiotes que consideri hostes de poblacions geogràficament diferenciades [4].

La microbiota intestinal també es troba al tracte gastrointestinal de molts animals [5], a on també desenvolupa processos fisiològics importants [6]. No obstant, aquests processos han estat molt menys estudiats que aquells realitzats per la microbiota humana [7]. Tot i així, s'ha constatat la utilitat de la recerca feta en l'àmbit de la microbiota intestinal dels animals per combatre desequilibris nutricionals en aquests [8] o per facilitar la reintroducció d'espècies en el seu habitat natural [9].

Per aquesta raó, durant la meva estada en pràctiques externes curriculars en el grup de recerca SEES:lab del DEQ[1] de la URV, vaig realitzar un projecte amb la Dra. Marta Sales Pardo, consistent en l'anàlisi de la composició de la microbiota intestinal d'individus salvatges i captius en vint-i-cinc espècies de vertebrat, les mostres de les quals es van obtenir de l'estudi [10]. Dins d'aquest projecte vaig realitzar quatres estudis diferents: Un estudi de diversitats, un estudi de similituds, un estudi de correlacions entre similituds i diversitats i un estudi de *nestedness*. Per cada un, vaig haver d'implementar un codi en Python per organitzar les abundàncies absolutes de cada gènere bacterià en la microbiota en l'estructura de dades adient i, sobre aquesta estructura, fer els càlculs necessaris per generar uns resultats i organitzar-los en diferents gràfics per la seva discussió posterior. Tot aquest codi es troba disponible al següent repositori de GitHub [11].

De tots els estudis realitzats, el de major interès científic és l'estudi de *nestedness*. En les microbiotes humanes s'ha observat que, si aquestes s'organitzen en matrius binaries d'abundàncies relatives d'espècies bacterianes (o agrupacions d'aquestes espècies bacterianes) en un conjunt d'individus (o agrupacions d'aquests individus), aquestes segueixen un determinat patró jeràrquic niat anomenat **patró *nested*** [2]. Matemàticament, les matrius binàries amb aquest patró presenten una sobreabundància de uns per sobre de la diagonal secundària i una sobreabundància de zeros per sota d'aquesta [2] (Figura 1).

Aquest patró ja s'havia observat en altres sistemes ecològics com en faunes d'hàbitats fragmentats o arxipèlags [12], en comunitats d'endoparàsits d'animals [13] [14] o en organitzacions mutualistes [16] [17]. En les microbiotes d'individus de l'espècie humana, un patró *nested* indica, d'una banda, l'existència de microbis generalistes que estan presents en la major part dels individus (o agrupacions d'aquests individus) i microbis especialistes que estan presents només en una part d'aquests, i, per l'altra banda, l'existència d'hostes

[1] Departament d'Enginyeria Química

generalistes que presenten la major part dels microbis (o agrupacions d'aquests microbis) i hostes especialistes que presenten tan sols una part d'aquests [2].

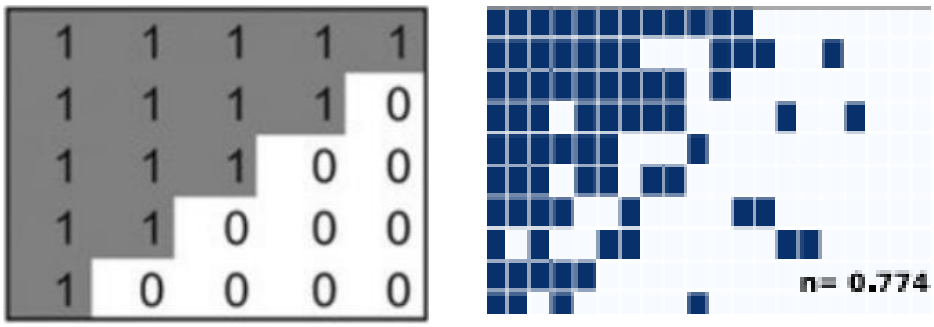


Figura 1. Representació d'una matriu binaria teòrica amb un patró *nested* perfecte (esquerra) i d'una matriu d'abundàncies relatives discretitzades de grups de microbis en grups d'individus que segueix un patró *nested* real amb el seu valor de *nestedness* (dreta).

El coneixement de l'existència d'aquest patró en les microbiotes humanes és rellevant perquè es tracta d'un factor que cal tenir en consideració en la investigació en tractaments que impliquen medicina personalitzada [2]. En el cas de les microbiotes d'espècies d'animals no humans, el descobriment d'un patró *nested* pot tenir interès científic i, fins i tot, aplicacions igual de rellevants a l'esmentada en éssers humans.

Per desgracia, el temps d'execució de l'algorisme que vaig dissenyar i implementar per realitzar un test de *nestedness* era massa elevat i, per tant, aquest va ser l'únic estudi del qual no vam poder obtenir uns resultats per discutir. Aquest és el punt de partida del projecte descrit en aquesta memòria.

A continuació, s'expliquen les diferents seccions que integren l'estudi de *nestedness*.

1.1.1 Obtenció de les dades de microbiotes intestinals d'animals

Per realitzar l'estudi de *nestedness*, han estat necessaris dos fitxers del repositori de GitHub associat a l'estudi [10]. El primer, i més important, és el fitxer 'count_Genus_all.tsv', que és un fitxer estructura en forma de matriu que conté el número d'espècies de cada gènere bacterià en cada individu del qual s'han obtingut mostres (**abundàncies absolutes**). Les dades contingudes en aquest fitxer son les que s'emmagatzemaven en qualsevol de les estructures de dades del projecte descrit anteriorment, incloses les matrius utilitzades en aquest estudi. Per altra banda, també ha estat necessari el fitxer 'metadata.csv', que conté la informació per identificar tots els individus. D'aquest fitxer, únicament ens interessa conèixer el codi de l'espècie de vertebrat a la que pertany l'individu i el tipus d'individu (salvatge i captiu). Aquesta informació és imprescindible per construir la matriu de vertebrats (descrita en el següent subapartat).

1.1.2 Matrius d'abundàncies relatives

Per a poder portar a terme l'estudi de *nestedness*, les dades del fitxer 'count_Genus_all.tsv' es van organitzar en matrius d'abundàncies relatives. Una **matriu d'abundàncies relatives** és una matriu **M** en la que cada element $\mathbf{M_{ij}}$ representa l'abundància relativa d'un gènere bacterià **i** en un individu o grup d'individus **j**, de manera que es compleix que $\sum_i M_{ij} = 1$.

$$M_{ij} = \frac{Número\ de\ bacteris\ del\ gènere\ bacterià\ i\ en\ l'individu/grup\ j}{Número\ de\ bacteris\ totals\ en\ l'individu/grup\ j} \qquad (1)$$

L'individu o grup d'individus **j** fa referència a un individu d'una espècie de vertebrat o a la totalitat d'individus d'un sol tipus (salvatge o captiu) d'una espècie de vertebrat.

Tot i que això pot donar lloc a una gran varietat de matrius d'abundàncies relatives, per el desenvolupament d'aquest projecte tant sols s'han tingut en compte dues. Una és la **matriu d'individus**, que s'obté dividint les abundàncies absolutes de cada gènere bacterià entre la suma de les abundàncies absolutes de cada individu. Aquesta primera matriu té sis-cents quaranta-quatre files i mil cinquanta-sis columnes. L'altra és la **matriu de vertebrats**, que s'obté dividint les abundàncies absolutes de cada gènere bacterià entre la suma de les abundàncies absolutes de cada agrupació d'individus d'un tipus (salvatge o captiu). Aquesta segona matriu té cinquanta files i mil cinquanta-sis columnes.

1.1.3 Discretitzat de les matrius d'abundàncies relatives

Per a poder fer un estudi de *nestedness* d'una matriu, cal que aquesta sigui una matriu binaria [2]. Per tant, s'ha de discretitzar la matriu d'abundàncies relatives **M** que s'estigui avaluant per donar lloc a una matriu binària **Q**. És a dir, cal escollir un valor llindar **T** per sota del qual es pot considerar que les abundàncies relatives $\mathbf{M_{ij}}$ son negligibles.

$$Q = \begin{cases} q_{ij} = 1 \ \ si \ \ M_{ij} \geq T \\ q_{ij} = 0 \ \ si \ \ M_{ij} < T \end{cases} \qquad (2)$$

El llindar T idoni serà aquell que maximitzi el valor de *nestedness* de la matriu Q una vegada s'hagi calculat el valor de *nestedness* per tots els llindars possibles d'un conjunt preestablert. Així, el llindar T idoni de la **matriu de vertebrats** és $\mathbf{10^{-6}}$ i el llindar T idoni de la **matriu d'individus** és $\mathbf{10^{-4}}$.

1.1.4 Càlcul del valor de nestedness d'una matriu binaria

Per dissenyar l'algorisme per calcular el valor de *nestedness* d'una matriu binària, es parteix de la següent equació [2]:

$$n = \frac{\sum_{a<b} q_{ab}{}^{(i)} + \sum_{a<b} q_{ab}{}^{(j)}}{\sum_{a<b} \min(q_a{}^{(i)}, q_b{}^{(i)}) + \sum_{a<b} \min(q_a{}^{(j)}, q_b{}^{(j)})} \qquad (3)$$

On **i** representa el gènere bacterià (files); **j** representa l'individu o grup d'individus d'un tipus (columnes); **q_a** (**q_b**) representa el número d'interaccions (elements diferents de zero) en una fila (columna); **q_ab** representa el número d'interaccions compartides entre dues files o dues columnes (elements diferents de zero que tenen en comú dues files i dues columnes).

En la versió original de l'algorisme per calcular el valor de *nestedness* d'una matriu binaria, cada un dels quatre sumatoris que fan la funció d'operands en l'equació s'ha implementat com una estructura iterativa anidada.

1.1.5 Test de nestedness

Per si sol, el valor de *nestedness* d'una matriu binaria no ens proporciona suficient informació per determinar si aquesta té o no té una patró *nested*. Aquesta disjuntiva ve determinada per la hipòtesi nul·la. La **hipòtesi nul·la** proposa que el valor de *nestedness* calculat és només una conseqüència de la densitat de la matriu d'abundàncies discretitzades Q. Per tant, és necessari comparar el valor de *nestedness* calculat amb el de la hipòtesi nul·la. D'això se'n diu fer un **test de *nestedness***.

És en aquest escenari on entra en joc el concepte de randomització. Una **randomització** és una matriu binaria que es genera redistribuint aleatòriament aquelles posicions de la matriu Q que tenen un valor igual a 1. Per tant, les randomitzacions tenen el mateix nombre de files, el mateix nombre de columnes i el mateix nombre de valors iguals a 1 que la matriu Q, però es diferencien en les posicions que aquest valors iguals a 1 ocupen (Figura 2).

Per a que el test de *nestedness* sigui consistent, cal generar, com a mínim, **mil randomitzacions**. Per cada una d'aquestes randomitzacions, se'n calcula el valor de *nestedness* i s'emmagatzemen en una llista de manera conjunta amb el de la matriu Q. Aquesta llista s'ordena i se'n calcula el **p-valor**. En aquest estudi, el p-valor és **la fracció de randomitzacions que tenen un valor de *nestedness* més gran que el de la matriu Q** i indica si aquest valor de *nestedness* es pot considerar estadísticament significatiu o no. Es considera que la matriu Q té un patró *nested* si el p-valor és igual o inferior a **0.05** [17].

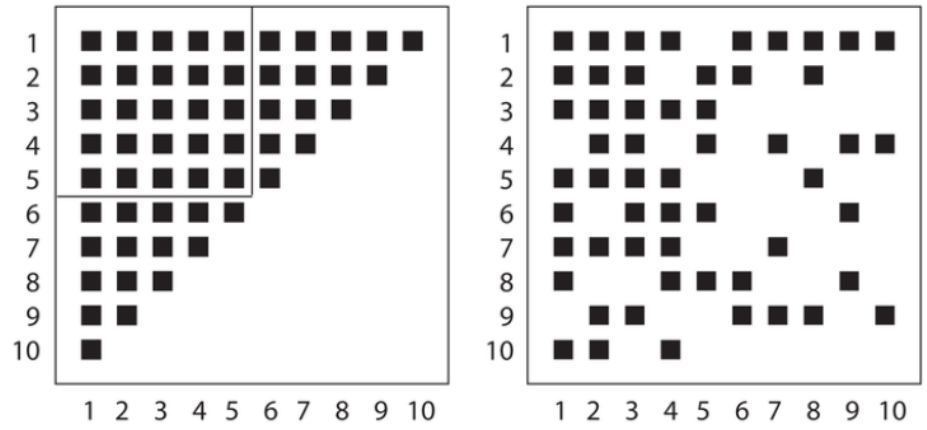


Figura 2. Comparativa entre una matriu binaria amb un patró *nested* perfecte i una de les seves possibles randomitzacions.

1.2 Esquema de l'algorisme original

A continuació es mostra un esquema en pseudocodi de certes estructures destacades de la versió original de l'algorisme per realitzar un test de *nestedness*. Aquest esquema permet obtenir una visió general del funcionament del programa que facilitarà la comprensió de les optimitzacions seqüencials i paral·leles que s'han plantejat durant el desenvolupament del projecte.

```
algorisme estudi_nestedness és

      $ Lectura dels fitxers amb les dades d'entrada

      $ i organització d'aquestes dades en matrius d'abundàncies relatives.


      matriu_binària <- discretitzar_matriu(matriu_abundàncies, llindar);


      valor_nested, p_valor <- test_nestedness(matriu_binària,
      número_randomitzacions);

falgorisme



funció discretitzar_matriu(matriu_abundàncies: reals, llindar: real) retorna
matriu_binària: enters és

      $ El codi Python per discretitzar una matriu es mostra en l'annex 1.

ffunció
```

```
funció test_nestedness(matriu_binària: enters, número_randomitzacions: enter)
retorna valor_nested: real, p_valor: real és

	llista_valors_nested <- generar_valors_nested_randomitzats(matriu_binària,
	número_randomitzacions);


	valor_nested <- calcular_valor_nested(matriu_binària);

	llista_valors_nested[número_randomitzacions]  <- valor_nested;


	ordenar(llista_valors_nested);


	$ Càlcul del p-valor a partir de número_randomitzacions i de l'índex que
	$ valor_nested ocupa en llista_valors_nested.


	retorna valor_nested, p_valor

ffunció




funció generar_valors_nested_randomitzats(matriu_binària: reals,
número_randomitzacions: enter) retorna llista_valors_nested: reals és

	número_uns <- comptar_uns(matriu_binària);

	per (pos <- 0; pos < número_randomitzacions; pos <- pos + 1) fer

		randomització <- generar_randomització(matriu_binària, número_uns);

		llista_valors_nested[pos] <- calcular_valor_nested(randomització);

	fper

	retorna llista_valors_nested;

ffunció




funció calcular_valor_nested(matriu_binària: enters) retorna valor_nested: real és

	primera_component <- primera_estructura_iterativa_anidada;

	segona_component <- segona_estructura_iterativa_anidada;

	tercera_component <- tercera_estructura_iterativa_anidada;

	quarta_component <- quarta_estructura_iterativa_anidada;


	valor_nested <- (primera_component + segona_component) / (tercera_component
	+ quarta_component);

	retorna valor_nested;

ffunció
```

```
$ El codi Python per realitzar el test de nestedness d'una matriu binaria es mostra
$ en l'annex 2 i el codi Python per calcular el valor de nestedness d'una matriu
$ binaria es mostra en l'annex 3.
```

1.3 Objectius

- Aplicar els coneixements apressos sobre c, optimització de codi i paral·lelitzat amb les eines OpenMP i MPI així com les competències desenvolupades durant el grau a un cas pràctic del món real.
- Millorar el rendiment del programa fins assolir el mínim temps d'execució possible.
- Esbrinar si les dues matrius de l'estudi són, en efecte, *nested*.

2 Materials i metodologies

2.1 Ordinador Asus ROG Strix G531G

El projecte s'ha desenvolupat i se n'han realitzat les primeres proves d'execució en el meu ordinador portàtil, un Asus ROG Strix G531G. Les especificacions *hardware* d'aquesta màquina son les següents [18] [19]:

- Processador **Intel Core i7-9750H** (2,60 GHz – 4,50 GHz). 6 *cores* 12 *threads*. Memòria *Cache* 12 MB.
- Targeta gràfica NVIDIA GeForce GTX 1660Ti 4GB GDDR6.
- Memòria RAM 16 GB DDR4 2666 MHz.
- SSD NVMe Western Digital 256 GB.
- SSHD SATA III Segate 1 TB.

2.2 Sistema Operatiu Ubuntu (64-bit) virtualitzat

Tot i que el SO[2] natiu del ordinador Asus ROG Strix G531G és un Windows 10 Pro [18] [19], el projecte s'ha desenvolupat en un SO **Ubuntu** (64-bit) virtualitzat amb l'ajut de l'administrador de màquines virtuals Oracle VM VirtualBox [20]. Ubuntu és una distribució de **Linux** basada en Debian amb *software* lliure i de codi obert [21]. Del *hardware* descrit en el subapartat anterior, a aquest SO virtualitzat se li han assignat 6 unitats d'execució (3 *cores* i 6 *threads*), 8 GB de memòria principal i 100 GB de disc.

El motiu pel qual es va decidir en primera instància desenvolupar el projecte que vaig realitzar durant la meva estada en pràctiques en una distribució de Linux va ser perquè el desenvolupament de projectes en el llenguatge de programació Python en aquest SO és més senzill, còmode i productiu [22] [23] [24]. Cal tenir en compte que el rendiment d'un programa en un SO Linux virtualitzat probablement sigui inferior al que aquest mateix programa tindrà en un SO Linux natiu sobre la mateix màquina.

2.3 Git

Git és un sistema de control de versions distribuït, lliure i de codi obert [25]. Cada vegada que s'ha fet un canvi substancial en el projecte, s'ha creat una nova versió amb l'ajut d'aquest *software*. Això ofereix tota una sèrie d'avantatges: Primer, permet la mobilitat entre diferents versions del projecte per a tornar a executar codi antic que ha estat modificat en la versió actual. Segon, permet crear diferents camins (*branches*) cap versions diferents del projecte, en cas de que no es tingui clar quina pot ser la més adequada. Finalment, permet retrocedir fins a una versió anterior del projecte en cas de que no s'aconsegueixi que l'actual funcioni correctament.

[2] Sistema Operatiu

2.4 GitHub

GitHub és una plataforma de desenvolupament *software* per crear, emmagatzemar i gestionar projectes que utilitzen el sistema de control de versions **git** [26]. La utilització d'aquesta eina permet mantenir tot el codi del projecte actualitzat periòdicament en un repositori remot, de manera que se'l protegeix front a possibles pèrdues degut a una fallada greu del sistema en que es desenvolupa de forma local. Es pot accedir al projecte complet que es descriu en aquesta memòria a través del següent enllaç [27].

2.5 Bash

GNU Bash[3] és una implementació específica de un *shell* de Unix [28]. Un *shell* de Unix és una interfície d'usuari de línia de comandes per al SO Unix i els seus derivats, així com un llenguatge de *scripting* [29].

En el projecte que vaig desenvolupar durant l'estada en pràctiques, Bash es va utilitzar per desenvolupar tots els *scripts* per executar el codi en Python del projecte. En el projecte actual, s'han dissenyat nous *scripts* per executar el codi Python de manera paral·lela o amb l'intèrpret PyPy3 i també s'ha utilitzat per construir el Makefile per compilar i executar les diferents versions del codi en C del projecte, així com per esborrar els fitxers binaris que s'hagin generat durant el procés de compilació quan ja no siguin necessaris.

2.6 Python

Python és un llenguatge de programació **interpretat**, d'alt nivell i de propòsit general amb semàntica dinàmica [30] [31]. La sintaxi de Python es caracteritza per la seva llegibilitat [31] i per l'ús de sagnies (quatres espais) per mostrar l'estructura en blocs en lloc dels claudàtors propis d'altres llenguatges de programació [30].

Python és molt utilitzat en Desenvolupament Ràpid d'Aplicacions degut a les seves estructures de dades d'alt nivell i al seu tipat i enllaçat dinàmic [31]. És multi paradigma i, per tant, suporta programació orientada a objectes [30] [31] o programació funcional [32], tot i que el codi que constitueix la base d'**aquest projecte utilitza exclusivament programació estructurada**.

Per últim, és important destacar que Python s'ha convertit en un dels llenguatges de programació més populars en l'àmbit de la ciència de dades [33] [34], motiu pel qual és el llenguatge de programació amb el que treballa el grup de recerca SEES:lab en el que, com he esmentat anteriorment, vaig realitzar les pràctiques curriculars.

A continuació, es descriuen les llibreries de Python que s'han utilitzat en aquest projecte:

[3] GNU Bourne Again SHell

2.6.1 Pandas

Pandas és una llibreria de Python per treballar amb conjunts de dades com si fossin dades estructurades [33] [35]. Per exemple, permet importar fitxers amb extensió '.csv' i crear-ne els *DataFrames* [33]. A més, conté rutines per analitzar, netejar, explorar i manipular aquestes dades [35]. En aquest projecte, pandas s'utilitza per tractar els dos fitxers que contenen les dades d'entrada.

2.6.2 NumPy

NumPy és una llibreria matemàtica per realitzar computació científica [33] [36]. L'objecte base que proporciona és *ndarray*, un *array* multidimensional [33] [36]. D'aquest se'n deriven múltiples objectes (matrius, *arrays* emmascarats,...) i rutines per fer operacions ràpides entre *arrays* (àlgebra lineal, transformades de Fourier,...) [33] [36]. En aquest projecte, la llibreria NumPy s'utilitza per crear les matrius que emmagatzemen les dades sobre les quals es fan els càlculs ja que el seu accés és més ràpid que si s'utilitzen llistes anidades.

2.7 PyPy

PyPy és una implementació alternativa de Python que és compatible amb aquest [37]. La principal característica de PyPy és la seva rapidesa: És, de mitjana, **4,8 vegades més ràpid que Python** [37]. Això el converteix en una opció rellevant per executar programes escrits en Python que tinguin un elevat temps d'execució.

2.8 PyCharm

PyCharm és l'IDE[4] de Python per a desenvolupadors professionals dissenyat per l'empresa de desenvolupament de *Software* JetBrains [38]. Aquest IDE proporciona eines que augmenten la productivitat en el procés de desenvolupament de codi.

2.9 C

C és un llenguatge de programació procedimental, que és un tipus de programació estructurada, i de propòsit general [39] [40]. Es tracta d'un llenguatge **compilat** i, per tant, un programa escrit en C requereix d'un compilador que el converteixi en un fitxer binari per a poder ser executat per un computador [39] [40]. C és un llenguatge amb tipus de dades estàtics, tipat feble i de mig nivell, és a dir, combina les estructures típiques dels llenguatges d'alt nivell amb accés de baix nivell a la memòria del sistema [39] [41] [42].

C es va dissenyar per desenvolupar els SOs Unix i els programes que s'hi han d'executar [39] [43]. Posteriorment es va popularitzar en molts altres àmbits. Tot i l'auge d'altres llenguatges de programació que són considerats més adequats per al desenvolupament d'aplicacions web i per a mòbils [44], a dia d'avui segueix sent un dels llenguatges més utilitzats [45] i és fonamental en la creació de SOs [40] [43], *divers* de dispositiu [46] i compiladors d'altres llenguatges de programació [43].

C s'ha utilitzat com l'eina principal per assolir els objectius del projecte per dues raons. Primer, un programa escrit en C i compilat amb el mecanisme per defecte, pot arribar a ser fins a **45 vegades més ràpid que el mateix programa escrit en Python** [47]. A més, el compilador de C disposa de l'opció -O3. Aquesta opció indica **tercer nivell d'optimització** i permet generar un codi màquina molt més eficient que l'opció per defecte. Amb l'aplicació d'aquesta eina, un programa escrit en C pot ser entre **450 i 45000 vegades més ràpid que el seu equivalent en Python** [47].

La segona raó per la que s'ha utilitzat C com a llenguatge principal ha estat que les eines OpenMP i MPI, utilitzades per resoldre el pas d'optimització paral·lela, no es troben disponibles en Python, cosa que ha fet obligatòria la traducció del codi del projecte al llenguatge C.

2.10 CLion

CLion és l'IDE multi plataforma per a C i C++ dissenyat per l'empresa de desenvolupament de *Software* JetBrains [48]. Al igual que PyCharm, aquest IDE proporciona tota una sèrie d'eines que faciliten el procés de desenvolupament de codi, augmentant-ne així la productivitat.

2.11 Gprof

Gprof és una eina, freqüentment associada als llenguatges de programació C i C++ així com al SO Linux**,** per fer ***profiling*** [49] [50]. Per tant, serveix per recopilar i organitzar estadístiques del **percentatge del temps d'execució total que representa cada procediment en un programa** [50].

Per realitzar el *profiling* amb l'eina gprof s'han de seguir els passos següents [49]:

- Primer, es compila el codi C amb el compilador gcc afegint el *flag* `-pg`.
- A continuació, s'executa el fitxer binari resultant com es faria normalment. Després d'aquest pas s'observarà que en el directori de treball apareix un fitxer anomenat gmon.out.
- Finalment, es processa el fitxer gmon.out amb la comanda `gprof executable gmon.out > profiling_file`.

El fitxer resultant profiling_file contindrà les estadístiques del temps d'execució de cada procediment en el programa. El nom d'aquest fitxer és decisió del programador i dependrà de quina versió del codi és aquella per la que s'hagin generat les estadístiques.

2.12 OpenMP

OpenMP és una API[4] que s'utilitza per executar programes escrits en C, C++ o FORTRAN de forma paral·lela [51] [52]. Segueix el paradigma **MIMD** (*Multiple Instruction Multiple Data*) y totes les dades es troben emmagatzemades en la mateixa **memòria compartida** [51] [52]. Consisteix en un conjunt de variables d'entorn, funcions de llibreria i directives de compilació que s'afegeixen a les seccions de codi que es volen paral·lelitzar per influir el seu comportament en temps d'execució [51]. Per tant, no es una eina intrusiva en el codi original, cosa que fa que el seu aprenentatge sigui ràpid i senzill [52].

OpenMP utilitza el model d'execució paral·lela ***fork-join*** [51] [52] (Figura 3), que funciona bàsicament de la següent manera: Un únic t*hread master* inicia l'execució del programa de forma seqüencial. Quan arriba a una regió paral·lela, crea tants *threads* com s'hagi especificat al principi del programa (*fork*). Un cop tots els *threads* hagin acabat d'executar la secció que els hi correspon, es sincronitzaran i finalitzaran, sent el *thread master* l'únic que continuarà l'execució del programa de forma seqüencial (*join*) [51] [52].

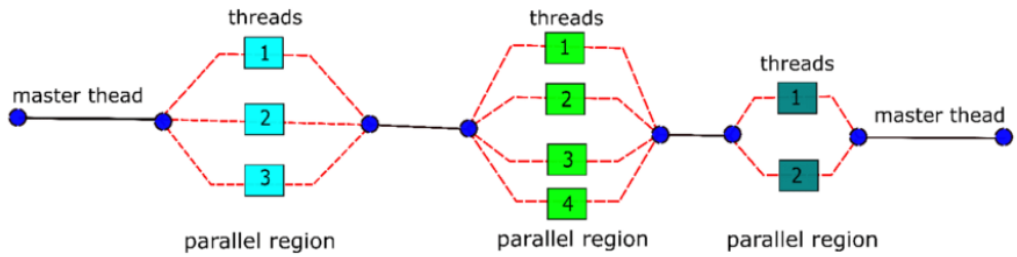


Figura 3. Diagrama del model d'execució paral·lela fork-join que utilitza OpenMP.

2.13 MPI

MPI (*Message Passing Interface*) és una API per executar programes de forma paral·lela [53]. Igual que OpenMP, segueix el paradigma **MIMD**, però es distingeix en que la memòria és distribuïda, és a dir, cada procés té la seva pròpia **memòria privada** [53]. En conseqüència, en el model d'execució d'MPI cada procés és independent i aquests s'hauran de comunicar mitjançant **pas de missatges** [53] (Figura 4).

El principal avantatge de MPI és la possibilitat d'executar el programa no només en màquines que comparteixen la memòria i els processadors d'un mateix servidor sinó en maquines que es troben distribuïdes en diferents servidors [53]. No obstant, la programació amb aquesta eina és més complexa que amb OpenMP, cosa que fa que el seu procés d'aprenentatge sigui més lent [53].

[4] Application Programming Interface

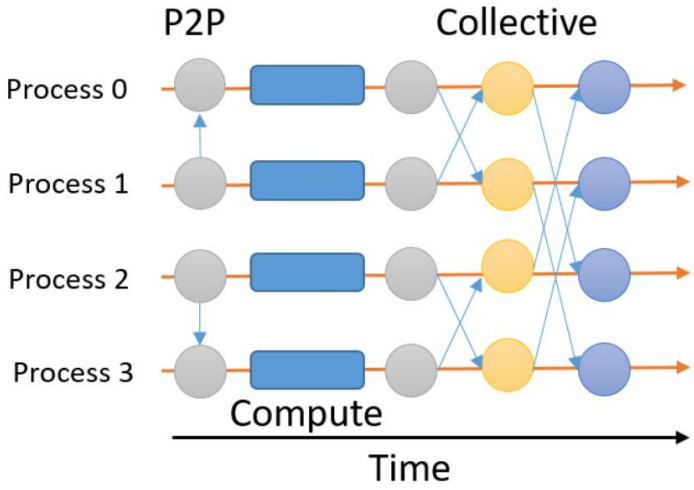


Figura 4. Diagrama del model d'execució paral·lela amb pas de missatges que utilitza MPI.

2.14 Zoo

Per cada eina per executar codi de forma paral·lela de les que s'han descrit es tractarà de dissenyar la versió més òptima del programa que sigui possible en l'ordinador portàtil. Un cop s'hagi assolit aquest objectiu, es pujarà aquesta versió del programa al Zoo. El **Zoo** és un conjunt de serveis i servidors del DEIM als que es pot accedir de forma remota des de qualsevol de les xarxes de la URV. Per connectar-se al Zoo s'utilitza l'eina ssh i per pujar-hi arxius l'eina scp. Un cop la versió més òptima del codi paral·lel estigui dins d'aquest entorn, l'objectiu serà avaluar com evoluciona el rendiment del programa a mesura que s'augmenta el nombre d'unitats d'execució paral·lela (processos i/o *threads*).

Per assolir aquest objectiu, el Zoo proporciona quatre màquines. Abans de descriure'n les prestacions de cada una, cal aclarir que totes aquestes màquines permeten executar codi seqüencial o codi paral·lel que utilitzi qualsevol de les eines que s'han descrit. No obstant, les seves especificacions fan que cada una sigui més adequada per treballar amb una de les eines en específic.

La primera d'aquestes màquines és la màquina Orca. **Orca** té **un** únic **node** amb **un** processador **AMD Rhyzen Threadripper PRO 3995WX**. Aquest processador té 64 *cores* i 128 *threads* i una freqüència màxima de 4,20 GHz [54]. Respecte als nivell de memòria *cache*, l'L1 té 4 MB, l'L2 té 32 MB i l'L3 té 256 MB en total [54]. Aquesta màquina va desplaçar a la màquina Gat i està pensada per executar programes paral·lelitzats amb OpenMP.

La següent màquina és Teen. **Teen** té **un** únic **node** amb **dos** processadors **Intel Xeon E5-2690**. Cada un d'aquests processadors té 8 *cores* i 16 *threads*, una memòria *cache* de 20 MB i una freqüència màxima de 3,80 GHz [55]. Aquesta màquina també està pensada per executar programes paral·lelitzats amb OpenMP, tot i que, al tractar-se d'una màquina més antiga, **els processadors són menys potents que el que utilitza Orca**.

La tercera d'aquestes màquines és Pop. **Pop** té **vuit nodes**, cada un amb **dos** processadors **AMD Second Generation Opteron 2210**. Aquest processador té 2 *cores* sense capacitat *multithreading* i una freqüència de 1,80 GHz [56]. Respecte a la memòria *cache*, presenta una L1 de dades i una L1 d'instruccions de 64 KB cada una per cada *core* i dues L2

d'1 MB [56]. Tot i que **aquest processador també és molt menys potent que el d'Orca**, com que aquesta màquina disposa de més d'un node, això la converteix en una opció interessant per executar programes paral·lelitzats amb MPI.

Existeix una quarta màquina anomenada **Roquer**. Aquesta màquina és similar a Orca i a Teen, ja que té un sol node i està pensada per executar programes en els que el paral·lelisme s'ha aconseguit amb l'eina Cuda. Com que no s'ha fet cap implementació del programa que faci servir l'API de Cuda, no s'ha considerat necessari descriure les prestacions d'aquesta màquina.

2.15 Sala dels Mac

La **Sala dels Mac** és el nom amb el que es va començar a anomenar al laboratori 205 del edifici E3 després de que s'hi incorporés el *hardware* que es detallarà a continuació. Aquest laboratori té **vint-i-un** ordinadors **iMac**. D'aquests, **setze** tenen un processador **Intel Core i9-10910** amb 10 *cores* i 20 *threads*, una memòria *cache* de 20 MB i una freqüència màxima de 5,00 GHz [57]. Els **cinc** restants tenen un processador **Intel Core i7-10700K** amb 8 *cores* i 16 *threads*, una memòria *cache* de 16 MB i una freqüència màxima de 5,10 GHz [58].

Tot i que el segon processador té pitjors especificacions que el primer, el rendiment d'un programa en qualsevol dels dos hauria de ser superior al que tindrà en el meu ordinador portàtil. Ara bé, el principal avantatge dels iMac d'aquest laboratori és que es pot repartir l'execució d'un codi paral·lel entre totes les màquines de manera similar a com es fa en la màquina Pop però amb l'agregat de que cada node té el seu propi **disc privat**. En altres paraules, és tracta d'un clúster de computadors que pot tenir fins a vint-i-un nodes. Per aquestes raons, la Sala dels Mac és una eina útil en l'avaluació del rendiment d'un programa, especialment si aquest ha estat paral·lelitzat utilitzant MPI.

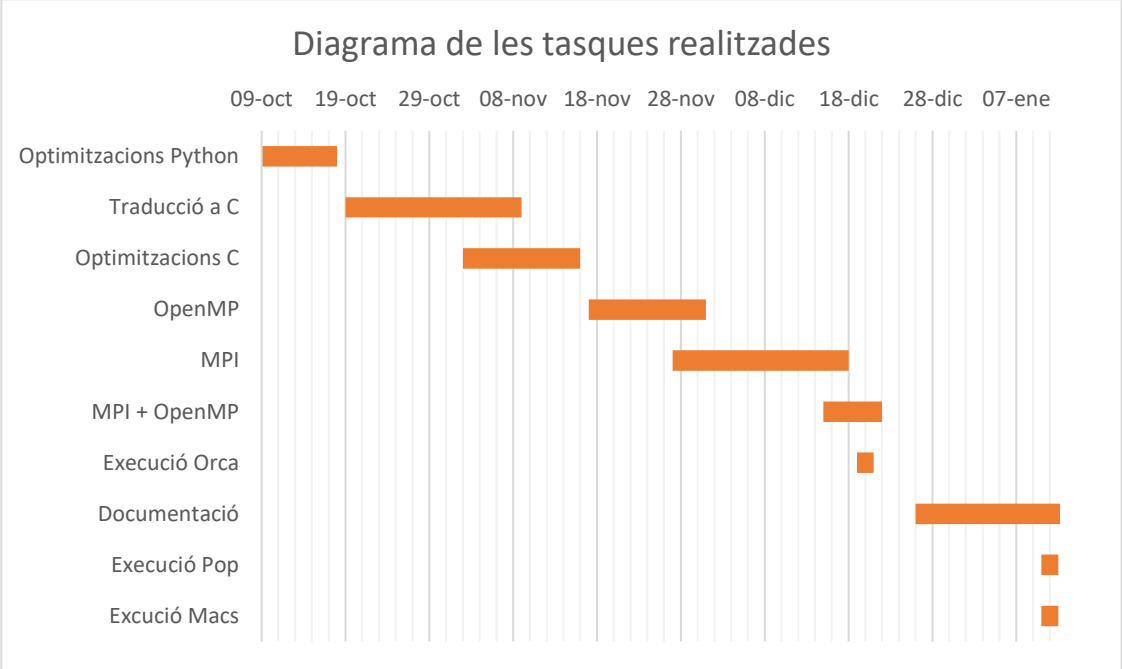
2.16 Taula resum recursos *hardware*

	Ordinador portàtil	**Orca**	**Pop**	**Sala dels Mac**	
Número de nodes	1	1	8	16	5
Número de processadors/node	1	1	2	1	1
Número de *cores*	6	64	2	10	8
Número de *threads*/*core*	2	2	1	2	2
Número de *threads*/node	12	128	2	20	16
Número total de *threads*	12	128	32	400	
Memòria cache (MB)	12	256	2	20	16
Freqüència màxim (GHz)	4,50	4,20	1,80	5,00	5,10

3 Disseny i implementació

3.1 Diagrama de les tasques realitzades

A continuació es mostra un diagrama de Gantt amb les tasques que han conformat aquest projecte i la seva durada aproximada. Les tasques es descriuen en profunditat en la resta de subapartats d'aquesta secció.



Gràfic 1. Diagrama de Gantt amb les tasques realitzades

3.2 Aïllat del codi necessari per portar a terme l'estudi de *nestedness*

El projecte que vaig desenvolupar durant les pràctiques curriculars consta d'una gran quantitat de procediments organitzats en diferents fitxers segons el tipus de funció que han de realitzar. De tots aquests procediments, tan sols una part és necessària per realitzar **l'estudi de *nestedness***. Així, per a que sigui més fàcil treballar sobre el codi rellevant pel desenvolupament del projecte actual, **s'han agrupat aquests procediments en un fitxer anomenat** `nestedness_test.py` que s'ha traslladat al repositori del projecte. Aquests procediments són els que permeten tractar els dos fitxers, crear les dues matrius d'abundàncies relatives, discretitzar-les i fer el test de *nestedness*.

3.3 Traducció del codi de Python a C

Degut a que cap de les eines per paral·lelitzar codi que s'han descrit en la secció de materials i metodologies està disponible en Python, ha estat necessari traduir el codi del programa al llenguatge C. Aquesta secció del projecte és la que ha requerit d'una major

inversió de temps degut a la manera en que està definit aquest segon llenguatge de programació.

Els procediments per **discretitzar una matriu** i per fer un **test de *nestedness*** han estat els menys complexes de traduir degut a que no presenten instruccions per realitzar E/S. Algunes de les funcions que en Python eren proporcionades per les llibreries sys o numpy, en C les he hagut de crear manualment. Així, he hagut d'implementar procediments per **inicialitzar una matriu amb zeros**; **per crear una randomització d'una matriu binaria** a partir del nombre de files, nombre de columnes i nombre d'uns; per **ordenar una llista amb valors de *nestedness*** i per **obtenir l'índex que un valor de *nestedness* concret ocupa en aquesta llista**. Per ordenar la llista s'ha recuperat una implementació de l'algorisme ***quicksort*** feta en una assignatura del grau [59] i per obtenir l'índex s'ha implementat una **cerca dicotòmica**. Per avaluar que el funcionament d'aquests procediments fos correcte, es van crear manualment petites matrius estàtiques amb valors que simulessin abundàncies relatives organitzades amb un patró perfectament *nested* o completament aleatori. La implementació original de l'algorisme per calcular el valor de *nestedness* d'una matriu binaria en C es mostra en l'annex 6.

La part més complexa ha estat el **tractament dels** dos **fitxers** tipus 'csv'. En Python, la llibreria pandas disposa de la funció `read_csv()` que permet llegir el contingut d'aquests fitxers i organitzar-lo en DataFrames [60]. Aquesta estructura de dades ofereix diversos mecanismes per accedir a les dades d'abundàncies absolutes que faciliten el càlcul de les abundàncies relatives i el seu emmagatzemament en la matriu corresponent. **En C, no existeixen estructures de dades com els DataFrames** [61]**, cosa que dificulta el tractament de les dades**. Es per aquest motiu que el disseny de les accions per tractar i emmagatzemar les dades així com les funcionalitats d'aquells procediments que es criden des d'aquestes accions difereixen bastant del codi equivalent en Python. De les funcions utilitzades en aquest apartat, m'agradaria destacar-ne dues. La primera és la funció `fgets()` que serveix per llegir un fitxer línia a línia [62]. La segona és la funció `strtok()` que serveix per tallar una cadena de caràcters (en aquest cas una línia del fitxer) en diferents fragments segons un delimitador [63].

Tot i que l'impacte d'aquesta secció del codi en el còmput global del programa és mínima degut a la quantitat de matrius que s'han de tractar en el test de *nestedness*, quan es treballa amb una única matriu sí que s'aprecia la quantitat de temps que el programa ha de dedicar al tractament del fitxer i l'emmagatzemament del seu contingut en les matrius d'abundàncies. Per aquesta raó, s'ha dedicat un cert temps a optimitzar la utilització d'aquestes eines. Per exemple, s'ha raonat detingudament el **delimitador idoni per separar els camps** de la capçalera dels fitxers o s'ha **minimitzat l'ús d'estructures de dades intermèdies**. A més, la dificultat més gran que he hagut d'afrontar en temps d'execució ha estat un error relacionat amb la funció `strtok()`. Degut a que C és un llenguatge de programació en el que es persegueix el màxim rendiment en l'accés a memòria, quan aquesta funció retorna un *token*, és a dir, un element de la línia que s'està tractant; no el retorna realment, sinó que crea un apuntador a aquest element. En conseqüència, quan es llegeix la següent línia del fitxer i es sobreescriu la variable en la que estava emmagatzemada la línia anterior, també es sobreescriuen tots els *tokens*. Aquest error és relativament senzill de solucionar però molt difícil de detectar si es ve de treballar amb un llenguatge de programació com Python i es desconeix el funcionament intern de la funció `strtok()`.

L'últim pas va ser redissenyar les **matrius d'abundàncies relatives** com a estructures **dinàmiques**. Aquest pas és recomanable en la majoria de projectes i, en aquest cas, a més és imprescindible. Això és degut a que en el SO Linux, la mida de la pila està

limitada a 8 MB [64]. Per tant, una matriu implementada amb memòria estàtica que tingui una mida superior generarà un error de tipus *segmentation fault* [64]. Tot i que la matriu d'individus, que és la matriu més gran, té una mida inferior a la meitat d'aquest límit; hi ha ocasions com, per exemple, en el pas de discretitzat de la matriu, en que, degut a la crida de procediments, s'està treballant simultàniament amb més de dues matriu d'aquestes dimensions i, per tant, si s'utilitzen estructures estàtiques, es desborda la pila.

Per assignar memòria dinàmica a les estructures que contindran les abundàncies relatives, s'ha utilitzat la funció `malloc()` [65] i per alliberar-la la funció `free()` [66]. Aquestes funcions s'han encapsulat dins d'unes altres funcions que he creat manualment per facilitar la reutilització d'aquests mecanismes d'assignació i alliberat de memòria dinàmica en funció del tipus de dades que contingui la matriu. Els vectors que emmagatzemen les interaccions de cada fila i de cada columna han d'estar inicialitzats a zero i, per tant, en lloc d'utilitzar la funció `malloc()` s'ha utilitzat la funció `calloc()` [67]. En el cas de les randomitzacions, les quals també han d'inicialitzar-se amb zeros, quan es treballa amb estructures dinàmiques aquesta acció es porta a terme amb la funció `memset()` [68].

3.4 Optimització seqüencial

Abans d'explicar qualsevol de les optimitzacions que s'han dissenyat, és necessari assenyalar que la implementació de l'algorisme detallat fa un ús intensiu dels recursos de la màquina. En conseqüència, si s'executa el codi d'aquest projecte en un ordinador portàtil, és molt recomanable que aquest portàtil estigui **endollat a la llum**. D'aquesta manera, l'energia subministrada es tradueix en una **millora** considerable en el **rendiment del programa**, com es mostra posteriorment en l'apartat de resultats i discussió.

Tot i que en el subapartat anterior s'han explicat unes poques optimitzacions fetes en la resta del programa, la gran majoria de les optimitzacions seqüencials s'han plantejat sobre la funció `calculate_nestedness_value()`, tant en Python com en C, ja que, com es detallarà en un subapartat posterior, és a on es concentra la quasi totalitat del temps d'execució del programa.

3.4.1 Python

En la versió de l'algorisme implementada amb el llenguatge de programació **Python** s'han plantejat **sis optimitzacions** diferents. Per comprendre bé en que consisteixen aquestes optimitzacions es mostra el següent fragment del codi de la funció, que es correspon amb les estructures iteratives anidades per calcular la primera i la tercera component, anomenades `first_isocline` i `third_isocline`.

```
for first_row in range(matrix.shape[0]):
    for second_row in range(matrix.shape[0]):
        if first_row < second_row:
            for col in range(matrix.shape[1]):
```

```
                if matrix[first_row][col] == 1 and matrix[second_row][col] == 1:
                    first_isocline += 1


for first_row in range(matrix.shape[0]):
    for second_row in range(matrix.shape[0]):
        if first_row < second_row:
            first_acum = second_acum = 0
            for col in range(matrix.shape[1]):
                first_acum += matrix[first_row][col]
                second_acum += matrix[second_row][col]
            third_isocline += min(first_acum, second_acum)
```

La **primera optimització** consisteix en **estalviar-se la última iteració dels bucles més externs** en cada una de les quatre estructures anidades de la funció. Això es degut a que només es realitzaran els càlculs necessaris per modificar cada un dels components de l'equació si es compleix la primera condició de les estructures i, en la última iteració del bucle més extern, aquesta condició no es complirà mai.

La **segona optimització** consisteix en **inicialitzar el valor del comptador dels segons bucles més externs amb el valor del comptador dels bucles més externs més un** i **eliminar els condicionals per avaluar que el primer comptador sigui menor que el segon comptador** en cada una de les quatre estructures anidades de la funció. D'aquesta manera s'aconsegueix evitar avaluar totes les iteracions que no compleixen la condició. Es interessant mencionar que aquesta optimització modifica la forma de les estructures anidades, fent que aquesta passi de ser quadrada a ser triangular. Això serà rellevant quan s'apliqui paral·lelisme amb OpenMP.

La **tercera optimització** consisteix en, per una banda, **ajuntar la primera i tercera estructura anidada** i, per l'altra, **ajuntar la segona i quarta estructura anidada**. Això permet reduir a la meitat el nombre d'estructures anidades. Aquesta modificació és possible perquè, encara que els càlculs fets sobre els components de l'equació són diferents, els tres bucles que conformen el parell d'estructures anidades son idèntics.

La **quarta optimització** consisteix en **substituir** tots els accessos a la matriu binaria del tipus `matrix[row][col]` **per** accessos del tipus `matrix[row, col]`. En Python, les matrius es poden implementar de diferents maneres. El mecanisme més senzill que existeix és com una **llista anidada**. És a dir, una matriu serà una llista en cada posició de la qual hi ha un altra llista. El primer tipus d'accés està pensat per accedir a matrius implementades d'aquesta manera. El problema és que, en aquest cas, per accedir a una posició de la matriu es crea una **nova llista**, s'hi copien tots els elements de la fila a la que es vol accedir i després s'accedeix a la posició d'aquesta nova llista que es troba en la columna corresponent. Aquest mecanisme és molt ineficient [69]. Per altra banda, una matriu també es pot implementar com un ***array* bidimensional** amb la llibreria **NumPy**. En aquest cas, no es crea cap *array* que sigui una copia de la fila que es vol accedir sinó un **punter a aquesta fila**, cosa que comporta un menor cost d'accés a una posició de la matriu [69].

La **cinquena optimització** afecta al càlcul del nombre d'interaccions d'una fila i al nombre d'interaccions d'una columna. Com s'ha explicat a la introducció, el nombre

d'interaccions(q_a o q_b) és el nombre d'elements diferents de zero (nombre de uns). En el codi sense optimitzar aquest valor es calcula diverses vegades per una mateixa fila o per una mateixa columna degut a la manera en que estan organitzades les estructures anidades. En conseqüència, aquesta optimització consisteix en **precalcular les interaccions de cada fila i les interaccions de cada columna i emmagatzemar-les en dos vectors respectivament** (un per les files i l'altre per les columnes). Les interaccions de la fila o de la columna que s'estigui avaluant es podran accedir en el vector corresponent utilitzant el mateix índex (fila o columna) amb el que s'estigui accedint a la matriu.

La **sisena optimització** consisteix en utilitzar la funció **zip() per accelerar el càlcul de les interaccions compartides entre files i entre columnes**. Com s'ha explicat a la introducció, el nombre d'interaccions compartides (q_ab) és el nombre d'elements diferents de zero (nombre de uns) que tenen en comú dues files o dues columnes. La funció **zip()** serveix per crear un *iterator* de tuples (es retorna d'un sol cop), a partir de dos elements iterables [70]. Per tant, rep com a entrada les dues files o les dues columnes que s'estan avaluant i crea una estructura de dades, en cada posició de la qual hi ha una tupla. Cada una d'aquestes tuples contindrà els dos elements que es troben en la posició corresponent de les dues files o les dues columnes. A més, per accelerar encara més el càlcul d'aquesta secció s'utilitza l'operador lògic '**and**' en lloc de l'operador aritmètic '*', ja que les operacions lògiques son més ràpides que les multiplicacions.

Després d'aplicar aquestes optimitzacions, el fragment de codi que s'ha mostrat a l'inici del subapartat queda de la següent manera:

```
for first_row in range(matrix.shape[0] - 1):
    for second_row in range(first_row + 1, matrix.shape[0]):
        for col in range(matrix.shape[1]):
           if matrix[first_row, col] == 1 and matrix[second_row, col] == 1:
               first_isocline += 1
                  third_isocline += min(sum_rows[first_row], sum_rows[second_row])
```

No s'ha inclòs la **sisena optimització** en la versió definitiva perquè, com es mostra en l'apartat de resultats i discussió, **produeix un augment substancial del temps d'execució** del programa quan es calcula el valor de *nestedness* de la matriu d'individus. El codi Python complet de la funció calculate_nested_value() amb totes les optimitzacions que han millorat el rendiment del programa s'inclou en l'annex 4.

3.4.2 PyPy

S'ha creat un ***script*** anomenat `executable_pypy3.sh` que executi el projecte utilitzant l'intèrpret **pypy3** d'idèntica manera amb que es fa amb l'intèrpret python3. Tot i les expectatives dipositades en aquesta eina, el **rendiment** del programa ha estat **molt pitjor** que el que ha tingut amb l'intèrpret python3. Per tant, s'ha descartat la utilització de PyPy com a eina per assolir els objectius del projecte.

3.4.3 C

Qualsevol versió del codi d'aquest projecte que s'implementi amb el llenguatge de programació **C** i es vulgui executar, es compilarà sempre amb l'opció **-O3**. Tot i que l'ús d'aquesta eina augmenta el temps de compilació, aquest fet no té cap mena d'importància en aquest cas, ja que l'objectiu essencial del projecte és reduir el temps d'execució del programa. **La millora obtinguda compilant amb tercer nivell d'optimització és molt alta** i també es mostra en l'apartat d'avaluació.

De les sis optimitzacions seqüencials descrites en el subapartat de Python n'hi ha quatre que també s'han pogut reutilitzar pel codi en C. Aquestes son:

- **Estalvi de la ultima iteració del bucle més extern** en les quatre estructures anidades.
- **Inicialitzar el valor del comptador dels segons bucles més externs amb el valor del comptador dels bucles més externs més un** i **esborrar els condicionals per avaluar que el primer comptador sigui menor que el segon comptador** en les quatre estructures anidades.
- **Ajuntar la primera amb la tercera estructura anidada** i **la segona amb la quarta estructura anidada**.
- **Precalcular les interaccions de cada fila** i **les interaccions de cada columna** i emmagatzemar-les en dos vectors respectivament.

L'optimització feta en Python relacionada amb l'accés a una posició de la matriu no cal aplicar-la perquè en C els accessos a memòria sempre es fan amb el mateix mecanisme que, en Python, proporciona la llibreria NumPy. És a dir, no es fa cap copia de la fila per a després accedir a l'element utilitzant la columna com si fos l'índex, sinó que es crea un apuntador a la fila corresponent. Com ja s'ha explicat, aquest mecanisme és molt més eficient i, per tant, més coherent amb la filosofia de C. Tampoc s'ha provat d'utilitzar la funció zip() o una d'equivalent perquè no hi ha cap rutina en el repertori de les llibreries típiques de C que realitzi aquesta funcionalitat.

A més, també s'han avaluat tres noves optimitzacions aportades per primer cop en la implementació del programa en C que cal descriure en més profunditat.

La **primera nova optimització** consisteix en avaluar el rendiment del programa en funció de amb quin **tipus de dades es defineix la matriu binaria**. Fins aquest moment, les matrius d'abundàncies relatives s'havien guardat com a matrius reals (tipus `double`) i les matrius binaries s'havien guardat com a matrius d'enters (tipus `int`). En el SO utilitzat, un element definit amb el tipus `int` tindrà una mida de 32 bits (4 Bytes). Ara bé, qualsevol element de la matriu binària només pot tenir dos valors possibles, '1' o '0', i, per representar aquests dos valors, es suficient amb 1 bit. Durant l'execució del programa, el processador copiarà un subconjunt de valors de la matriu binaria a nivells més baixos de *cache* per a que l'accés a aquestes dades sigui més ràpid. Quant menor sigui la mida d'aquestes dades, més elements de la matriu es podran copiar a jerarquies més baixes de memòria. Per tant, és raonable plantejar-se que el tipus de dades més adequat des de el punt de vista de l'optimització de codi és aquell que tingui la mida més petita. Així, el tipus `short` té una mida de 16 bits (2 Bytes) i els tipus `char` i `bool` tenen tots dos una mida de 8 bits. Curiosament, per la matriu de vertebrats tots aquests tipus de dades han donat un pitjor rendiment del programa que el tipus `int` i, per la matriu d'individus, únicament el tipus `short` ha resultat en una millora del rendiment del programa (apartat de resultats i discussió). Com

que, quan una optimització millora el rendiment del programa per una de les matrius però l'empitjora per l'altra, es dona prioritat a la matriu més gran (matriu d'individus), les matrius binaries han passat a ser definides amb el tipus `short`.

La **segona nova optimització** consisteix en utilitzar la **matriu transposada** de la matriu binaria per a calcular les dues components (sumatoris) de l'equació que s'obtenen a partir de les columnes. A la memòria de l'ordinador, les matrius estan ordenades per files. És a dir, primer està emmagatzemada la primera fila, després la segona i així successivament. Per tant, quan es desplaça una subconjunt de la matriu a nivells més baixos de la jerarquia de memòria, l'accés per columnes generarà moltes més fallides de *cache* que l'accés per files. **Si es combina l'accés per files utilitzant la matriu binària amb l'accés per columnes utilitzant la matriu transposada de la matriu binaria, s'aconsegueix reduir el nombre de fallides de *cache***. La funció per transposar una matriu ha estat implementada manualment.

Finalment, **la tercera** i última **nova optimització** consisteix en **agrupar el càlcul de les interaccions de les files i el càlcul de les interaccions de les columnes** que es fa al principi, en la mateixa estructura anidada.

3.5 Prerequisits per l'optimització paral·lela

La **llei d'Amdahl** és una llei proposada per Gene Amdahl en l'àmbit de les ciències de la computació que exposa que la millora obtinguda en el rendiment d'un sistema degut a l'alteració d'un dels seus components està limitada per la fracció de temps que s'utilitza aquest component [71]. Aquest enunciat implica que, per estimar si el paral·lelitzat d'una secció del codi pot millorar el rendiment d'un programa de manera significativa, **s'ha de valorar l'impacte** que aquesta secció té **en el temps d'execució seqüencial del programa**.

Com s'ha explicat en subapartats anteriors, la secció del codi que representa la major part del temps d'execució del programa en el projecte que ens ocupa és la funció `calculate_nested_value()`. Ara bé, per decidir si és interessant implementar optimitzacions paral·leles, no n'hi ha prou amb saber aquesta informació, sino que és necessari conèixer quin percentatge del temps d'execució del programa representa aquesta funció. És a dir, cal fer el ***profiling*** del codi del programa. En total, s'han generat **vuit fitxers de *profiling***. Aquests fitxers son el resultat de tres valoracions, per cada una de les quals hi ha dues opcions possibles.

Primer, s'avalua la funció `calculate_nested_value()` en la seva primera versió (sense aplicar les optimitzacions seqüencials) i sent executada una única vegada. En aquest primer cas, s'han recopilat estadístiques tant del temps d'execució del programa quan es treballa amb la **matriu de vertebrats** com quan es treballa amb la **matriu d'individus**. El que es vol valorar és com el volum de dades amb els que es treballa afecta al percentatge del temps d'execució que ocupa aquesta funció, ja que el paral·lelisme acostuma a donar molt bons resultats quan la secció que es vol executar de forma paral·lela utilitza grans volums de dades. Així, si es treballa amb la **matriu de vertebrats** la funció `calculate_nested_value()` representa un **85,84 %** del temps d'execució total mentre que si es treballa amb la **matriu d'individus** representa un **99,80 %**. Aquesta diferència en els percentatges també està relacionada amb que el procés de creació de la **matriu de vertebrats** és molt més complexa que el de creació de la **matriu d'individus** i això repercuteix en el temps d'execució. Tot i

així, aquestes estadístiques preveuen que el paral·lelisme donarà millors resultats per la **matriu d'individus** que per la **matriu de vertebrats**.

Seguidament, s'han recopilat estadístiques de l'execució del programa comparant la funció `calculate_nested_value()` **sense aplicar cap de les optimitzacions seqüencials** amb la versió resultant **després d'aplicar totes les optimitzacions seqüencials**. Novament, la funció es crida una única vegada. Aquest anàlisi s'ha fet amb l'objectiu de determinar si les optimitzacions seqüencials redueixen el percentatge del temps d'execució que representa aquesta secció del codi fins al punt en que no es pot esperar que el paral·lelisme doni bons resultats. El que s'observa és que, si es treballa amb la **matriu de vertebrats** el temps d'execució que representa la funció `calculate_nested_value()` optimitzada es redueix fins a un **77,89 %** mentre que si es treballa amb la **matriu d'individus** es redueix fins a un **98,91 %**. Aquestes estadístiques reforcen les conclusions extretes de les descrites en el paràgraf anterior. És a dir que, encara que la funció `calculate_nested_value()` s'hagi optimitzat, el paral·lelisme pot donar molt bons resultats per **la matriu d'individus** però no tant bons per la matriu de vertebrats.

Finalment, la tercera consideració que s'ha fet és que cal recopilar estadístiques tant del impacte que la funció `calculate_nested_value()` té en el programa quan només es calcula el valor de *nestedness* de la matriu avaluada, com quan es calcula el valor de *nestedness* d'aquesta i de totes les seves randomitzacions per fer el test de *nestedness*. És a dir, en el primer cas, la funció es crida **una** única vegada, mentre que, en el segon cas, es crida tantes vegades com indiqui el **número de randomitzacions més una**. Aquí es tracta de valorar com el paral·lelisme pot impactar en el rendiment del programa quan es varia el número de randomitzacions o, el que és el mateix, de crides a la funció que es vol paral·lelitzar. Aquesta última valoració és especialment rellevant perquè, com s'ha explicat en la introducció, el valor de *nestedness* d'una matriu per si sol no aporta cap informació, sino que s'ha de comparar amb el d'almenys mil randomitzacions. Per tant, no té gaire sentit executar aquesta funció una única vegada, sinó que, si es volen obtenir resultats amb contingut real, s'ha d'executar un mínim de mil una vegades. D'aquesta manera, si es fa el **test de *nestedness* amb mil randomitzacions** sobre la **matriu de vertebrats**, el temps d'execució que representa la funció `calculate_nested_value()` és del **99,73 %** del total mentre que si es fa sobre la **matriu d'individus**, el temps d'execució és del **99,99 %**.

En conclusió, no es pot garantir que l'ús del paral·lelisme generi sempre una millora en el rendiment del programa quan es vulgui calcular el valor de *nestedness* d'una matriu binaria. Tot dependrà del volum de dades utilitzat. Per altra banda, **si el paral·lelisme s'aplica sobre un cas real**, com el que representa el test de *nestedness* que es va voler fer sobre la matriu de vertebrats i sobre la matriu d'individus, **la millora en el rendiment del programa hauria de ser absoluta**. L'única qüestió és, doncs, com implementar les optimitzacions paral·leles més adequades.

3.6 Disseny de l'optimització paral·lela

Després d'aplicar les optimitzacions seqüencials descrites en el subapartat anterior del mateix nom i amb les que s'ha obtingut una millora en el temps d'execució del programa, la funció `calculate_nested_value()` queda organitzada en **tres estructures iteratives anidades**. La forma típica de paral·lelitzar estructures d'aquest tipus és **assignar grups d'iteracions del bucle més extern a cada una de les unitats d'execució** amb les que

s'estigui treballant (*threads* o processos). El mecanisme concret per aconseguir això dependrà de les característiques de l'estructura en qüestió i de l'eina utilitzada. Ara bé, a grans trets, cada unitat d'execució tindrà la seva pròpia variable de gestió del bucle privada (en aquest cas `row`, `col`, `first_row` o `first_col`) i realitzarà tantes iteracions del bucle com indiqui el valor resultant de dividir el valor límit (`num_rows` o `num_cols`) entre el número de *threads* o processos creats. **Les variables de les que tan sols se'n llegeix el contingut mai comporten cap problema de dependències** i cal parar especial atenció a aquelles de les que se'n modifica el contingut a partir del contingut que s'ha calculat en una iteració anterior del bucle.

La primera estructura anidada és el doble bucle que serveix per calcular les interaccions de les files i les interaccions de les columnes i que es mostra a continuació:

```
for (row = 0; row < num_rows; row++) {
	for (col = 0; col < num_cols; col++) {
		sum_rows[row] += matrix[row][col];
		sum_cols[col] += transposed_matrix[col][row];
	}
}
```

En aquesta estructura anidada es recorre la matriu binaria que la funció `calculate_nested_value()` rep per paràmetre i se n'acumula el contingut de cada fila en la posició corresponent del vector `sum_rows`. Simultàniament, es recorre la matriu transposada d'aquesta matriu binaria i se n'acumula el contingut de cada fila (de cada columna respecte a la matriu no transposada) en la posició corresponent del vector `sum_cols`.

En aquest cas, hi ha quatre variables només de lectura, que són `num_rows`, `num_cols`, `matrix` i `transposed_matrix`. Aquestes variables no poden tenir cap problema de dependències, així que no cal preocupar-se'n. Per altra banda, les variables `row` i `col` són variables de comptatge que s'encarreguen de gestionar els seus respectius bucles i, per tant, tenen dependències internes. Ara bé, ja s'ha explicat abans que, en aquest tipus de variables les dependències es resolen fent que cada *thread* o procés tingui la seva pròpia còpia privada de la variable. Després, si el paral·lelisme s'aplica al bucle més extern, la variable `sum_rows` no té cap dependència interna, ja que cada iteració en modificarà una posició diferent. Finalment, en la variable `sum_cols` cada posició del vector és la suma d'un conjunt d'elements, cada un dels quals es llegirà en una iteració diferent del bucle més extern. Per tant, caldrà llegir el valor resultant de sumar els elements de totes les iteracions anteriors. Això la converteix en la variable amb les dependències més complexes i, depenent de l'eina utilitzada, aquestes seran molt difícils de solucionar.

No obstant, recordem que aquesta estructura anidada no existia en el codi original, sinó que és el resultat de vàries optimitzacions seqüencials que s'han fet posteriorment. Per tant, es possible desfer la última d'aquestes optimitzacions seqüencials i desplegar aquesta estructura en les dues estructures anidades següents:

```
for (row = 0; row < num_rows; row++) {
	for (col = 0; col < num_cols; col++) {
```

```
            sum_rows[row] += matrix[row][col];
      }
}

for (col = 0; col < num_cols; col++) {
      for (row = 0; row < num_rows; row++) {
            sum_cols[col] += transposed_matrix[col][row];
      }
}
```

En aquestes dues estructures anidades ja no existeix el problema de dependències que s'ha descrit anteriorment pel vector `sum_cols`. **Si el paral·lelisme s'aplica sobre els bucles més externs, les variables `sum_rows` i `sum_cols` no tindran problemes de dependències perquè en cada nova iteració del bucle més extern s'accedirà a una posició diferent d'aquests vectors**. Aquesta modificació serà imprescindible per provar algunes de les optimitzacions paral·leles (com es veurà en subapartats posteriors), però, en d'altres, també pot ser recomanable, ja que les dues estructures obtingudes són molt més senzilles de paral·lelitzar que l'anterior.

Per últim, s'han de tenir en compte dues coses. D'una banda, aquesta estructura (o estructures) és **quadràtica**, és a dir, està composada per dos bucles anidats i, per tant, el nombre d'iteracions que conté és ínfim en comparació amb el de les dues estructures posteriors, que són cúbiques. Per altra banda, com s'ha assenyalat abans, aquesta estructura és el resultat d'aplicar tot un conjunt d'optimitzacions seqüencials i, per tant, serà bastant ràpida. En conseqüència, **no s'hauria d'esperar una gran millora en el rendiment del programa al paral·lelitzar les dues estructures quadràtiques** (si és que n'hi hagués).

Les dues estructures anidades que s'explicaran a continuació són conceptualment idèntiques i, per tant, les dependències internes seran les mateixes i el procediment per aplicar-hi paral·lelisme serà equivalent. La primera serveix per realitzar els càlculs que es corresponen amb la primera i tercera component de l'equació per obtenir el valor de *nestedness* d'una matriu binaria (descrita en la introducció), mentre que la segona serveix per fer els càlculs de la segona i de la quarta component de l'equació. En altres paraules, totes dues estructures anidades fan els mateixos càlculs, però **la primera treballa sobre les files i la segona treballa sobre les columnes**. El codi de ambdues és el següent:

```
for (int first_row = 0; first_row < num_rows - 1; first_row++) {
      for (int second_row = 0; second_row < num_rows; second_row++) {
            if (first_row < second_row) {
                  for (col = 0; col < num_cols; col++) {
                        first_isocline     +=     matrix[first_row][col]     &
                        matrix[second_row][col];
                  }
                  if (sum_rows[first_row] < sum_rows[second_row]) {
                        third_isocline += sum_rows[first_row];
```

```
                        } else {
                                third_isocline += sum_rows[second_row];
                        }
                }
        }
}


for (int first_col = 0; first_col < num_cols - 1; first_col++) {
        for (int second_col = 0; second_col < num_cols; second_col++) {
                if (first_col < second_col) {
                        for (row = 0; row < num_rows; row++) {
                                second_isocline += transposed_matrix[first_col][row] &
                                transposed_matrix[second_col][row];
                        }
                        if (sum_cols[first_col] < sum_cols[second_col]) {
                                fourth_isocline += sum_cols[first_col];
                        } else {
                                fourth_isocline += sum_cols[second_col];
                        }
                }
        }
}
```

El procediment que s'ha seguit per avaluar les dependències internes en les variables d'aquestes dues estructures ha estat el mateix que en el cas anterior. Primer de tot, s'identifiquen les variables de lectura. En la primera estructura, aquestes són `num_rows`, `num_cols`, `matrix` i `sum_rows`. En el cas de la segona, les variables de lectura són `num_cols`, `num_rows`, `transposed_matrix` i `sum_cols`. Com ja s'ha explicat, les variables només de lectura no tenen dependències internes i, per tant, no cal preocupar-se'n.

D'entre les variables d'escriptura, s'han identificat les variables de comptatge `first_row`, `second_row` i `col` en la primera estructura i les variables `first_col`, `second_col` i `row`, en la segona. Les variables `first_row` i `first_col` són les variables de control del bucle més extern i, per tant, en cada iteració d'aquest bucle tindran un valor diferent. Les variables `second_row`, `second_col`, `row` i `col`, en canvi, s'encarreguen de gestionar els bucles interns, als que no s'aplicarà cap mena de paral·lelisme. Tot i així, les dependències continuen existint ja que en cada iteració del bucle més extern es faran una gran quantitat de lectures i escriptures en aquestes variables, que provocaran conflictes entre les diferents unitats d'execució. Per tant, cada *thread* o procés haurà de treballar amb la seva còpia privada de totes aquestes variables.

Finalment, queden les variables `first_isocline` i `third_isocline` en la primera estructura i les variables `second_isocline` i `fourth_isocline` en la segona que, al tractar-se de variables d'escriptura, cada *thread* o procés també en tindrà una còpia privada. Ara bé,

s'ha de tenir en compte que aquestes variables acumulen tots els càlculs intermedis que es fan en cada una de les dues estructures anidades. Dit d'una altra manera, no només han de llegir el valor que se'ls ha escrit en la iteració anterior, sinó el valor de totes les iteracions fetes fins al moment. Això significa que no n'hi ha prou amb assegurar-se de que cada unitat d'execució només pugui modificar el contingut de la còpia de la variable que li pertoca, com en els cassos anteriors, sinó que caldrà afegir un últim pas de **reducció**. Aquest pas consisteix en sumar el contingut de totes les còpies d'aquesta variable amb les que ha treballat cada *thread* o procés i la seva aplicació és típica en comptadors o acumuladors que, quan s'hagin executat totes les iteracions del bucle, n'hagin de contenir el resultat final.

Aquestes dues estructures anidades tenen forma **cúbica**, cosa que preveu uns millors resultats quan s'executin de forma paral·lela que la estructura quadràtica que s'ha explicat anteriorment. No obstant, s'ha de tenir en compte que el nombre de crides a la funció `calculate_nested_value()` dependrà del nombre de randomitzacions que es vulguin tractar, que no hauria de ser inferior a mil. Per tant, tots els mecanismes per aconseguir el paral·lelisme (fork-join, pas de missatges,...) s'executaran al menys mil una vegades. Es per aquest motiu que, com denoten els resultats, **només s'hauria d'esperar una gran millora al paral·lelitzar aquestes estructures cúbiques quan es treballi amb grans volums de dades**, cosa que, en el nostre estudi, es compleix per les columnes de les matrius, però no per les files.

La funció `calculate_nested_value()` no presenta més estructures iteratives i, tenint en compte els resultats obtinguts en el pas de *profiling*, es podria concloure prematurament que no hi ha cap més secció de codi en la que l'aplicació de paral·lelisme es tradueixi en una millora en el rendiment del programa. Ara bé, si el que s'està portant a terme és el test de *nestedness*, cal recordar que aquesta funció no es cridarà una única vegada, sinó tantes vegades com randomitzacions s'hagi especificat que es volen crear més una. Dit d'una altra manera, tot i que es cert que la funció `calculate_nested_value()` representa la quasi totalitat del temps d'execució del programa, també es cert que, en el test de *nestedness*, la quasi totalitat de les crides a aquesta funció es faran des una estructura iterativa que es troba en la funció `generate_nested_values_randomized()`, que és la següent:

```
for (int pos = 0; pos < num_randomized_matrices; pos++) {

	initialize_matrix_shorts_zeros(randomized_matrix, num_rows, num_cols);

	generate_randomized_matrix(randomized_matrix,	num_rows,	num_cols,
	num_ones);

	nested_values_randomized[pos]	=	calculate_nested_value(randomized_matrix,
	num_rows, num_cols);

}
```

Aquest bucle té dues característiques que asseguren que serà en el que el paral·lelisme donarà més bons resultats. Primer, si el paral·lelisme s'aplica sobre aquest bucle, els mecanismes per aconseguir que s'executi de forma paral·lela (fork-join, pas de missatges,...) es generaran una única vegada, cosa que és molt més eficient que el que es produeix en les estructures anidades de la funció `calculate_nested_value()`.

Segon, el bucle s'encarrega d'inicialitzar les randomitzacions amb zeros, escriure'n un número fix d'uns en posicions aleatòries i calcular-ne el valor de *nestedness*. Es a dir, aquest bucle no realitza càlculs com fan les estructures anidades de la funció

`calculate_nested_value()`, sinó que repeteix un conjunt de crides a funcions tantes vegades com randomitzacions s'hagi especificat que es volen utilitzar en el test de *nestedness*. Això comporta que cada iteració tingui menys dependències amb les altres, que les que tenen les estructures anidades descrites anteriorment.

Per entrar amb més de detall, les variables `num_randomized`, `randomized_matrix`, `num_rows`, `num_cols` i `num_ones`, són variables només de lectura i, per tant, no tenen problemes de dependències. La variable `nested_values_randomized` és el vector que emmagatzemarà els valors de *nestedness* de les randomitzacions a mesura que es vagin calculant i, per tant, es tracta d'una variable d'escriptura. Ara bé, cada iteració del bucle modificarà una posició del vector diferent, de manera que no existeix cap dependència interna. Així, la única variable amb la que s'haurà de tenir cura, és la variable de control del bucle `pos`, el contingut de la qual s'incrementa amb cada iteració. De nou, cada *thread* o procés haurà de tenir la seva pròpia còpia privada d'aquesta variable.

3.7 Implementació de l'optimització paral·lela

3.7.1 Python

Degut a la inversió en temps que es va preveure que podria suposar la traducció de tot el codi del fitxer `nestedness_test.py` a C, pas necessari per a poder utilitzar les eines descrites en l'apartat de materials i metodologies, abans es va voler saber el guany potencial que podria aportar el paral·lelisme a l'algorisme en qüestió. Per tant, es va crear un *script* anomenat `executable_python3_parallel.sh` que permetés executar el programa de forma paral·lela havent de modificar el mínim possible el codi en Python. El bucle que es va paral·lelitzar va ser el bucle encarregat d'inicialitzar les randomitzacions i calcular-ne el valor de *nestedness*, ja que, com s'ha explicat en el subapartat anterior, és el que s'esperava que donés els millors resultats. Així, l'únic canvi que es va fer sobre el codi va ser que el valor de la variable `num_randomized` en lloc d'estar fixat en el propi codi, el decideixi el programador com un dels arguments d'entrada del programa. El codi de l'*script* es mostra en l'annex 5.

El nombre d'unitats d'execució que es van crear amb l'execució d'aquest *script* va ser el nombre total de fils d'execució *hardware* que s'han assignat al SO Ubuntu virtualitzat menys un (que es deixa lliure per a administrar el SO). Per tant, es van crear **5 unitats d'execució**. Per aquesta quantitat, **l'*speedup* aconseguit va ser 4,1**. Aquest resultat va ser una premonició de l'elevat potencial de millora que podia arribar proporcionar el paral·lelisme quan s'utilitzessin eines més sofisticades com OpenMP i MPI i màquines més potents que un ordinador portàtil.

Cal mencionar que, per a que aquest *script* fos correcte des d'un punt de vista formal, caldria afegir una fase de reducció. No obstant, com que els resultats globals son fàcilment extrapolables a partir dels resultats obtinguts i a que l'objectiu d'implementar aquest *script* va ser comprovar si l'optimització paral·lela era una camí prometedor, es va considerar innecessari afegir aquest pas, com si que s'ha fet per les optimitzacions paral·leles que s'han proposat en C.

3.7.2 C

La utilització d'eines més sofisticades i intrusives que el mètode descrit pel codi en Python, ofereix moltes més opcions alhora d'aconseguir que el programa s'executi de forma paral·lela. Algunes d'aquestes opcions seran exclusives per OpenMP o MPI, mentre que d'altres seran comunes per totes dos. Totes les alternatives que s'han implementat es descriuen en els següents subapartats.

3.7.2.1 OpenMP

La primera possibilitat que s'ha plantejat és dividir el codi per seccions i executar-les de forma paral·lela. Aquestes dues seccions són els càlculs fets respecte les files i els càlculs fets respecte les columnes de la matriu binaria. Per paral·lelitzar el codi d'aquesta manera, cal desfer la última optimització seqüencial plantejada i pre calcular les interaccions de les files i les interaccions de les columnes en estructures anidades separades. És a dir, cal desplegar la primera estructura anidada en les dues estructures anidades que ja s'han descrit en la fase de disseny de les optimitzacions paral·leles, cada una de les quals pertanyerà a una secció diferent. Tot i que la funció `calculate_nested_value()` queda completament dividida en aquestes dues seccions, **la millora en el temps d'execució que comporta el paral·lelitzat per seccions depèn de la forma de la matriu**. Si la matriu fos quadrada, el temps d'execució es podria reduir fins a gairebé la meitat. No obstant, **les matrius de les que hem fet el test de *nestedness* tenen una forma molt rectangular i, per tant, no és esperable obtenir una reducció important en el temps d'execució**, ja que per obtenir els valors de les dues components de l'equació relacionades amb les columnes, es requereixen moltes més iteracions que per obtenir els valors de les dues components relacionades amb les files.

Les següents opcions que s'han plantejat aborden el paral·lelitzat del bucle més extern de cada una de les estructures anidades de la funció `calculate_nested_value()` (tres en cas de que es mantingui la última optimització seqüencial i quatre en cas de que s'esborri) i del bucle per inicialitzar les randomitzacions i calcular-ne el valor de *nestedness*.

Abans d'explicar com es paral·lelitzen aquestes estructures, és important destacar que **OpenMP només pot paral·lelitzar bucles que siguin canònics**. Un bucle és canònic quan compleix tres requisits: Ha d'estar gestionat per una única variable de comptatge, la condició del bucle ha de ser una comparació entre la variable de comptatge i un valor numèric (que pot estar o no emmagatzemat en una variable o constant) i l'operació d'increment de la variable de comptatge ha de ser una suma, una resta o un producte [72]. Totes les estructures anidades compleixen les tres condicions excepte les estructures per calcular les components de l'equació, les quals no compleixen la segona, ja que la condició del bucle és una comparació entre la variable de comptatge i una operació. Per solucionar aquest inconvenient, s'emmagatzema prèviament el resultat d'aquesta operació en una nova variable anomenada `total_rows` o `total_cols` i és modifica la condició per a que sigui una comparació entre la variable de comptatge i aquesta nova variable. El bucle més extern quedaria de la següent manera:

```
for (first_row = 0; first_row < total_rows; first_row++)
```

```
for (first_col = 0; first_col < total_cols; first_col++)
```

La directiva d'OpenMP per executar un bucle de forma paral·lela és la següent:

```
#pragma omp parallel for
```

Ara bé, com s'ha explicat en l'apartat de descripció del paral·lelisme, cada estructura iterativa té el seu propi conjunt de variables, cada una de les quals pot tenir dependències internes. Per solucionar aquestes dependències, OpenMP permet afegir a aquesta directiva tot un seguit de clàusules. Per facilitar la comprensió de la metodologia aplicada, s'ha considerat pertinent explicar primer totes les clàusules utilitzades i després es proporcionarà, a mode d'exemple, la directiva completa que proporciona els millors resultats. Degut a que totes les directives són semblants, aquesta explicació és extrapolables a la resta d'estructures iteratives.

En primer lloc, s'afegeix la clàusula `private()`. Aquesta clàusula serveix per indicar quines seran les variables de les quals es crearà una còpia privada per cada *thread*. Les variables englobades sota aquesta clàusula seran, generalment, les variables de comptatge, ja sigui les que s'encarreguen de gestionar el bucle extern a paral·lelitzar o les que s'encarreguen de gestionar els bucles interns en les estructures anidades.

Després s'afegeix la clàusula `shared()`, que té la finalitat contrària, assenyalar aquelles variables a les que tots els *threads* podran accedir sense que es produeixi cap conflicte. Les variables englobades sota aquesta clàusula són les variables només de lectura i els vectors als que en cada iteració del bucle s'accedeix a una posició diferent.

A continuació, trobarem la directiva `reduction()`, que com el seu nom indica, serveix per aplicar una operació de reducció. Sota aquesta clàusula s'hi col·loquen generalment variables com comptadors o acumuladors. En el nostre cas, s'utilitza per sumar les mateixes posicions dels vectors `sum_cols` intermedis de cada *thread* (en l'estructura anidada per precalcular conjuntament les interaccions de les files i les interaccions de les columnes) i per sumar els valors parcials de les quatre components de l'equació per calcular el valor de *nestedness* (en les dues estructures anidades cúbiques).

Seguidament, es recomanable incorporar la clàusula `default(none)`, que no s'aplica a cap variable en concret, sinó que obliga a afegir totes les variables de l'estructura iterativa a dins d'alguna de les altres clàusules. D'aquesta manera es força al programador a conèixer les dependències internes de cada variable per a poder classificar-les correctament. Si no s'afegís aquesta clàusula, no caldria afegir tampoc la clàusula `shared()`, ja que es tracta de l'opció a la que les variables pertanyen per defecte.

Per últim, trobem la clàusula `schedule()`, la qual té una finalitat diferent de les anteriors. Aquesta opció serveix per indicar el mecanisme mitjançant el qual es repartiran les iteracions del bucle entre els *threads*. El format `schedule(static)` serveix per assignar blocs de `num_iteracions / num_threads` a cada *thread* a l'inici de l'execució i és el que hauria de donar els millors resultats quan el bucle té forma quadrada. Per altra banda, el format `schedule(static, 1)` serveix per assignar blocs de una iteració a cada *thread* a l'inici de l'execució i està pensat per bucles que tenen forma triangular. Finalment, el format `schedule(dynamic)` serveix per assignar una a una les iteracions a mesura que cada *thread*

vagi acabant. Aquest format és el més recomanable per bucles que tinguin una forma completament irregular. És interessant fer proves amb els diferents formats d'aquesta clàusula, especialment en les dues estructures anidades cúbiques si es vol mantenir la segona optimització seqüencial.

A continuació es presenta la directiva d'OpenMP seleccionada per paral·lelitzar el bucle per inicialitzar les randomitzacions i calcular-ne el valor de nestedness:

```
#pragma omp parallel for private(pos) shared(num_randomized_matrices, randomized_matrices, num_rows, num_cols, num_ones, nested_values_randomized) default(none) schedule(dynamic)
```

Es interessant observar que no és necessari cap pas de reducció i que la major part de les variables són compartides. La implementació del paral·lelitzat d'aquest bucle ha requerit una modificació una mica més important del codi que la resta. Això es degut a que aquest bucle si modifica el contingut de les matrius binàries, en aquest cas, les randomitzacions. Es podria pensar que és suficient amb que les randomitzacions s'estableixin sota la directiva `private()` però, tot i que aquestes matrius sí que s'inicialitzen dins del bucle, per motius d'eficiència es defineixen i se'ls i assigna i allibera memòria a fora d'aquest. Explicat d'una altra manera, es defineix una sola randomització, se li assigna memòria dinàmica, s'inicialitza amb zeros, s'hi escriuen els uns en posicions aleatòries i se'n calcula el valor de *nestedness*. Després, aquest procés es repeteix tantes vegades com randomitzacions es vulguin generar però des de la fase d'inicialització amb zeros.

Per a solucionar aquesta limitació, s'han de definir tantes randomitzacions com *threads* es vulguin crear. Per fer això s'utilitza un vector anomenat `randomized_matrices` de mida `omp_get_max_threads()` [73]. En cada posició d'aquest vector s'emmagatzemarà el punter a la primera posició de la randomització. Així, cada fragment del bucle assignat a un *thread* tindrà la seva pròpia randomització que s'anirà reinicialitzant amb cada nova iteració d'aquest fragment. Per indexar el vector en cada conjunt d'iteracions s'utilitza la funció `omp_get_thread_num()`, que indica el número del *thread* que està executant el codi en aquell moment [74].

Tot i que el bucle té forma quadrada, el repartiment dinàmic de les iteracions és el que ha donat el menor temps d'execució, com es mostra en l'apartat de resultats i discussió. A més, **es confirma que l'optimització paral·lela amb OpenMP que ha donat els millors resultats és el paral·lelitzat del bucle per inicialitzar les randomitzacions i calcular-ne el valor de nestedness**.

3.7.2.2 MPI

Com s'ha explicat en l'apartat de Materials i metodologies, cada procés que es crea amb MPI tindrà la seva pròpia secció de memòria privada, a la que la resta de processos no pot accedir. En conseqüència, a diferència d'OpenMP, cada procés tindrà la seva pròpia còpia de cada variable que es defineixi en el codi. Quan un procés modifiqui la variable, aquesta només es modificarà en la memòria d'aquest procés. A més, també cal recordar que MPI és més intrusiu en el codi que OpenMP. Per tant, per executar el programa de forma paral·lela

amb MPI, ha calgut fer-hi més modificacions que les que ha calgut per executar-lo de forma paral·lela amb OpenMP.

Primer de tot, es defineixen les variables `rank_process`, que emmagatzemarà el rang del procés, i `num_processes`, que emmagatzemarà el número de processos. La infraestructura d'MPI junt amb aquestes dues variables s'inicialitzen sempre amb les següents rutines:

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank_process);
MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
```

Posteriorment, ha estat necessari decidir quines seccions del codi es vol que s'executin de forma seqüencial i quines de forma paral·lela. Les seccions que es vol que s'executin de forma seqüencial seran executades per un únic procés. Tot i que aquesta funció la podria portar a terme qualsevol procés, és recomanable, i així està estandarditzat, que sigui el procés 0, ja que aquest és l'únic cas en el que sempre s'executarà aquesta part del codi, fins i tot quan hi hagi una sola unitat d'execució en el computador. Totes les definicions de variables, inicialitzacions i crides a procediments que es vol que siguin executades exclusivament pel procés 0, s'executaran dins de la següent estructura condicional:

```
if (rank_process == 0) {
	// Codi seqüencial.
}
```

Per altra banda, les seccions que es vol que s'executin de forma paral·lela, seran executades per tots els processos i, per a que això sigui possible, seran les que patiran més modificacions. D'entrada, només s'haurien de modificar per a ser executades per tots els processos, aquelles estructures que s'han descrit en el subapartat de disseny del paral·lelisme. És a dir, les cinc estructures anidades possibles de la funció `calculate_nested_value()` i el bucle per inicialitzar les randomitzacions i calcular-ne el valor de *nestedness*.

Ara bé, en tots aquells procediments que es voldria que només fossin executats pel procés 0, es pot calcular el contingut de variables que els altres processos necessitin per executar amb èxit la part paral·lela. Aquest és el cas de les variables `num_rows` i `num_cols`, que serveixen per emmagatzemar el nombre de files i el nombre de columnes, respectivament, de la matriu a la que se li està fent el test de *nestedness*. En MPI aquesta limitació es soluciona amb pas de missatges.

MPI proporciona diverses rutines que implementen diversos mecanismes de pas de missatges. La més senzilla és MPI_Bcast(), que serveix per enviar un missatge des d'un procés a la resta. Aquesta rutina es podria utilitzar per, un cop el procés 0 ha guardat el nombre de files i el nombre de columnes a les seves pròpies variables, enviar aquest valor a la resta de processos de la següent manera:

```
MPI_Bcast(&num_rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&num_cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

No obstant, abans de decidir utilitzar-les cal tenir dues coses en compte. Primer de tot, l'enviament i recepció d'un missatge comporta un *overhead* important. Segon, **l'últim pas que porta a terme aquesta rutina** (així com la resta de rutines de pas de missatges) **és una suspensió temporal de l'execució del procés** fins que es rebi el missatge esperat. Aquest pas és imprescindible perquè, si no hi fos, el procés podria executar instruccions en les que s'utilitzi una variable, el valor de la qual no ha estat actualitzat i, per tant, és incorrecte. Aquests dos factors poden tenir un impacte negatiu en el rendiment del programa.

En conseqüència, pel cas d'algunes variables, serà recomanable que cada procés se'n calculi el contingut encara que aquest sigui igual per a tots els processos. Per aquest motiu, és millor que l'acció `select_matrix(argv[3], &num_rows, &num_cols)`, encarregada d'inicialitzar les variables `num_rows` i `num_cols`, sigui executada per tots els processos independentment de que el nombre de files i de columnes de la matriu és el mateix per tots ells. D'aquesta manera, ens podem estalviar aquest pas de missatges.

El cas contrari el trobem en la variable `num_ones`, que emmagatzema el número de uns que tindrà la matriu binària a la que es fa el test de *nestedness*. El contingut d'aquesta variable és important per saber quants uns s'han de distribuir de manera aleatòria cada vegada que s'inicialitza una de les randomitzacions. Si es volgués evitar el pas de missatges, seria necessari que tots els processos llegissin el fitxer amb les dades d'entrada, generessin la mateixa matriu d'abundàncies relatives, fessin el discretitzat d'aquesta matriu i en calculessin el número d'uns. D'aquests passos, el més problemàtic seria la lectura del fitxer d'entrada, perquè cal accedir a la memòria secundària o disc. En general, s'ha d'evitar que més d'un procés accedeixi simultàniament a memòria secundària. L'únic cas en el que podria ser interessant contemplar aquesta possibilitat és quan cada unitat d'execució del computador en la que s'executi el procés tingui la seva pròpia memòria secundaria privada.

Una altra possibilitat seria que la matriu d'abundàncies relatives o la matriu binaria fossin generades pel procés 0 i enviades a la resta de processos. Aquesta opció requereix de l'enviament d'una gran quantitat de dades i, com ens podem imaginar, serà molt ineficient. En aquest cas, la millor alternativa és que el càlcul del contingut de la variable `num_ones` el porti a terme el procés 0, i aquest l'enviï a la resta de processos. D'aquesta manera, s'evita que més d'un procés accedeixi a la memòria secundària i la rutina de pas de missatges és més eficient perquè només envia una sola dada. El procés 0 comparteix el contingut de la variable `num_ones` amb la resta de la següent manera:

```
MPI_Bcast(&num_ones, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Abans de continuar, cal aclarir que tant sols serà necessari que tots els processos tinguin actualitzada la variable `num_ones` quan l'estructura que s'executi de forma paral·lela sigui el bucle per inicialitzar les randomitzacions i calcular-ne el valor de *nestedness*. En cas de que les rutines d'MPI s'utilitzin només sobre les estructures anidades de la funció `calculate_nested_value()`, l'únic procés que treballarà amb aquesta variable serà el procés 0. Tot i així, he pensat que era més interessant explicar tot això abans de descriure els

mecanismes pels quals s'ha provat d'executar de forma paral·lela el contingut de la funció `calculate_nested_value()`.

Com es pot intuir pel seu funcionament, el mecanisme per paral·lelitzar un bucle amb MPI és conceptualment diferent a com es fa en OpenMP. En MPI, els bucles que es vol que siguin executats per tots els processos es mantenen fora dels condicionals que delimiten el codi que ha de ser executat pel procés 0. Ara bé, si aquest fos l'únic pas que es fes, tots els processos executarien totes les iteracions dels bucles i això seria absurd. Per tant, s'ha de definir una nova variable que emmagatzemi el resultat de dividir el número d'iteracions del bucle entre el número de processos que es volen crear. Aquesta variable serà la que determini el nombre d'iteracions del bucle que haurà de fer cada procés i serà calculada independentment per cada procés ja que, igual que passa amb les variables `num_rows` i `num_cols`, que la calculés el procés 0 i l'enviés a la resta de processos seria més costós. El nom d'aquesta variable serà `num_rows_per_process` o `num_cols_per_process` en funció del bucle amb el que s'estigui treballant.

A més, s'ha de tenir en compte que el número d'iteracions d'un bucle no té perquè ser divisible pel número de processos que es vulguin crear. La solució més senzilla a aquesta inconvenient es calcular el residu de la divisió, que s'emmagatzemarà en una variable anomenada `remainder`, i sumar-li a la variable `num_rows_per_process`, `num_cols_per_process` o `num_randomized_per_process` del procés 0.

L'estructura anidada per calcular el nombre d'interaccions de les files i el nombre d'interaccions de les columnes de manera conjunta m'ha resultat impossible de paral·lelitzar. En concret, no he aconseguit que el vector `sum_cols` tingui els valors correctes després de la fase final de reducció. Tenint en compte el resultats obtinguts amb OpenMP (apartat d'avaluació) no hi havia cap expectativa de que el paral·lelitzat d'aquesta estructura anidada es traduís en una millora en el rendiment del programa, per tant s'ha desistit amb aquesta pas.

Les estructures anidades per calcular el nombre d'interaccions de les files i per calcular el nombre d'interaccions de les columnes, les quals resulten de desplegar l'estructura anterior, sí que s'han aconseguit paral·lelitzar i, en totes dues s'ha fet de la mateixa manera. Primer de tot, es fa una dispersió de files de la matriu que s'ha d'avaluar entre cada procés. És a dir, es reparteixen tantes files entre els diferents processos com indica la variable `num_rows_per_process` en cas de que es vulguin calcular les interaccions de les files o com indica la variable `num_cols_per_process` en cas de que es vulguin obtenir les interaccions de les columnes. En el primer cas, s'utilitzarà la matriu que la funció rep per paràmetre, mentre que en el segon cas s'utilitzarà la transposada d'aquesta matriu. Les rutines per fer aquest pas en cada cas són les següents:

```
MPI_Scatterv(&matrix[0][0], fragments, scroll, MPI_SHORT, &matrix[0][0], num_rows_per_process * num_cols, MPI_SHORT, 0, MPI_COMM_WORLD);
```

```
MPI_Scatterv(&transposed_matrix[0][0], fragments, scroll, MPI_SHORT, &transposed_matrix[0][0], num_cols_per_process * num_rows, MPI_SHORT, 0, MPI_COMM_WORLD);
```

Les variables `scroll` i `fragments` són vectors de mida `num_processes`. **L'ús d'aquests dos vectors és necessari quan es treballa amb rutines de dispersió i/o agrupació de**

vectors i/o matrius en les que els fragments en que es divideix o s'ha dividit l'estructura de dades tenen mides diferents. La primera variable emmagatzema les posicions de la matriu o vector en les que comença cada fragment mentre que la segona variable emmagatzema la mida d'aquests fragments.

Després del pas de dispersió, s'executa l'estructura anidada. S'ha de tenir en compte que, tot i que la mida del vector `sum_rows` o del vector `sum_cols`, en funció del cas, es manté intacta; cada procés omplirà només les `num_rows_per_process` o `num_cols_per_process` primeres posicions del vector.

Finalment, quan tots els processos hagin recopilat les interaccions que els hi pertoca, es fa un últim pas d'agrupació dels elements dels vectors. És a dir, es còpia el contingut del vector `sum_rows` o del vector `sum_cols` de cada procés en les posicions lliures del vector `sum_rows` o `sum_cols` del procés 0. Les rutines per fer aquest pas en cada cas són les següents:

```
MPI_Gatherv(sum_rows, num_rows_per_process, MPI_INT, sum_rows, fragments, scroll, MPI_INT, 0, MPI_COMM_WORLD);
```

```
MPI_Gatherv(sum_cols, num_cols_per_process, MPI_INT, sum_cols, fragments, scroll, MPI_INT, 0, MPI_COMM_WORLD);
```

El procediment per aplicar paral·lelisme a les dues estructures anidades encarregades de calcular les quatre components de l'equació per calcular el valor de *nestedness* és, de nou, el mateix per ambdues. En aquest cas, s'ha de tenir compte que l'estructura anidada per calcular les interaccions de les files i les interaccions de les columnes serà executada només pel procés 0. Per tant, el primer pas serà enviar el vector `sum_rows` o el vector `sum_cols` (en funció de l'estructura anidada que es vulgui paral·lelitzar), a la resta de processos. Les directives per portar a terme aquest pas són les següents:

```
MPI_Bcast(sum_rows, num_rows, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(sum_cols, num_cols, MPI_INT, 0, MPI_COMM_WORLD);
```

També s'haurà d'enviar a la resta de processos la matriu o la transposada de la matriu, cosa que s'aconsegueix amb les següents directives:

```
MPI_Bcast(&matrix[0][0], num_rows * num_cols, MPI_SHORT, 0, MPI_COMM_WORLD);
MPI_Bcast(&transposed_matrix[0][0], num_cols * num_rows, MPI_SHORT, 0, MPI_COMM_WORLD);
```

Igualment, s'ha de tenir en compte que, per executar aquestes estructures anidades de forma paral·lela, els processos no tenen perquè fer el mateix número d'iteracions i hauran d'accedir a posicions diferents de la matriu. Per aconseguir que això funcioni, es recuperen els vectors `scroll` i `fragments`, els quals tenen el mateix contingut que abans, però, en aquest cas, aquest contingut el genera només el procés 0 i s'utilitza amb una finalitat diferent. Cada posició de cada un d'aquests vectors s'envia a un procés diferent. En cada procés, l'element

de la variable `scroll` que s'ha rebut representarà la posició de la matriu a la que s'haurà d'accedir en la primera iteració del bucle, mentre que l'element de la variable `fragments` que s'ha rebut representarà el número d'iteracions que haurà de realitzar aquell procés. Les directives per dispersar el vector `scroll` són les següents:

```
MPI_Scatterv(scroll, 1, MPI_INT, &first_row, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatterv(scroll, 1, MPI_INT, &first_col, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Les directives per dispersar el vector `fragments` són les següents:

```
MPI_Scatterv(fragments, 1, MPI_INT, &num_rows_per_process, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatterv(fragments, 1, MPI_INT, &num_cols_per_process, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Un cop executada l'estructura anidada que a la que s'ham aplicat totes aquestes optimitzacions paral·leles, queda el pas de reducció dels càlculs intermedis de les quatre components de l'equació per calcular el valor de *nestedness*. Amb MPI, aquest pas es porta a terme amb les següents directives:

```
MPI_Reduce(&first_isocline_per_process, &first_isocline, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&third_isocline_per_process, &third_isocline, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Reduce(&second_isocline_per_process, &second_isocline, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&fourth_isocline_per_process, &fourth_isocline, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Abans de continuar, és necessari aclarir algunes coses. Primer, a part de les directives de reducció, el pas de missatges que s'ha descrit és extremadament ineficient. No obstant, la seva optimització de manera exitosa no és senzilla. Es veritat que les directives encarregades de la dispersió de la posició inicial de la matriu i de la quantitat d'iteracions es poden eliminar si cada procés fa aquests càlculs pel seu compte, de manera similar a com es fa amb les variables `num_rows_per_process` i `num_cols_per_process`. Ara bé, l'única manera d'evitar l'enviament dels vectors `sum_rows` i `sum_cols` és que cada procés executi també la estructura anidada encarregada de calcular les interaccions de les files i de les columnes.

Això ens porta a la conclusió de que no té gaire sentit aplicar optimitzacions paral·leles a les estructures anidades per calcular les components de l'equació per calcular el valor de *nestedness* sinó s'apliquen també a l'estructura anidada per calcular les interaccions de les files i les interaccions de les columnes (la qual només s'ha aconseguit paral·lelitzar després de desplegar-la en les dues estructures anidades equivalents més

senzilles). Però fins i tot afegint aquest canvi, encara seria necessària una directiva de pas de missatges per enviar o dispersar la matriu que la funció `calculate_nested_value()` rep per paràmetre. La única manera d'evitar aquest pas, seria que cada procés llegís el fitxer d'entrada de la memòria secundària, alternativa que, com ja s'ha explicat, només donaria bons resultats si cada procés tingués la seva pròpia memòria secundària privada.

La conclusió que podem extreure de tot això, és que **les optimitzacions paral·leles plantejades per a les estructures anidades de la funció `calculate_nested_value()` són complexes de programar i**, a més, **requereixen de l'enviament de grans volums de dades**. **Això significa que els resultats que es poden esperar seran pitjors que els d'OpenMP**.

Tot el contrari del que passa amb la última estructura iterativa, que és el bucle per inicialitzar les randomitzacions i calcular-ne el valor de *nestedness*. Tot i que aquest bucle es troba dins de la funció `generate_nested_values_randomized()`, les modificacions aplicades s'han focalitzat en la funció `nestedness_test()`, que és des de la que es crida a aquesta funció.

Primer de tot, en la funció `nestedness_test()`, es defineix una variable anomenada `num_randomized_per_process` en la que s'emmagatzema el resultat de dividir el número de randomitzacions entre el número de processos. A continuació, aquesta variable es passa per paràmetre en la crida a la funció `generate_nested_values_randomized()` en la posició que abans ocupava la variable `num_randomized`. D'aquesta manera, s'aconsegueix que el bucle de la funció faci només el número d'iteracions que li pertoca a cada procés. A més, com que els processos tenen la seva pròpia copia privada de la variable `randomized_matrix`, no es produiran els problemes d'accés a memòria que s'han hagut de solucionar en la implementació amb OpenMP. Quan finalitzi l'execució de la funció `generate_nested_values_randomized()`, cada procés tindrà un vector `nested_values` amb `num_randomized_per_process` valors de *nestedness*. Per ajuntar tots els valors de *nestedness* en el vector `nested_values` del procés 0 primer s'ha utilitzat una directiva d'agrupació, que és la següent:

```
MPI_Gatherv(nested_values, num_randomized_matrices_per_process, MPI_DOUBLE,
nested_values, fragments, scroll, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Si s'utilitza aquesta directiva, els passos de càlcul del valor de *nestedness* de la matriu binaria original, emmagatzemament d'aquest valor de *nestedness* en el vector `nested_values`, ordenament de tots els valors de *nestedness* del vector, obtenció de la posició que aquest valor de *nestedness* ocupa en el vector ordenat i càlcul del p-valor, són realitzats pel procés 0.

Ara bé, **el pas de missatges amb aquesta directiva d'agrupació és costós**, ja que cada procés ha d'enviar `num_randomized_matrices_per_process` valors de *nestedness* al procés 0. Una manera d'optimitzar aquest codi de manera que no s'hagin d'enviar tants elements és que cada procés calculi la posició que el valor de *nestedness* de la matriu binaria original ocuparà en el seu propi vector `nested_value` i després sumar totes aquestes posicions intermèdies. La posició que el valor de *nestedness* de la matriu binaria original ocupa en el vector `nested_values` del procés 0 obtingut amb la directiva d'agrupació explicada és igual a la suma de les posicions que aquest mateix valor de *nestedness* ocupa en els vectors `nested_values` de cada procés. En aquest cas, el procés 0 segueix sent l'encarregat de

calcular el valor de *nestedness* de la matriu binaria original, però després haurà d'enviar-lo a la resta de processos amb la següent directiva:

```
MPI_Bcast(&nested_elements.nested_value, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Per altra banda, com que ara l'objectiu no és agrupar els valors de *nestedness* per a que el procés 0 faci els passos restants; cada procés s'ha d'encarregar d'afegir el valor de *nestedness* de la matriu binaria original al seu vector `nested_value`, d'ordenar aquest vector i, finalment, d'obtenir la posició que el valor de *nestedness* de la matriu binaria original ocupa en aquest vector quan ja està ordenat. L'enviament i agrupació de cada posició intermèdia es farà amb la següent directiva de reducció:

```
MPI_Reduce(&partial_index, &global_index, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
```

Finalment, el procés 0, un cop hagi obtingut la posició global, és el que s'encarrega de calcular el p-valor.

Sembla que amb aquesta última optimització, la quantitat de missatges que s'ha de enviar ja no es pugui reduir més, però encara hi ha una última cosa que es pot fer. Aquesta consisteix en enviar les variables `num_ones` i `nested_value` conjuntament amb una única directiva de pas de missatges. Això s'aconsegueix amb les següents instruccions i la següent directiva:

```
message[0] = (double) count_ones_binary_matrix(matrix, num_rows, num_cols);
message[1] = calculate_nested_value_optimized(matrix, num_rows, num_cols);
MPI_Bcast(message, 2, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Amb aquest últim canvi, **la comunicació entre els processos en l'optimització paral·lela d'aquest bucle implementada amb MPI queda reduïda a únicament dues directives de pas de missatges amb les que**, en total, **s'envien tres elements**.

Abans d'acabar aquest subapartat, només queda afegir que, per finalitzar tota la infraestructura creada amb MPI, és necessari executar la rutina `MPI_Finalize()`.

3.7.2.3 MPI i OpenMP

En el dos subapartats anteriors s'ha descrit tot un ventall d'optimitzacions paral·leles fetes en diferents seccions del codi. La majoria d'aquestes optimitzacions paral·leles es poden implementar tant utilitzant OpenMP com utilitzant MPI. Fins ara, aquestes eines s'han fet servir per separat però, de la mateixa manera que una d'aquestes eines pot combinar vàries de les optimitzacions descrites, es pot **combinar la utilització de les dues**, tenint en

compte que cada una s'ha d'ocupar d'una optimització diferent. Degut a que aquest plantejament s'ha fet després d'obtenir els resultats (explicats en l'apartat següent), s'ha decidit combinar les dues optimitzacions paral·leles que han proporcionat el menor temps d'execució.

Així, OpenMP s'ha utilitzat per paral·lelitzar la estructura anidada de la funció `calculate_nested_value()` que s'encarrega de calcular les dues components de l'equació per calcular el valor de *nestedness* relatives a les columnes. En concret, s'utilitza la directiva:

```
#pragma omp parallel for private(first_col, second_col, row) shared(total_cols, num_cols, num_rows, transposed_matrix, sum_cols) reduction(+:second_isocline, fourth_isocline) default(none) schedule(dynamic)
```

Per altra banda, MPI s'ha utilitzat per paral·lelitzar el bucle que s'encarrega d'inicialitzar les randomitzacions i calcular-ne el valor de *nestedness*. Per tant, caldrà reutilitzar totes les directives d'MPI estàndard i les que s'han descrit al final del subapartat anterior.

4 Resultats i discussió

L'avaluació del rendiment del codi del projecte després d'aplicar cada una de les optimitzacions plantejades s'ha organitzat de manera similar a l'apartat anterior. Així, totes les optimitzacions seqüencials o optimitzacions paral·leles que s'han fet en cada llenguatge de programació i/o estàndard per aplicar paral·lelisme estaran agrupades en taules. A més, també s'han utilitzat les màquines del zoo Orca i Pop, així com el clúster d'ordinadors de la Sala dels Mac per avaluar l'evolució del rendiment a mesura que s'augmenta el número d'unitats d'execució (*threads* o processos). Per cada optimització avaluada en l'ordinador portàtil, s'han recopilat els temps d'execució del programa tant quan només es calcula el valor de *nestedness* de la matriu binaria (una sola crida a la funció `calculate_nested_value()`) com quan s'executa el test de *nestedness* descrit (mil una crides a la funció `calculate_nested_value()`). En les altres màquines només s'ha avaluat el test de *nestedness*. Cada prova s'ha fet sempre per la matriu de vertebrats i per la matriu d'individus.

Cada temps d'execució presentat és la mitjana dels temps d'execució obtinguts després d'executar el programa deu vegades. Si l'optimització proposada millora el rendiment del programa respecte a la versió anterior o respecte a la versió seqüencial definitiva, el temps d'execució es marca de color verd. Per altra banda, si el temps d'execució empitjora, aquest es marca de color groc en cas de que l'augment no sigui prou significatiu com per descartar l'optimització i es marca de color vermell si l'augment és prou significatiu com per descartar l'optimització. En cas de que una optimització millori el rendiment en una de les dues matriu però l'empitjori en l'altre, es donarà més prioritat a la matriu d'individus per tractar-se de la matriu més gran.

4.1 Optimitzacions seqüencials

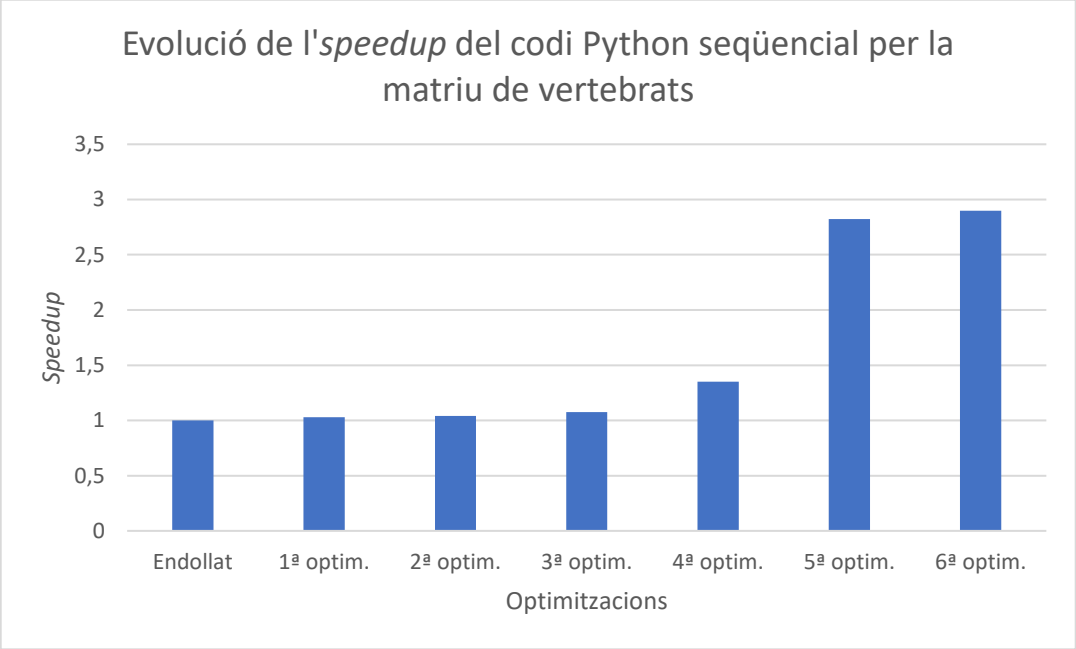
4.1.1 Python

4.1.1.1 Ordinador portàtil

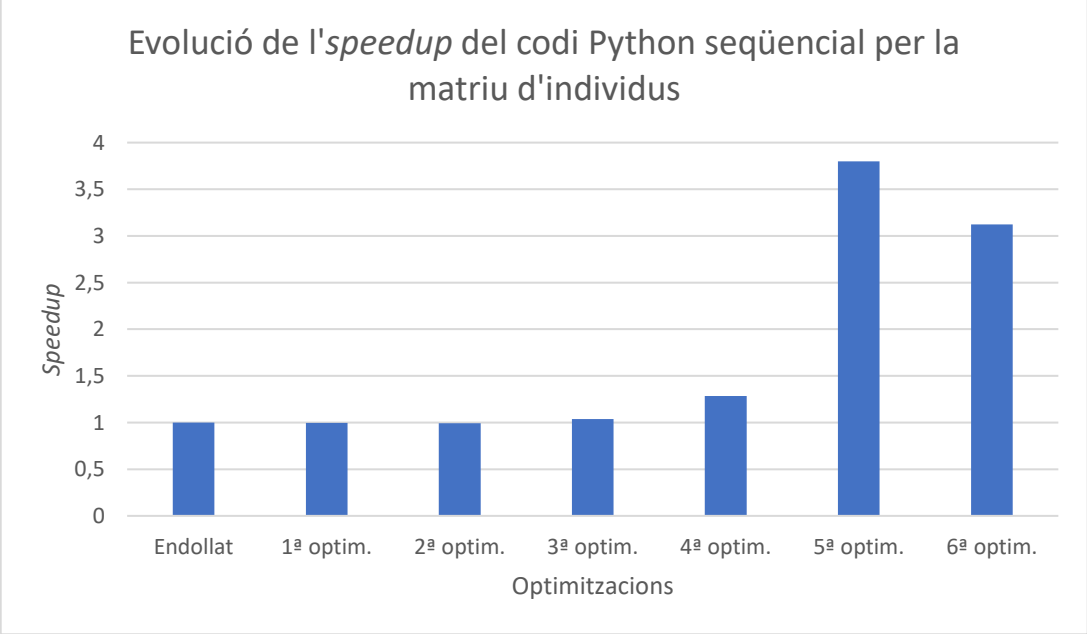
Com ja s'ha avançat en la introducció, la implementació del programa en Python triga tant de temps en executar-se en el meu ordinador portàtil que no ha estat possible recopilar els temps d'execució del test de *nestedness* per cada una de les optimitzacions proposades. En conseqüència, la taula i les gràfiques proporcionades només inclouen els temps requerits per obtenir el valor de *nestedness* de les dues matrius. És a dir, els temps d'execució requerits per executar una sola crida a la funció `calculate_nested_value()`.

Propostes	**Temps d'execució (s)**	
	Càlcul del valor de *nestedness*	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
Codi original.	98'983	31 min 19,508
Portàtil endollat a la llum.	56,181	16 min 45,483
Eliminat última iteració.	54,519	16 min 50,052

Canvi inicialització variable. Esborrat del condicional.	53,915	16 min 53,058
Agrupació parells d'estructures anidades.	52,250	16 min 10,372
Accés ràpid a les matrius.	41'629	13 min 1,971
Precalculat de les interaccions.	19,902	4 min 24,640
Ús de la funció zip() amb l'operador 'and'.	19,383	5 min 21,837



Gràfic 2. Gràfic de barres en el que es representa l'evolució de l'*speedup* del programa en Python amb l'aplicació de les optimitzacions de codi seqüencials per calcular el valor de *nestedness* de la matriu de vertebrats.



Gràfic 3. Gràfic de barres en el que es representa l'evolució de l'*speedup* del programa en Python amb l'aplicació de les optimitzacions de codi seqüencials per calcular el valor de *nestedness* de la matriu d'individus.

Una de les coses que crida l'atenció dels resultats mostrats en la taula és la millora que s'aconsegueix com a conseqüència d'endollar el portàtil a la llum i és per aquesta raó que se n'ha fet una menció en la fase de disseny. Aquest temps d'execució s'agafa com a temps base per calcular els *speedups* que es mostren en les gràfiques. A més, amb aquests resultats es pot especular que **l'execució del test de *nestedness* de la matriu d'individus en aquest portàtil, si aquest està endollat a la llum, requeriria entre 279 i 280 hores**.

Respecte les optimitzacions aplicades, es descarta l'ús de la funció zip() combinada amb l'operador 'and'. Tot i que hi ha una lleugera millora en el temps d'execució de la matriu de vertebrats, aquesta no sembla que es pugui considerar significativa i, a més, per la matriu d'individus el temps d'execució empitjora de forma considerable. Per altra banda, tot i que l'eliminat de la última iteració i el canvi en la inicialització de la variable combinada amb l'esborrat del condicional també han empitjorat el temps d'execució de la matriu d'individus, s'ha decidit mantenir aquestes optimitzacions. Això es degut a que els resultats són molt semblants i podria ser conseqüència de la variació en els temps d'execució del programa. Igualment, aquestes no són les optimitzacions amb les que s'esperava aconseguir la major millora en el rendiment del programa.

Amb l'optimització d'agrupar les quatres estructures anidades per calcular cada una de les quatre components de l'equació en dues estructures anidades segons si els càlculs es fan sobre les files o sobre les columnes s'aconsegueix una petita millora en el temps d'execució, i amb l'accés a matrius amb el mecanisme propi de numpy una millora ja més significativa. No obstant, **és amb el precalculat de les interaccions de les files i de les interaccions de les columnes que s'aconsegueix optimitzar el programa de manera absoluta**, com s'observa clarament en els gràfics.

Un cop s'han obtingut el temps d'execució **després d'aplicar totes les optimitzacions**, es pot especular novament el que requerirà fer **el test de *nestedness* de la matriu d'individus**. Aquesta vegada, **es preveu que serà d'entre 75** i **76 hores**.

Per últim, es proporciona l'*speedup* del codi Python després d'haver aplicat totes les optimitzacions respecte el codi Python en la seva versió original per la matriu d'individus. Totes dues execucions fetes amb el portàtil endollat a la llum.

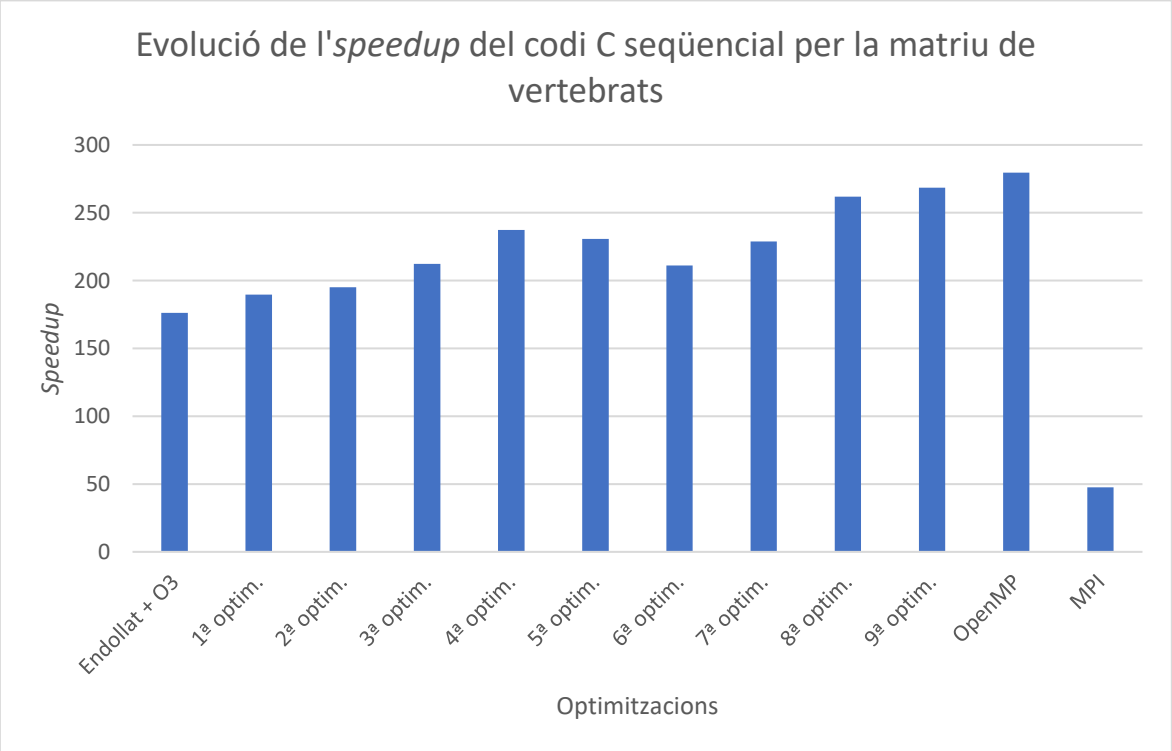
$$Speedup\ Python\ optimitzat = \frac{1005{,}483}{264{,}640} = 3{,}799$$

4.1.2 C

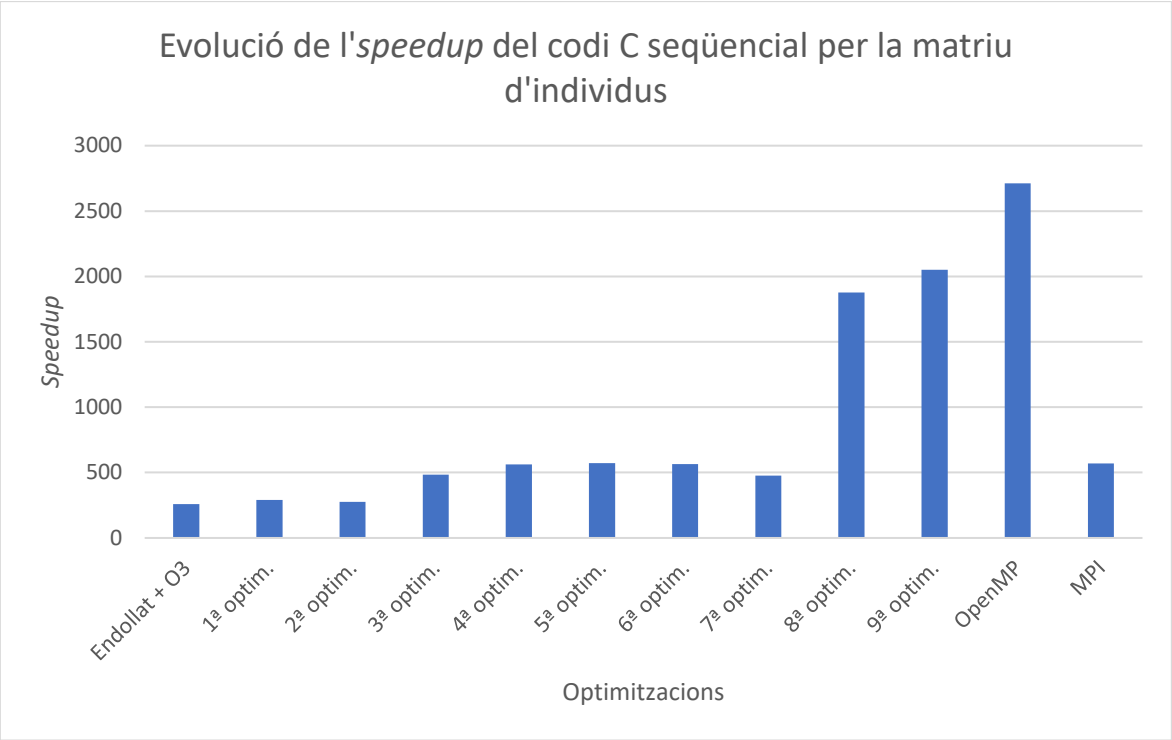
4.1.2.1 Ordinador portàtil

Abans de mostrar els resultats, cal explicar que, tot i que en la taula es proporcionin els temps d'execució del test de *nestedness*, com que el temps d'execució utilitzat com a temps base per obtenir els *speedups* és el temps d'execució de la versió més optimitzada del programa en Python, les gràfiques mostren l'evolució dels *speedups* pel càlcul del valor de *nestedness*. D'aquesta manera, es pot fer una comparativa entre les gràfiques en Python i les gràfiques en C. Si es volgués observar l'evolució dels *speedups* pel test de *nestedness*, caldria utilitzar com a temps base el temps d'execució especulat per realitzar el test de *nestedness* en la versió més òptima del codi Python.

Propostes	Temps d'execució (s)			
	Matriu de vertebrats (50 x 1056)		Matriu d'individus (644 x 1056)	
	Càlcul del valor de *nestedness*	Test de *nestedness* (1000 random.)	Càlcul del valor de *nestedness*	Test de *nestedness* (1000 random.)
Codi original.	0,687	6 min 54,470	10,049	2 h 28 min 30,119
Portàtil endollat a la llum.	0,203	2 min 6,908	2,950	52 min 22,930
Tercer nivell d'optimització.	0,113	40,584	1,018	17 min 17,838
Eliminat última iteració.	0,105	32,056	0,910	15 min 26,992
Canvi inicialització variable. Esborrat del condicional.	0,102	31,050	0,958	16 min 18,007
Agrupació parells d'estructures anidades.	0,0937	22,096	0,546	8 min 50,009
Precalculat interaccions.	0,0839	13,866	0,470	6 min 48,926
Matriu binària shorts.	0,0863	15,778	0,462	6 min 36,931
Matriu binària caràcters.	0,0943	21,864	0,469	6 min 39,661
Matriu binària booleans.	0,0870	15,870	0,555	7 min 58,410
Ús de la matriu transposada.	0,0760	4,824	0,141	1 min 20,688
Agrupat precalculat de les interaccions.	0,0741	4,619	0,129	1 min 7,158



Gràfic 4. Gràfic de barres en el que es representa l'evolució de l'*speedup* del programa en C amb l'aplicació de les optimitzacions de codi seqüencials per calcular el valor de *nestedness* de la matriu de vertebrats.



Gràfic 5. Gràfic de barres en el que es representa l'evolució de l'*speedup* del programa en C amb l'aplicació de les optimitzacions de codi seqüencials per calcular el valor de *nestedness* de la matriu d'individus.

En el cas d'aquests resultats, sobta la millora en el temps d'execució que s'aconsegueix quan l'algorisme s'implementa en C, sobre tot si es compila amb el tercer nivell d'optimització. L'*speedup* del codi en C amb tercer nivell d'optimització respecte el codi en Python encarregats de la realització del test de *nestedness* de la matriu d'individus es mostra a continuació:

$$Speedup\ C\ O3\ reespecte\ Python = \frac{1005{,}843}{1{,}018} = 988{,}058$$

La millora és impressionant i, tot i que encara queda lluny del màxim teòric, només amb la traducció del codi del programa a aquest llenguatge de programació ja es pot executar el test de *nestedness* en un temps raonable. Això ha permès descobrir que **tant la matriu de vertebrats com la matriu d'individus són matrius *nested***.

Respecte les optimitzacions que ja s'havien plantejat, l'evolució de l'*speedup* per la matriu de vertebrats és semblant al que s'observa en el codi en Python. Per altra banda, en la matriu d'individus, l'eliminat de la última iteració del bucle ara sí que dona una millora significativa en el temps d'execució. Per altra banda, el canvi en la inicialització de la variable combinada amb l'esborrat del condicional l'empitjora de manera també substancial. Per tant, aquesta última iteració es descarta. L'agrupat de les estructures anidades segons si el càlcul es fa sobre les files o sobre les columnes i el precalculat de les interaccions de les files i de les columnes també segueixen donant reduccions en el temps d'execució, sent ara la primera d'aquestes dues optimitzacions la que, per la matriu d'individus, té un major impacte en el rendiment del programa.

Respecte les noves propostes d'optimització seqüencial, quan la matriu binaria que la funció `calculate_nested_value()` rep per paràmetre és la matriu de vertebrats, qualsevol dels tipus de dades amb els que aquesta s'implementi donen sempre pitjors temps d'execució. Per altra banda, la matriu d'individus s'aconsegueix tractar més ràpid si els tipus de dades amb els que està implementada són shorts o caràcters. No així si el tipus de dades utilitzat és el booleà, que dona pitjors resultats per totes dues matrius. Al final, s'ha establert que els tipus de dades que contindrà la matriu binaria seran shorts, perquè és la que dona el menor temps d'execució per la matriu d'individus.

Finalment, **la utilització de la matriu transposada és l'optimització amb la que s'ha aconseguit la millora més gran en el rendiment del programa per la matriu d'individus** d'entre totes les plantejades i l'agrupat del precalculat de les interaccions de les files i de les interaccions de les columnes en una sola estructura anidada també ha reduït el temps d'execució.

A continuació, es mostra l'*speedup* del codi en C després d'haver aplicat totes les optimitzacions respecte el codi en C original, així com l'*speedup* del codi en C optimitzat respecte el del codi en Python optimitzat.

$$Speedup\ C\ optimitzat = \frac{1037{,}838}{67{,}158} = 15{,}455$$

$$Speedup\ C\ optimitzat\ respecte\ Python\ optimitzat = \frac{264{,}640}{0{,}129} = 2051{,}473$$

4.1.2.2 Orca

Propostes	**Temps d'execució (s)**	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 procés.	3,662	50,578

De entre totes les màquines en les que s'ha executat la versió seqüencial del test de *nestedness*, la màquina Orca és la que proporciona el temps més baix. Això es degut a que Orca és la màquina que utilitza el processador amb millors prestacions.

4.1.2.3 Pop

Propostes	**Temps d'execució (s)**	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 Node. 1 procés.	5,725	76,003

Per altra banda, la màquina Pop és la que ha proporcionat els pitjors temps d'execució. Fins i tot el programa executat en l'ordinador portàtil ha tingut un millor rendiment. Això és degut a que els processadors de Pop són molt més antics que els de qualsevol altra màquina, inclòs el de l'ordinador portàtil.

4.1.2.4 Sala dels Mac

Propostes	**Temps d'execució (s)**	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 Node. 1 procés.	4,528	66,512

Per executar la versió seqüencial del test de *nestedness* en un dels iMac del laboratori 205, ens hem assegurat de que aquest sigui un dels que disposa d'un processador i9. El temps d'execució és millor que en l'ordinador portàtil, tot i que queda bastant lluny del rendiment aconseguit en Orca. Això pot ser degut a que les primeres versions dels processadors i9 no proporcionen grans millores en el rendiment respecte als processadors i7.

4.2 Optimitzacions paral·leles

Totes les optimitzacions paral·leles de les que s'han recopilat resultats s'han obtingut amb les eines OpenMP i/o MPI. Per tant, totes les versions del codi paral·lelitzat han estat implementades en C.

4.2.1 OpenMP

4.2.1.1 Ordinador portàtil

OpenMP és la eina amb la que s'han desenvolupat més variants de les mateixes propostes de paral·lelisme. Ara bé, moltes d'aquestes no han repercutit en una millora en el rendiment del programa.

Propostes	**Temps d'execució (s)**			
	Matriu de vertebrats (50 x 1056)		**Matriu d'individus (644 x 1056)**	
	Càlcul del valor de *nestedness*	**Test de *nestedness* (1000 random.)**	**Càlcul del valor de *nestedness***	**Test de *nestedness* (1000 random.)**
Organització per seccions.	0,0771	6,025	0,110	47,707
Estructura anidada precalculat interaccions	0,0764	5,430	0,142	1 min 17,896

files i columnes. `Schedule(static).`				
Estructura anidada precalculat interaccions files. `Schedule(static).`	0,0759	4,943	0,131	1 min 10,822
Estructura anidada precalculat interaccions columnes. `Schedule(static).`	0,0775	6,533	0,144	1 min 17,965
Estructura anidada càlcul components relatives a les files. `Schedule(static).`	0,0775	5,028	0,121	55,057
Estructura anidada càlcul components relatives a les files. `Schedule(dynamic).`	0,0796	7,060	0,137	1 min 9,475
Estructura anidada càlcul components relatives a les files. `Schedule(static, 1).`	0,0775	7,024	0,136	1 min 12,115
Estructura anidada càlcul components files. `Schedule(static).` Segona optimització codi seqüencial.	0,0766	7,032	0,143	1 min 16,539
Estructura anidada càlcul components files. `Schedule(dynamic).` Segona optimització codi seqüencial.	0,0788	7,016	0,136	1 min 9,137
Estructura anidada càlcul components files. `Schedule(static, 1).` Segona optimització codi seqüencial.	0,0746	4,844	0,115	52,232
Estructura anidada càlcul components columnes. `Schedule(static).`	0,0717	2,208	0,110	50,128
Estructura anidada càlcul components columnes. `Schedule(dynamic).`	0,0712	1,408	0,0976	36,349
Estructura anidada càlcul components columnes. `Schedule(static, 1).`	0,0720	1,600	0,100	37,975
Estructura anidada càlcul components columnes. `Schedule(static).` Segona optimització codi seqüencial.	0,0718	2,361	0,107	45,627
Estructura anidada càlcul components columnes. `Schedule(dynamic).` Segona optimització codi seqüencial.	0,0762	1,424	0,101	36,643
Estructura anidada càlcul components relatives a les	0,0723	1,765	0,107	46,621

columnes. `Schedule(static, 1).` Segona optimització codi seqüencial.				
Bucle randomitzacions i valors de *nestedness*. `Schedule(static).`	-	1,161	-	16,583
Bucle randomitzacions i valors de *nestedness*. `Schedule(dynamic).`	-	1,138	-	16,288
Bucle randomitzacions i valors de *nestedness*. `Schedule(static, 1).`	-	1,168	-	16,349

El paral·lelitzat per seccions, tot i haver empitjorat el temps d'execució del programa quan s'utilitza la matriu de vertebrats, el millora substancialment quan s'utilitza la matriu d'individus. Això es degut a que la matriu d'individus té una forma molt més quadrada que la matriu de vertebrats i, per tant, els percentatges del temps d'execució del programa que es dedicaran als càlcul relacionats amb les files i als càlculs relacionats amb les columnes estan molt més equilibrats. Tot i així, el paral·lelitzat per seccions es descarta perquè no es pot saber si la matriu que rebrà per paràmetre la funció `calculate_nested_value()` tindrà forma quadrada o no. A més, aquest mecanisme només aprofita dos de les unitats d'execució disponibles, mentre que el paral·lelitzat de les estructures iteratives pot aprofitar-ne tantes com en proporcioni el computador.

Per altra banda, el paral·lelitzat de la estructura anidada per precalcular les interaccions de les files i les interaccions de les columnes ha empitjorat el temps d'execució del programa per les dues matrius. Si es desplega aquesta estructura anidada en les dues estructures anidades equivalents, el paral·lelitzat de qualsevol de les dues també empitjora el temps d'execució del programa per qualsevol de les matrius. Per tant, el paral·lelitzat de aquesta estructura o estructures queda completament descartat.

Totes les variants del paral·lelitzat de l'estructura anidada per calcular les components de l'equació relatives a les files han empitjorat el temps d'execució de la matriu de vertebrats. Això és degut a que aquesta matriu té moltes menys files que columnes i, per tant, la major part del temps d'execució del programa es concentra en el bucle següent. Respecte la matriu d'individus, únicament dues variants han aconseguit millorar el temps d'execució. D'una banda, si l'estructura anidada té forma quadrada (no s'aplica la segona optimització seqüencial) el repartiment estàtic de les iteracions en grups de número total d'iteracions entre número de *threads* millora el temps d'execució del programa. De l'altra banda, si l'estructura té forma triangular (s'aplica la segona optimització seqüencial) el repartiment estàtic de les iteracions en grups de una iteració en millora el temps d'execució. Aquesta optimització sí que podria tenir sentit mantenir-la, tot i que la millora es poc significativa en comparació amb les dos últimes.

Després, el paral·lelitzat de l'estructura anidada per calcular les components de l'equació relatives a les columnes ha millorat el temps d'execució per les dues matrius en totes les seves variants. Això es degut a que el número de columnes és superior al número de columnes. El millors resultats s'han obtingut quan l'estructura anidada té forma quadrada (no s'aplica la segona optimització seqüencial) i el repartiment de les iteracions es realitza de forma dinàmica. Aquests resultats no són els esperats, ja que el repartiment dinàmic de les iteracions acostuma a donar bons resultats quan les estructures iteratives tenen formes

completament irregulars. Aquesta optimització sí que té sentit mantenir-la, tot i que l'última optimització paral·lela que s'ha implementat dona resultats encara millors. Els resultats descrits en aquests dos últims paràgrafs permeten intuir la mida de la matriu a partir de la qual el paral·lelitzat d'aquests dos bucles comença a donar resultats positius, cosa que pot ser interessant per a futures optimitzacions paral·leles.

Per últim, el paral·lelitzat del bucle per inicialitzar les matrius d'individus i per calcular-ne el valor de *nestedness* és, amb diferència, el que ha millorat més el temps d'execució per les dues matrius. Aquests resultats ja s'havien previst en la fase de disseny i implementació. Les principals raons són les poques dependències existents entre les diferents iteracions del bucle i que les estructures per aplicar paral·lelisme es creen una única vegada. En aquest cas, totes les variants plantejades tenen un temps d'execució similar, sent el repartiment dinàmic de les iteracions lleugerament millor. De nou, aquest resultat no és l'esperat, ja que el bucle té forma quadrada.

L'*speedup* del paral·lelitzat d'aquest bucle amb OpenMP respecte el codi seqüencial es proporciona a continuació:

$$Speedup\ OpenMP\ respecte\ seqüencial\ (portàtil) = \frac{67{,}158}{16{,}288} = 4{,}123$$

4.2.1.2 Orca

La versió del codi OpenMP que s'ha executat en la màquina Orca és la que dona el menor temps d'execució. Es a dir, el paral·lelitzat el bucle per inicialitzar les randomitzacions i calcular-ne el valor de *nestedness* amb repartiment dinàmic de les iteracions.

Propostes	**Temps d'execució (s)**	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 procés. 4 *threads*.	1,014	12,424
1 procés. 8 *threads*.	0,700	6,663
1 procés. 12 *threads*.	0,844	5,425
1 procés. 16 *threads*.	1,100	6,162
1 procés. 32 *threads*.	1,204	7,786
1 procés. 64 *threads*.	1,236	8,526
1 procés. 128 *threads*.	1,512	12,022
1 procés. 256 *threads.* (*Oversubscribe*).	2,205	14,638

Propostes	***Speedup***	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 procés. 4 *threads*.	3,611	4,071

1 procés. 8 *threads*.	5,231	7,591
1 procés. 12 *threads*.	4,339	9,323
1 procés. 16 *threads*.	3,329	8,208
1 procés. 32 *threads*.	3,042	6,496
1 procés. 64 *threads*.	2,963	5,932
1 procés. 128 *threads*.	2,422	4,207
1 procés. 256 *threads*. (*Oversubscribe*).	1,661	3,455

Els resultats obtinguts mostren que el menor temps d'execució del test de *nestedness* de la matriu de vertebrats paral·lelitzat amb OpenMP s'obté quan es creen 8 *threads*, mentre que en l'execució del test de *nestedness* de la matriu d'individus, s'obté quan es creen 12 *threads*. Si el número de *threads* es segueix augmentant, el rendiment del programa empitjora de manera gradual. Això es degut a que els *threads* creats amb OpenMP comparteixen la memòria principal i, per tant, quan un d'ells modifica una dada, aquesta dada s'invalida en la memòria principal i, per tant, també en la resta de nivells de la jerarquia de memòria. Aquest fenomen s'anomena ***false sharing***, i és més habitual a mesura que s'augmenta el número de *threads*.

A més, es força la màquina a entrar en un estat d'*oversubscribe*, que consisteix en crear més *threads* que les unitats d'execució *hardware* de les que disposa per veure el seu comportament. Quan s'entra en aquest estat, l'empitjorament del rendiment del programa augmenta lleugerament respecte al que s'observa en les execucions anteriors.

L'*speedup* del test de *nestedness* de la matriu d'individus paral·lelitzat amb 12 *threads* d'OpenMP respecte el codi seqüencial en la màquina Orca es mostra a continuació:

$$Speedup\ OpenMP\ respecte\ seqüencial\ (orca) = \frac{50{,}578}{5{,}425} = 9{,}323$$

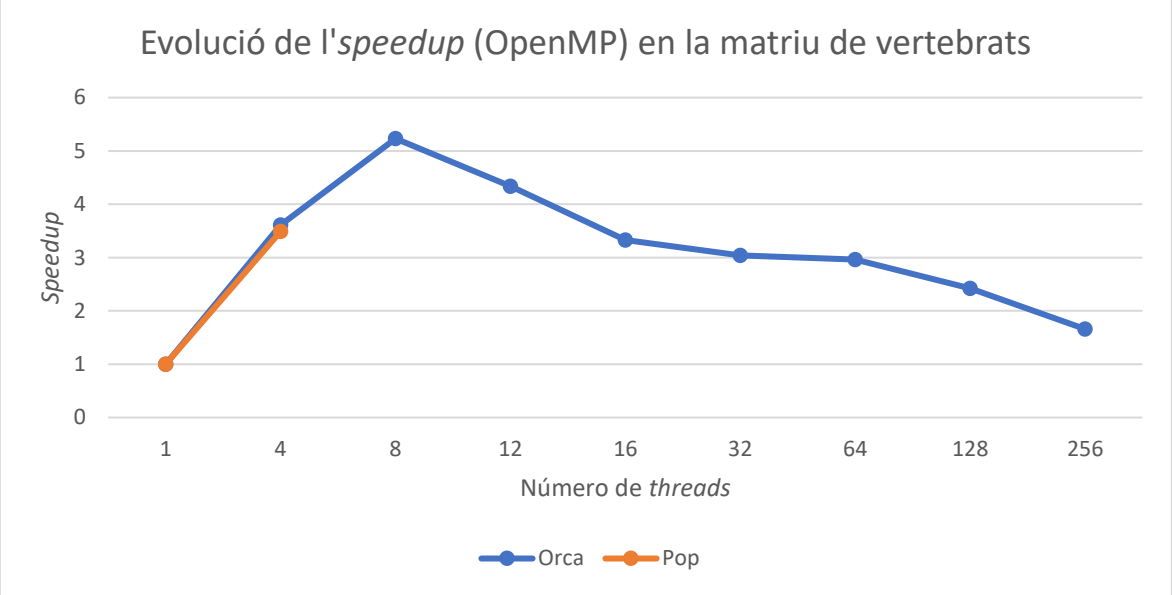
4.2.1.3 Pop

La versió del codi OpenMP que s'ha executat en la màquina Pop és, de nou, el paral·lelitzat el bucle per inicialitzar les randomitzacions i calcular-ne el valor de *nestedness* amb repartiment dinàmic de les iteracions.

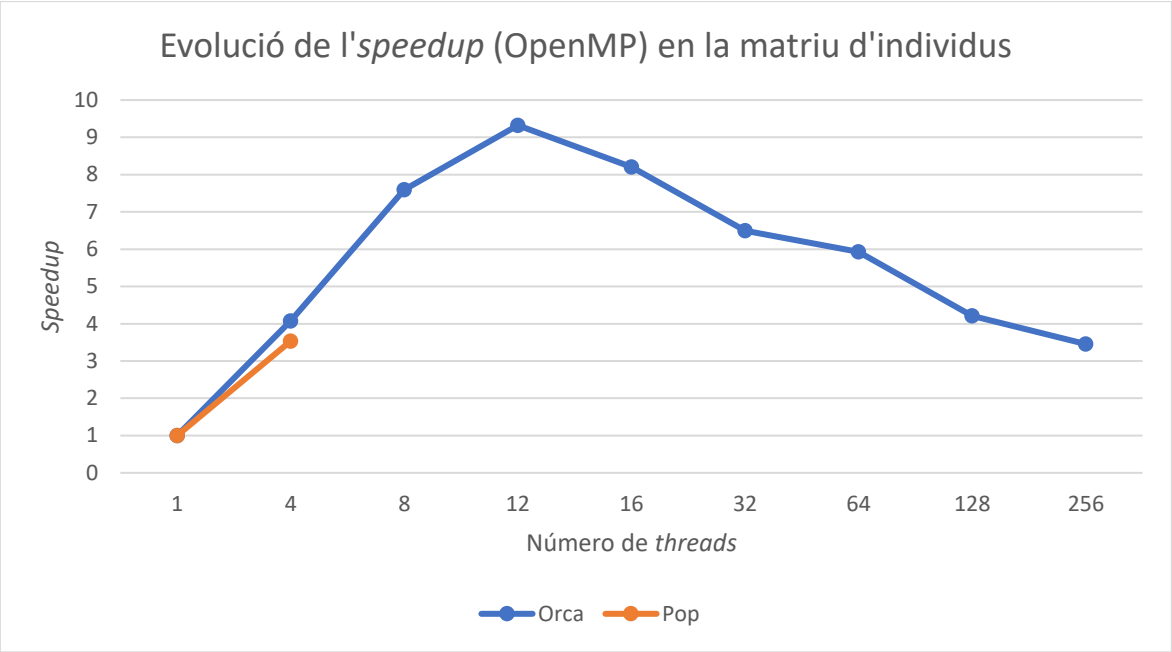
Propostes	**Temps d'execució (s)**	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 Node. 1 procés. 4 *threads*.	1,641	21,493

Propostes	***Speedup***	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 Node. 1 procés. 4 *threads*.	3,489	3,536

Els resultats obtinguts mostren una reducció del temps d'execució del programa respecte a la versió seqüencial, tot i que aquesta reducció no ha estat tan gran com la que s'aconsegueix en la màquina orca amb el mateix número de *threads*. A més, orca pot arribar a crear 12 *threads* abans de que el temps d'execució comenci a empitjorar. Per altra banda, com s'ha explicat a l'apartat de materials i metodologies, cada node de Pop té només dos processadors, cada un dels quals te només dos *cores*. Això significa que, com a màxim, un node de Pop podrà crear 4 *threads*. Finalment, tot i que Pop disposa de vuit nodes, **no és possible repartir els *threads* d'OpenMP entre els diferents nodes** perquè aquests es creen i es destrueixen en temps d'execució. Per a poder aprofitar la màquina Pop en la seva totalitat, caldrà utilitzar una eina d'optimització paral·lela que creï processos en lloc de *threads*, com és el cas d'MPI, de manera que les unitats d'execució es puguin repartir entre els diferents nodes.



Gràfic 6. Evolució de l'*speedup* en la matriu de vertebrats quan es paral·lelitza amb OpenMP.



Gràfic 7. Evolució de l'*speedup* en la matriu d'individus quan es paral·lelitza amb OpenMP.

L'*speedup* del test de *nestedness* de la matriu d'individus paral·lelitzat amb 4 *threads* d'OpenMP respecte el codi seqüencial en la màquina Pop es mostra a continuació:

$$Speedup\ codi\ c\ amb\ OpenMP\ (pop)\ respecte\ seqüencial = \frac{76{,}003}{21{,}493} = 3{,}536$$

4.2.2 MPI

4.2.2.1 Ordinador portàtil

La major part de les optimitzacions paral·leles implementades amb MPI són les mateixes que amb OpenMP, tot i que no existeix l'opció de desenvolupar-ne tantes variants com permet OpenMP. Ara bé, la major part de les propostes han donat encara pitjors resultats en el rendiment del programa que quan s'ha utilitzat OpenMP.

Propostes	**Temps d'execució (s)**			
	Matriu de vertebrats (50 x 1056)		**Matriu d'individus (644 x 1056)**	
	Càlcul del valor de *nestedness*	**Test de *nestedness* (1000 random.)**	**Càlcul del valor de *nestedness***	**Test de *nestedness* (1000 random.)**
Estructura anidada precalculat interaccions files.	0,428	5,404	0,490	1 min 12,171

Estructura anidada precalculat interaccions columnes	0,424	5,544	0,499	1 min 19,067
Estructures anidades precalculat interaccions files i interaccions columnes.	0,427	7,745	0,504	1 min 42,320
Estructura anidada càlcul components files.	0,435	6,949	0,500	1 min 16,904
Estructura anidada càlcul components columnes.	0,419	2,642	0,479	47,675
Esborrat pas de missatges `num_rows` i `num_cols`.	0,432	2,678	0,464	46,351
Bucle randomitzacions i valors de *nestedness*. Agrupació valors de *nestedness* (`Gatherv()`).	-	1,660	-	19,909
Bucle randomitzacions i valors de *nestedness*. Reducció posicions valor de *nestedness* matriu original (`Reduce()`).	-	1,658	-	19,862
Bucle randomitzacions i valors de *nestedness*. Un sol missatge (`Broadcast()`) amb `num_ones` i nested_value.	-	1,419	-	15,500

Primer de tot, com s'ha explicat en l'apartat de disseny i implementació, no s'ha aconseguit paral·lelitzar l'estructura anidada per precalcular les interaccions de les files i les interaccions de les columnes. Per tant, s'ha desplegat en les dues estructures anidades equivalents. Si es paral·lelitza qualsevol de aquestes dues estructures, el temps d'execució del programa empitjora per qualsevol de les dues matrius. Les raons per les que té lloc aquest fet són les mateixes que s'han explicat en la secció d'OpenMP, amb l'afegit de que l'*overhead* de la gestió dels processos és més gran perquè les estructures són més pesades i la comunicació es fa a través de pas de missatges. Així, el paral·lelitzat amb MPI d'aquesta estructura o estructures queda, de nou, completament descartat.

Igualment el paral·lelitzat de l'estructura anidada per calcular les components de l'equació relatives a les files ha empitjorat el temps d'execució per qualsevol de les proves fetes amb les dues matrius. Això és degut a que, ni tan sols en el cas de la matriu d'individus, el número d'iteracions que s'hauran de fer per tractar les files és prou alt com per compensar l'*overhead* de la comunicació entre processos. S'ha de tenir en compte que qualsevol de les directives de pas de missatges que es facin dins de la funció `calculate_nested_value()` s'executarà tantes vegades com crides es facin a la funció. Per tant, el paral·lelitzat d'aquesta estructura amb MPI també queda completament descartat.

Després, el paral·lelitzat de l'estructura anidada per calcular les components de l'equació relatives a les columnes ha donat uns resultats curiosos. En concret, el temps d'execució del càlcul del valor de *nestedness* de qualsevol de les dues matrius és pitjor que quan el programa s'executa de forma seqüencial. Per altra banda, el temps d'execució del test de *nestedness* és millor per totes dues matrius. En conseqüència, es pot concloure que, en aquesta ocasió, el número de columnes de les matrius sí és prou gran com per compensar

l'overhead de la comunicació entre processos. En canvi, una de las raons que pot explicar el pitjor rendiment del programa en el càlcul del valor de *nestedness* és l'*overhead* de la creació del processos. Així, aquesta optimització paral·lela permet compensar l'*overhead* de la gestió de processos en general, però només quan es crida vàries vegades a la funció `calculate_nested_value()`. Aquesta optimització paral·lela sí que es podria mantenir. Ara bé, la propera proposta dona resultats molt millors.

Finalment, el paral·lelitzat del bucle per inicialitzar les matrius d'individus i per calcular-ne el valor de *nestedness* és, de nou, amb el que s'obtenen els millors resultats per totes dues matrius. En aquest cas, l'*overhead* en la comunicació entre processos és molt menor, degut a que, en qualsevol dels cassos, hi ha menys directives de pas de missatges i aquestes s'executen una única vegada en tot el programa. D'entre totes les variants, amb la que s'obtenen els millors resultats és amb la que l'enviament de les variables `num_ones` i `nested_value` es realitzen amb la mateixa directiva `MPI_Bcast()` i es fa la reducció de la posició que la variable `nested_value` ocupa en el vector `nested_values`.

L'*speedup* d'aquesta última variant del bucle paral·lelitzat amb MPI respecte el codi seqüencial es proporciona a continuació:

$$Speedup\ MPI\ respecte\ seqüencial\ (portàtil) = \frac{67{,}158}{15{,}500} = 4{,}333$$

Aquest resultats mostren que l'*speedup* màxim aconseguit amb MPI és lleugerament superior a l'*speedup* màxim aconseguit amb OpenMP. Això significa que, tot i que la implementació del paral·lelisme amb MPI és més complexa i existeix un *overhead* important com a conseqüència de la gestió dels processos; si es selecciona l'estructura iterativa adient i es raona detingudament la comunicació entre processos més òptima, els resultats que es poden obtenir amb aquesta eina poden ser millors que amb OpenMP.

4.2.2.2 Orca

La versió del codi MPI que s'ha executat en la màquina Orca és, com en el cas d'OpenMP, la que dona el menor temps d'execució. En aquest cas, es tracta del paral·lelitzat del bucle per inicialitzar les randomitzacions i calcular-ne el valor de *nestedness*, amb l'enviament de les variables `num_ones` i `nested_value` en la mateixa directiva `MPI_Bcast()` i la reducció de la posició que la variable `nested_value` ocupa en el vector `nested_values`.

Propostes	**Temps d'execució (s)**	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
4 processos.	1,427	12,775
8 processos.	1,105	6,839
10 processos.	1,074	5,774
12 processos.	1,068	5,153
16 processos.	1,115	4,431

20 processos	1,216	3,581
30 processos.	1,483	3,604
32 processos.	1,533	3,493
34 processos.	1,584	3,788
64 processos.	2,567	5,507
128 processos.	5,194	11,152
256 processos. (*Oversubscribe*).	12,096	25,013

Propostes	***Speedup***	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
4 processos.	2,566	3,959
8 processos.	3,314	7,396
10 processos.	3,410	8,760
12 processos.	3,429	9,815
16 processos.	3,284	11,415
20 processos	3,012	14,124
30 processos.	2,469	14,034
32 processos.	2,389	14,480
34 processos.	2,312	13,352
64 processos.	1,427	9,184
128 processos.	0,705	4,535
256 processos. (*Oversubscribe*).	0,303	2,022

Els resultats obtinguts mostren que el menor temps d'execució del test de *nestedness* de la matriu de vertebrats s'obté quan es creen 12 processos, mentre que en l'execució del test de *nestedness* de la matriu d'individus, s'obté quan es creen 32 processos. En aquest cas, l'empitjorament del temps d'execució es comença a produir més tard que quan el codi es paral·lelitza utilitzant OpenMP. Això es degut a que, com la memòria principal dels processos creats amb MPI és privada, no es produeix el fenomen de *false sharing* que probablement s'hagi produït en l'execució del codi paral·lelitzat amb OpenMP.

També es força la màquina a entrar en un estat d'*oversubscribe*, en el que s'observa que l'empitjorament del rendiment del programa s'accentua substancialment.

L'*speedup* del test de *nestedness* de la matriu d'individus paral·lelitzat amb 32 processos d'MPI respecte el codi seqüencial en la màquina Orca es mostra a continuació:

$$Speedup\ MPI\ respecte\ seqüencial\ (orca) = \frac{50{,}578}{3{,}493} = 14{,}480$$

4.2.2.3 Pop

La versió del codi OpenMP que s'ha executat en la màquina Pop és, de nou, el paral·lelitzat del bucle per inicialitzar les randomitzacions i calcular-ne el valor de

nestedness, amb l'enviament de les variables `num_ones` i `nested_value` en la mateixa directiva `MPI_Bcast()` i la reducció de la posició que la variable `nested_value` ocupa en el vector `nested_values`.

Propostes	**Temps d'execució (s)**	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 Node. 4 processos.	11,684	151,683
8 Nodes. 4 processos.	13,627	150,017
8 Nodes. 8 processos.	9,768	78,579
8 Nodes. 12 processos.	8,566	56,569
8 Nodes. 16 processos.	8,013	46,526
8 Nodes. 20 processos.	7,427	35,917
8 Nodes. 32 processos.	7,080	29,272
8 Nodes. 64 processos. (*Oversubscribe*).	8,356	48,341
8 Nodes. 128 processos. (*Oversubscribe*).	10,984	86,660

Propostes	***Speedup***	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 Node. 4 processos.	0,490	0,501
8 Nodes. 4 processos.	0,420	0,507
8 Nodes. 8 processos.	0,586	0,967
8 Nodes. 12 processos.	0,668	1,344
8 Nodes. 16 processos.	0,714	1,634
8 Nodes. 20 processos.	0,771	2,116
8 Nodes. 32 processos.	0,809	2,596
8 Nodes. 64 processos. (*Oversubscribe*).	0,685	1,572
8 Nodes. 128 processos. (*Oversubscribe*).	0,521	0,877

Els resultats del programa paral·lelitzat amb MPI en la màquina Pop són molt pitjors que els del programa paral·lelitzat amb OpenMP. A més, també són molt pitjors que en la resta de màquines. Tot i que, de totes les màquines utilitzades, Pop és la que té els processadors amb les pitjor prestacions, aquests resultats són inesperats. En un principi, s'ha pensat que aquests temps d'execució tant elevats podien ser conseqüència de l'*overhead* de l'enviament dels processos als diferents nodes. No obstant, els resultats la proposta de la primera fila, en la que s'executen 4 processos en el mateix node, desmenteix aquesta hipòtesi. Així, l'explicació més probable és que la creació dels processos i la gestió de la comunicació entre ells penalitza considerablement el rendiment de la màquina Pop. Tot i així, els resultats segueixen sent estranys, ja que, com s'ha explicat en la fase de disseny i implementació, aquesta versió del codi només executa dues directives de pas de missatges, a través de les quals s'envien únicament tres dades.

Els millors resultats per totes dues matrius s'han obtingut amb 32 processos, és a dir, amb la utilització de totes les unitats d'execució de les que disposa la màquina Pop. En aquest cas, s'han afegit dos proves en les que es força la màquina a entrar en un estat d'*oversubscribe*. Tot i que s'observa un empitjorament en els temps d'execució respecte a la versió més òptima, a diferència de amb la màquina orca, els temps d'execució no són tan elevats com amb els d'aquelles execucions en les que es crea un menor número de processos.

L'*speedup* del test de *nestedness* de la matriu d'individus paral·lelitzat amb 32 processos d'MPI respecte el codi seqüencial en la màquina Pop es mostra a continuació:

$$Speedup\ MPI\ \ respecte\ seqüencial\ (pop) = \frac{76{,}003}{29{,}272} = 2{,}59$$

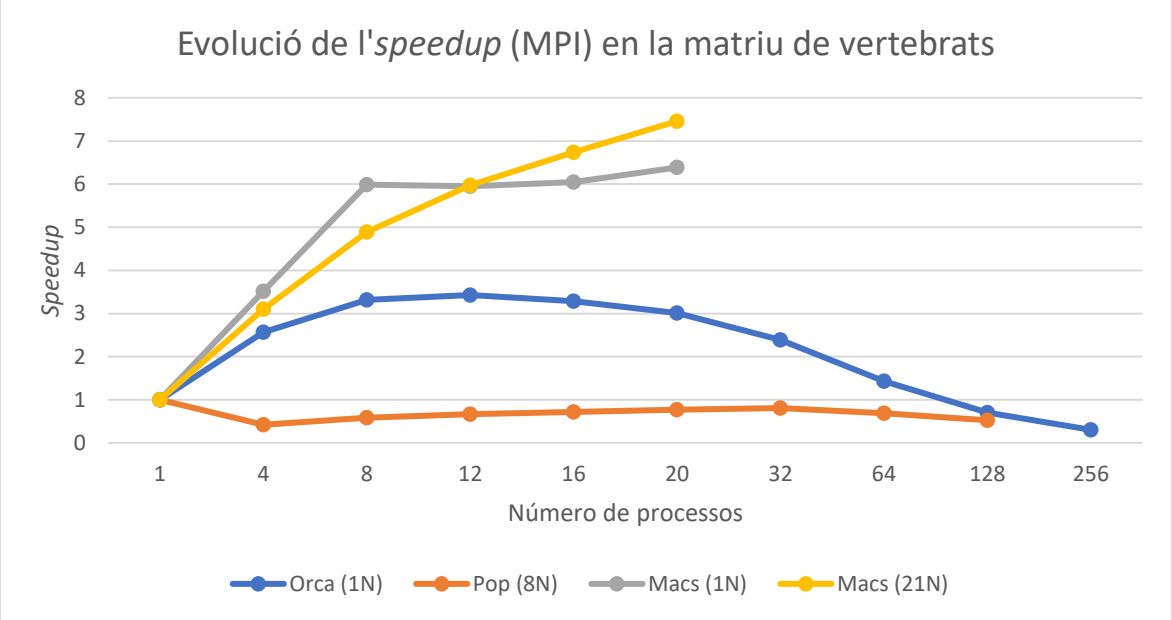
4.2.2.4 Sala dels Mac

Propostes	**Temps d'execució (s)**	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 Node. 2 processos.	2,355	33,607
1 Node. 4 processos.	1,288	17,447
1 Node. 8 processos.	0,756	9,142
1 Node. 12 processos.	0,761	7,970
1 Node. 16 processos.	0,749	8,467
1 Node. 20 processos.	0,709	8,609
1 Node. 40 processos. (*Oversubscribe*).	0,926	10,168
2 Nodes. 2 processos.	2,575	40,421
4 Nodes. 4 processos.	1,461	16,900
8 Nodes. 8 processos.	0,926	9,246
12 Nodes. 12 processos.	0,758	6,519
16 Nodes. 16 processos.	0,672	5,197
20 Nodes. 20 processos.	0,607	3,829
21 Nodes. 21 processos.	0,647	4,565
2 Nodes. 8 processos.	0,741	5,395
4 Nodes. 8 processos.	0,599	3,236
8 Nodes. 8 processos.	0,709	4,437
12 Nodes. 8 processos.	0,794	4,292
16 Nodes. 8 processos.	1,118	8,384
20 Nodes. 8 processos.	-	-
21 Nodes. 8 processos.	-	-

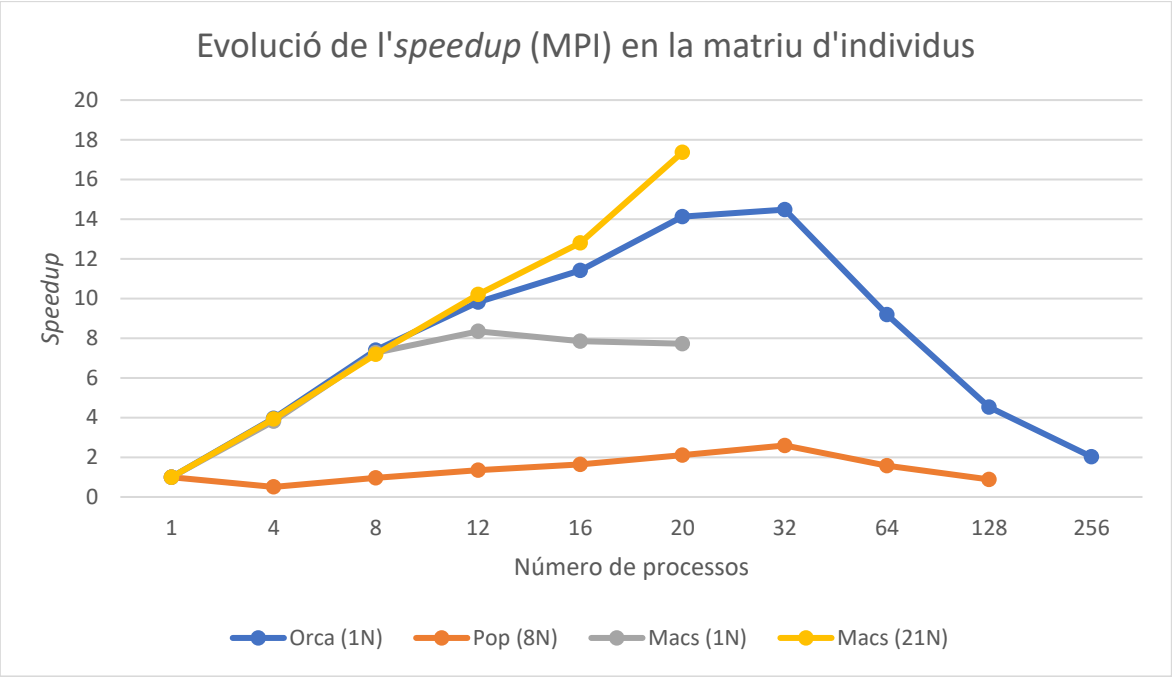
Propostes	***Speedup***	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
1 Node. 2 processos.	1,923	1,979
1 Node. 4 processos.	3,516	3,812

1 Node. 8 processos.	5,989	7,275
1 Node. 12 processos.	5,950	8,345
1 Node. 16 processos.	6,045	7,855
1 Node. 20 processos.	6,386	7,726
1 Node. 40 processos. (*Oversubscribe*).	4,890	6,541
21 Nodes. 2 processos.	1,758	1,645
21 Nodes. 4 processos.	3,099	3,935
21 Nodes. 8 processos.	4,890	7,194
21 Nodes. 12 processos.	5,974	10,203
21 Nodes. 16 processos.	6,738	12,798
21 Nodes. 20 processos.	7,460	17,370
21 Nodes. 21 processos.	6,998	14,570
2 Nodes. 8 processos.	6,111	12,328
4 Nodes. 8 processos.	7,559	20,553
8 Nodes. 8 processos.	6,386	14,990
12 Nodes. 8 processos.	5,703	15,497
16 Nodes. 8 processos.	4,050	7,933
20 Nodes. 8 processos.	-	-
21 Nodes. 8 processos.	-	-

$$Speedup\ MPI\ respecte\ seqüencial\ (1\ node\ Mac) = \frac{66{,}512}{7{,}970} = 8{,}345$$



Gràfic 8. Evolució de l'*speedup* en la matriu de vertebrats quan es paral·lelitza amb MPI.



Gràfic 9. Evolució de l'*speedup* en la matriu d'individus quan es paral·lelitza amb MPI.

4.2.3 MPI i OpenMP

la versió del codi que combina MPI amb OpenMP no s'ha executat en el meu ordinador portàtil perquè no s'ha estat treballant amb suficients unitats d'execució *hardware* per a que la comparativa tingui sentit. La versió que s'ha executat, tant en la màquina Orca com en la màquina Pop, combina les següents optimitzacions paral·leles. Per la part d'MPI, el paral·lelitzat del bucle per inicialitzar les randomitzacions i calcular-ne el valor de *nestedness* amb l'enviament de les variables `num_ones` i `nested_value` en la mateixa directiva `MPI_Bcast()` i la reducció de les posicions que la variable `nested_value` ocupa en els vectors `nested_values`. Amb OpenMP, el paral·lelitzat de l'estructura anidada per calcular les components de l'equació relatives a les columnes amb repartiment dinàmic de les iteracions.

4.2.3.1 Orca

Propostes	**Temps d'execució (s)**	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
8 processos. 8 *threads*/procés. 64 *threads* totals.	0,767	4,717
16 processos. 4 *threads*/procés. 64 *threads* totals.	0,969	3,570
32 processos. 2 *threads*/procés. 64 *threads* totals.	1,497	3,375
16 processos. 8 *threads*/procés. 128 *threads* totals.	0,995	3,410

32 processos. 4 *threads*/procés. 128 *threads* totals.	1,493	3,223
64 processos. 2 *threads*/procés. 128 *threads* totals.	2,512	5,258

Propostes	***Speedup***	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
8 processos. 8 *threads*/procés. 64 *threads* totals.	4,774	10,722
16 processos. 4 *threads*/procés. 64 *threads* totals.	3,779	14,168
32 processos. 2 *threads*/procés. 64 *threads* totals.	2,446	14,986
16 processos. 8 *threads*/procés. 128 *threads* totals.	3,680	14,832
32 processos. 4 *threads*/procés. 128 *threads* totals.	2,453	15,693
64 processos. 2 *threads*/procés. 128 *threads* totals.	1,458	9,619

En la màquina Orca, els resultats mostren que els millors temps d'execució s'obtenen quan el número de *threads* es modula a partir del número òptim de processos. Explicat d'una altra manera, el procediment correcte consisteix en obtenir el número de processos MPI amb els que s'obté el millor temps d'execució i després variar el número de *threads*. D'aquesta manera, els problemes que es produeixin com a conseqüència del *false sharing* seran mínims.

Per tant, el test de *nestedness* de la matriu de vertebrats que, quan s'ha paral·lelitzat utilitzat únicament MPI ha aconseguit el seu millor rendiment amb la creació de 12 processos, ara té el seu menor temps d'execució quan es creen 8 processos i 8 *threads*. En el cas del test de *nestedness* de la matriu d'individus que, quan s'ha paral·lelitzat utilitzat únicament MPI ha aconseguit el seu millor rendiment amb la creació de 32 processos, ara té el seu menor temps d'execució quan es creen 32 processos i 4 *threads*. Els resultats per la matriu d'individus són especialment rellevants, perquè mostren que **la utilització conjunta d'MPI i OpenMP permet aprofitar les 128 unitats d'execució *hardware* de les que disposa la màquina orca**, cosa que no s'ha pogut aconseguir amb la utilització de cap de les dues eines per separat.

L'*speedup* del test de *nestedness* de la matriu d'individus paral·lelitzat amb 32 processos d'MPI i 4 *threads* d'OpenMP respecte el codi seqüencial en la màquina Orca és mostra a continuació:

$$Speedup\ MPI\ i\ OpenMP\ respecte\ seqüencial\ (orca) = \frac{50{,}578}{3{,}223} = 15{,}692$$

4.2.3.2 Pop

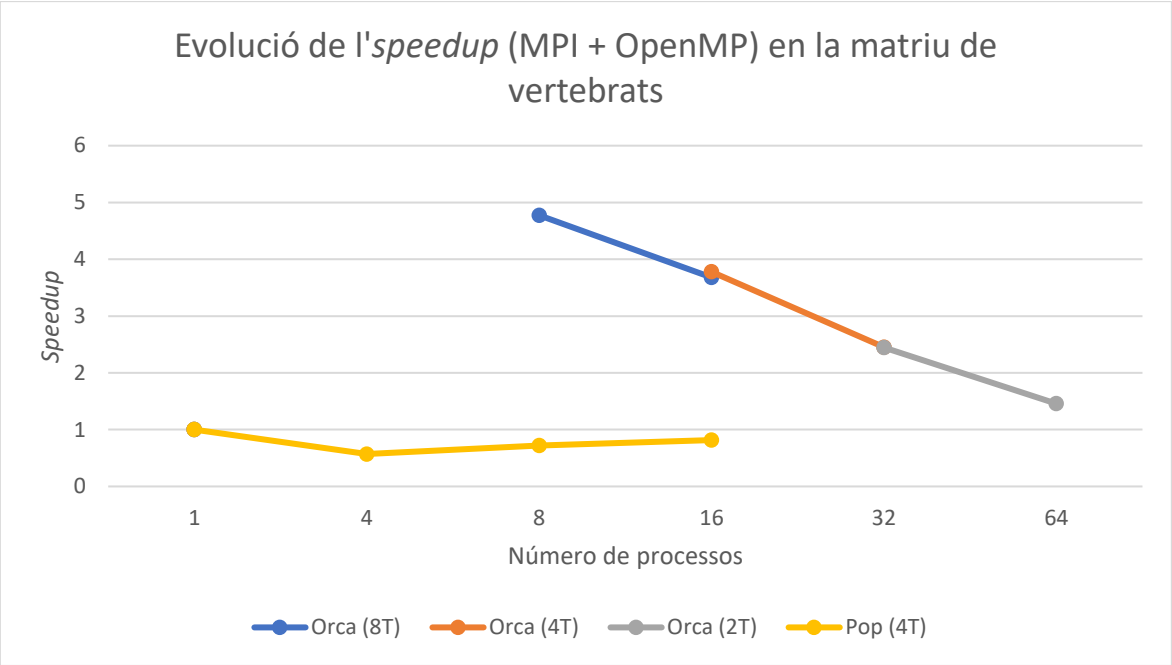
Propostes	**Temps d'execució (s)**	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
8 Nodes. 4 processos. 4 *threads*/procés. 16 *threads* totals.	10,028	106,68
8 Nodes. 8 processos. 4 *threads*/procés. 32 *threads* totals.	7,929	56,723
8 Nodes. 16 processos. 4 *threads*/procés. 64 *threads* totals.	6,995	34,546

Propostes	***Speedup***	
	Test de *nestedness* (1000 random.)	
	Matriu de vertebrats (50 x 1056)	**Matriu d'individus (644 x 1056)**
8 Nodes. 4 processos. 4 *threads*/procés. 16 *threads* totals.	0,571	0,712
8 Nodes. 8 processos. 4 *threads*/procés. 32 *threads* totals.	0,722	1,340
8 Nodes. 16 processos. 4 *threads*/procés. 64 *threads* totals.	0,818	2,200

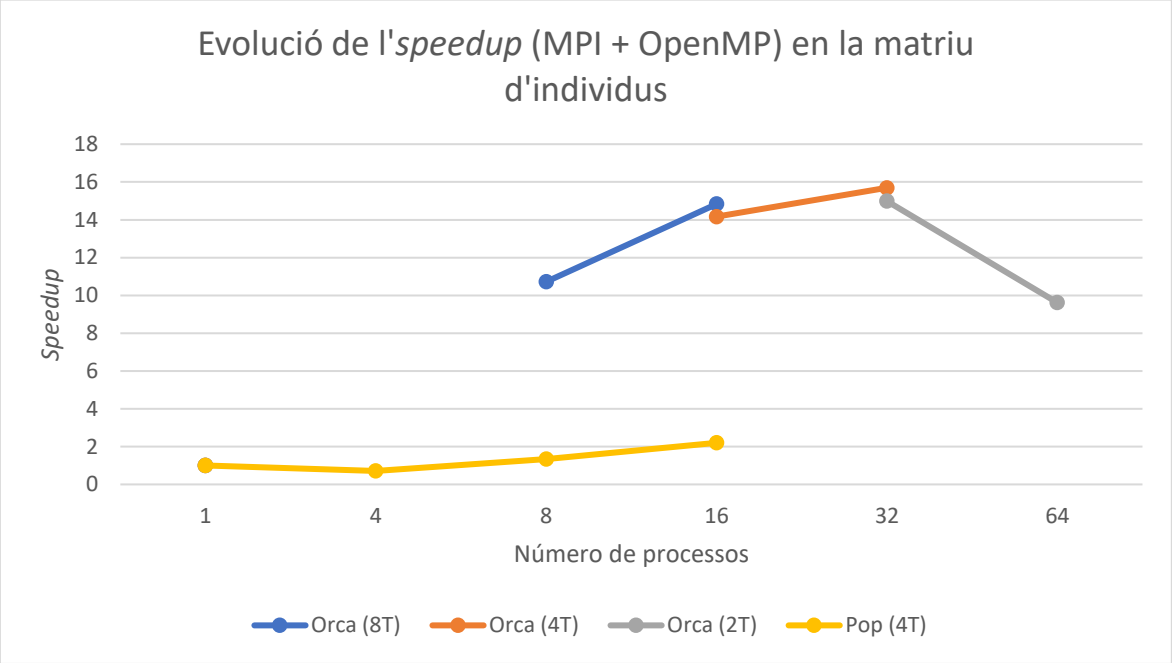
En la màquina Pop els resultats són els oposats als obtinguts amb la màquina orca. En aquest cas, la utilització d'MPI i OpenMP per a crear un combinació determinada de processos i *threads* dona pitjors temps d'execució per totes dues matrius que si es creen la mateixa quantitat de processos utilitzant MPI. Excepcionalment, si es força la màquina a entrar en estat d'*oversubscribe*, el rendiment del programa és millor. De fet, el millor temps d'execució per les dues matrius s'ha obtingut amb la creació de 16 processos i 4 *threads*.

L'*speedup* del test de *nestedness* de la matriu d'individus paral·lelitzat amb 16 processos d'MPI i 4 *threads* d'OpenMP respecte el codi seqüencial en la màquina Pop és mostra a continuació:

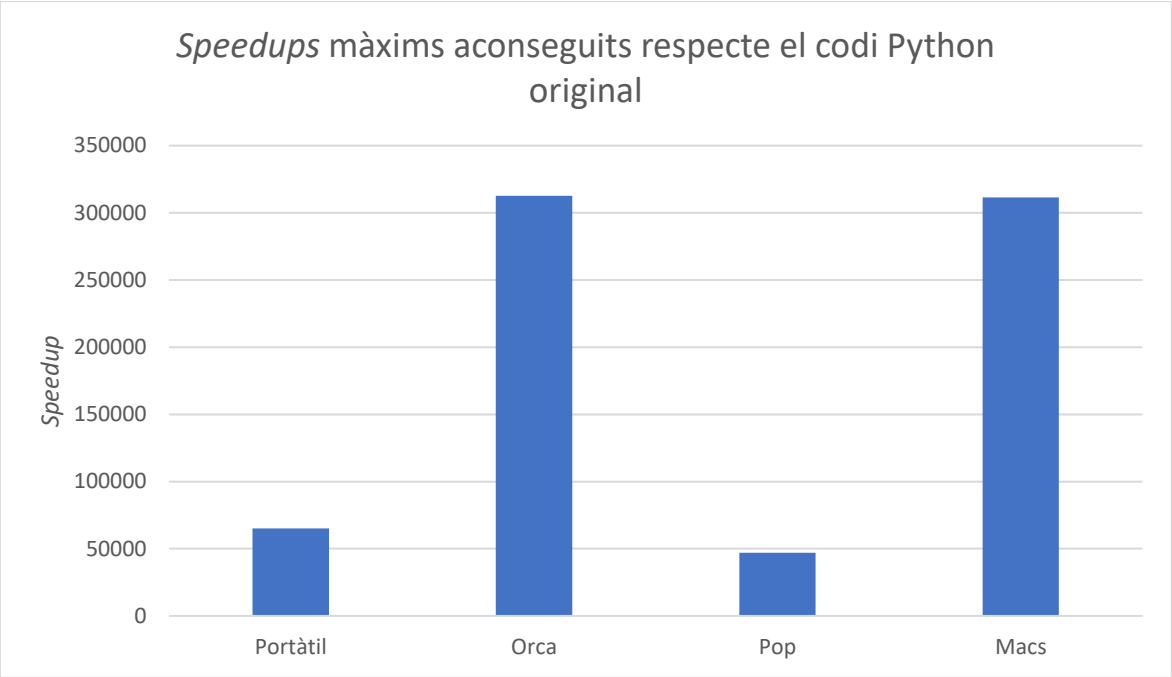
$$Speedup\ MPI\ i\ OpenMP\ respecte\ seqüencial\ (pop) = \frac{76{,}003}{34{,}546} = 2{,}200$$



Gràfic 10. Evolució de l'*speedup* en la matriu de vertebrats quan es paral·lelitza amb MPI i OpenMP.



Gràfic 11. Evolució de l'*speedup* en la matriu d'individus quan es paral·lelitza amb MPI i OpenMP.



Gràfic 12. *Speedups* màxims aconseguits en cada màquina respecte el codi Python original.

5 Conclusions i perspectives de futur

En primer lloc, s'ha de destacar que s'ha assolit l'objectiu que va motivar el desenvolupament d'aquest projecte. Així, els resultats obtinguts proven que **les dues matrius d'abundàncies relatives són *nested***. D'aquesta manera, s'obre una nova finestra en la recerca relacionada amb les microbiotes intestinals segons la qual els gèneres bacterians que hi podem trobar no només s'organitzen segons aquest patró en els individus de la mateixa espècie, sinó també en els individus de diferents espècies que pertanyen al mateix subfílum.

Per altra banda, ha quedat demostrada la capacitat que tenen les eines utilitzades, per generar codis més eficients. Així, encara que el llenguatge de programació Python abstreu al programador de moltes de les complexitats de la gestió de la memòria i això facilita el desenvolupament de codi, la millora en el rendiment que té l'algorisme si s'implementa amb el llenguatge de programació C i es compila aplicant tercer nivell d'optimització és absoluta. Només amb aquest canvi, ja es van aconseguir els resultats del primer paràgraf.

Igualment, aquest programa ha mostrat un elevat potencial de millora amb la aplicació de paral·lelisme, especialment si es treballa amb el volum de dades més gran. Ara bé, es important seleccionar adequadament la secció del codi que es vol executar de forma paral·lela. En aquest cas, el paral·lelitzat del bucle encarregat d'inicialitzar les randomitzacions i calcular-ne el valor de *nestedness*. Tant amb OpenMP com amb MPI el paral·lelitzat d'aquest bucle dona *speedups* propers al màxim teòric utilitzant 5 unitats d'execució *hardware* en el meu ordinador portàtil.

Els resultats si s'executa el programa amb aquesta optimització paral·lela en la resta de màquines també són interessants. En el cas de Orca, tant OpenMP com, sobre tot, MPI donen millores substancials en el rendiment del programa. A més, si es combinen les dues eines, s'aconsegueix aprofitar les 128 unitats d'execució *hardware* de les que disposa aquesta màquina i obtenir els resultats en 3,223 segons. Amb els Mac, tot i que el temps d'execució màxim és una mica més alt perquè els processadors són menys potents que el d'Orca, amb la creació de 8 processos repartits en 4 nodes, s'aconsegueix l'*speedup* més alt aconseguit respecte a la versió seqüencial executada en la mateixa màquina que és 20,553. Per altra banda, la màquina Pop ha donat un rendiment molt baix amb qualsevol de les variants que utilitzen MPI. Si el paral·lelisme s'implementa només amb OpenMP, els resultats són millors, però com a màxim es podran crear quatre *threads* i els temps d'execució segueixen sent més alts que els de la resta de màquines.

Tot i els bons resultats obtinguts amb les eines utilitzades, encara existeixen molts altres mecanismes a través dels quals encara es podria tractar de millorar el rendiment del programa. Per la part de l'optimització seqüencial, es podria redissenyar l'algorisme per calcular el valor de *nestedness* d'una matriu binaria de manera encara més eficient. En concret, es possible que es puguin utilitzar els vectors `sum_rows` i `sum_cols` no només per obtenir les dues components de l'equació calculades a partir de les interaccions de les files i de les interaccions de les columnes, sinó també les dues components calculades a partir de les interaccions compartides entre les files i de les interaccions compartides entre les columnes [75]. Caldria fer més recerca en aquest aspecte. A més, es pot millorar l'eficiència del funcionament de la jerarquia de memòria utilitzant tècniques de ***cache blocking optimization*** [76].

Per la part de l'optimització paral·lela seria interessant provar el rendiment del programa si s'utilitzés **Cuda**. Aquesta eina consisteix en un model de programació i una plataforma de computació paral·lela per executar codi C en les GPUs de NVIDIA [77]. L'objectiu es aprofitar l'elevat poder de computació per a fer càlculs del que disposen les targetes gràfiques [77]. No obstant, si es decidís utilitzar aquesta eina caldria recordar que una de les principals raons per les que aquest algorisme tenia un temps d'execució tan alt és per la quantitat d'accessos a memòria que cal fer i, per tant, no s'hauria d'esperar una millora tant gran com la que es produeix en altres programes.

Finalment, una última opció interessant seria utilitzar **serveis *Cloud***. La **llei de Gustafson** proposa que, a mesura que s'augmenta el *hardware* per paral·lelitzar, més gran serà el volum de dades per les que un codi paral·lel podrà obtenir els resultats [78]. Una de les característiques fonamentals dels serveis *Cloud* és la seva capacitat per **adaptar-se dinàmicament a la demanda**. Per tant, a mesura que la quantitat de dades d'entrada de un programa augmenti es podran aixecar més nodes perquè el rendiment no es vegi perjudicat. En aquest cas, és factible utilitzar **Lithops**, que és un marc de computació distribuïda en el Cloud per executar codi Python en una plataforma *serverless* [79], per variar la quantitat de dades o el número de randomitzacions que l'algorisme hagi de tractar mentre el temps d'execució es manté estable [80].

Referències

[1] Cobo-López, S., Gupta, V. K., Sung, J., Guimerà, R., & Sales-Pardo, M. (2022). *Stochastic block models reveal a robust nested pattern in healthy human gut microbiomes*. PNAS nexus, 1(3), pgac055. https://doi.org/10.1093/pnasnexus/pgac055

[2] Singh, R. K., Chang, H. W., Yan, D., Lee, K. M., Ucmak, D., Wong, K., Abrouk, M., Farahnik, B., Nakamura, M., Zhu, T. H., Bhutani, T., & Liao, W. (2017). *Influence of diet on the gut microbiome and implications for human health*. Journal of translational medicine, 15(1), 73. https://doi.org/10.1186/s12967-017-1175-y

[3] Gilbert, J. A., & Lynch, S. V. (2019). *Community ecology as a framework for human microbiome research*. Nature medicine, 25(6), 884–889. https://doi.org/10.1038/s41591-019-0464-9

[4] Costea, P. I., Hildebrand, F., Arumugam, M., Bäckhed, F., Blaser, M. J., Bushman, F. D., de Vos, W. M., Ehrlich, S. D., Fraser, C. M., Hattori, M., Huttenhower, C., Jeffery, I. B., Knights, D., Lewis, J. D., Ley, R. E., Ochman, H., O'Toole, P. W., Quince, C., Relman, D. A., Shanahan, F., … Bork, P. (2018). *Enterotypes in the landscape of gut microbial community composition*. Nature microbiology, 3(1), 8–16. https://doi.org/10.1038/s41564-017-0072-8

[5] McFall-Ngai, M., Hadfield, M. G., Bosch, T. C., Carey, H. V., Domazet-Lošo, T., Douglas, A. E., Dubilier, N., Eberl, G., Fukami, T., Gilbert, S. F., Hentschel, U., King, N., Kjelleberg, S., Knoll, A. H., Kremer, N., Mazmanian, S. K., Metcalf, J. L., Nealson, K., Pierce, N. E., Rawls, J. F.,… Wernegreen, J. J. (2013). *Animals in a bacterial world, a new imperative for the life sciences*. Proceedings of the National Academy of Sciences of the United States of America, 110(9), 3229– 3236. https://doi.org/10.1073/pnas.1218525110

[6] Lynch, J. B., & Hsiao, E. Y. (2019). *Microbiomes as sources of emergent host phenotypes*. Science (New York, N.Y.), 365(6460), 1405–1409. https://doi.org/10.1126/science.aay0240

[7] Levin, D., Raab, N., Pinto, Y., Rothschild, D., Zanir, G., Godneva, A., Mellul, N., Futorian, D., Gal, D., Leviatan, S., Zeevi, D., Bachelet, I., & Segal, E. (2021). *Diversity and functional landscapes in the microbiota of animals in the wild*. Science (New York, N.Y.), 372(6539), eabb5352. https://doi.org/10.1126/science.abb5352

[8] Gibson, K. M., Nguyen, B. N., Neumann, L. M., Miller, M., Buss, P., Daniels, S., Ahn, M. J., Crandall, K. A., & Pukazhenthi, B. (2019). *Gut microbiome differences between wild and captive black rhinoceros - implications for rhino health*. Scientific reports, 9(1), 7570. https://doi.org/10.1038/s41598-019-43875- 3

[9] Cheng, Y., Fox, S., Pemberton, D., Hogg, C., Papenfuss, A. T., & Belov, K. (2015). *The Tasmanian devil microbiome-implications for conservation and management*. Microbiome, 3, 76. https://doi.org/10.1186/s40168-015-0143-0

[10] Alberdi, A., Martin Bideguren, G., & Aizpurua, O. (2021). *Diversity and compositional changes in the gut microbiota of wild and captive vertebrates: a meta-analysis*. Scientific reports, 11(1), 22660. https://doi.org/10.1038/s41598- 021-02015-6

[11] Ferrero, A. (2023). https://github.com/AreyFerreroRamos/TFGBiotecnologia

[12] Patterson, B. D., & Atmar, W. (1986). *Nested subsets and the structure of insular mammalian faunas and archipelagos*. Biological Journal of the Linnean Society, 28(1-2), 65–82. https://doi.org/10.1111/j.1095-8312.1986.tb01749.x

[13] Poulin, R., & Valtonen, E. T. (2001). *Nested assemblages resulting from host size variation: the case of endoparasite communities in fish hosts*. International journal for parasitology, 31(11), 1194–1204. https://doi.org/10.1016/s0020- 7519(01)00262-4

[14] Warburton, E. M., Van Der Mescht, L., Khokhlova, I. S., Krasnov, B. R., & Vonhof, M. J. (2018). *Nestedness in assemblages of helminth parasites of bats: a function of geography, environment, or host nestedness?*. Parasitology research, 117(5), 1621–1630. https://doi.org/10.1007/s00436-018-5844-4

[15] Bastolla, U., Fortuna, M. A., Pascual-García, A., Ferrera, A., Luque, B., & Bascompte, J. (2009). *The architecture of mutualistic networks minimizes competition and increases biodiversity*. Nature, 458(7241), 1018–1020. https://doi.org/10.1038/nature07950

[16] Mariani, M. S., Ren, Z. M., Bascompte, J., & Tessone, C. J. (2019). *Nestedness in complex networks: observation, emergence, and implications*. Physics and Society, 813, 1–90. https://doi.org/10.1016/j.physrep.2019.04.001

[17] Molina Arias, M. (2017) *¿Qué significa realmente el valor de p?*. Rev Pediatr Aten Primary [en línea], vol. 19, n. 76, págs.377-381. ISSN 1139-7632. http://scielo.isciii.es/scielo.php?script=sci_arttext&pid=S1139-76322017000500014&lng=es&nrm=iso

[18] https://elchapuzasinformatico.com/2019/06/asus-rog-strix-g531g-review/. [Consultada el 3 de gener del 2024].

[19] https://rog.asus.com/es/laptops/rog-strix/rog-strix-g-g531-series/spec/. [Consultada el 3 de gener del 2024].

[20] https://www.virtualbox.org/. [Consultada el 3 de gener del 2024].

[21] https://web.archive.org/web/20180502065355/https://www.ubuntu.com/community/debian. [Consultada el 3 de gener del 2024].

[22] https://www.compuhoy.com/deberia-aprender-python-en-windows-o-linux/. [Consultada el 2 de gener del 2024].

[23] https://stackoverflow.com/questions/3765178/running-python-on-a-windows-machine-vs-linux. [Consultada el 2 de gener del 2024].

[23] https://forum.boltiot.com/t/why-do-we-use-python-on-linux-and-not-on-windows/4648/7. [Consultada el 2 de gener del 2024].

[25] https://git-scm.com/. [Consultada el 2 de gener del 2024].

[26] https://en.wikipedia.org/wiki/GitHub. [Consultada el 2 de gener del 2024].

[27] https://github.com/AreyFerreroRamos/TFGInformaticsEngineering.

[28] Hamilton, Naomi. (2008). *The A-Z of Programming Languages: BASH/Bourne-Again Shell*. Computerworld. https://a-z.readthedocs.io/en/latest/bash.html

[29] Johnson, Chris. (2009*). Pro Bash Programming: Scripting the Linux Shell.* Apress.

[30] Kuhlman, Dave. (2012). *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html.

[31] https://www.python.org/doc/essays/blurb/. [Consultada el 2 de gener del 2024].

[32] https://docs.python.org/3/howto/functional.html. [Consultada el 3 de gener del 2024].

[33] https://www.w3schools.com/datascience/ds_python.asp. [Consultada el 3 de gener del 2024].

[34] https://www.kdnuggets.com/2018/05/poll-tools-analytics-data-science-machine-learning-results.html/2. [Consultada el 3 de gener del 2024].

[35] https://www.w3schools.com/python/pandas/pandas_intro.asp. [Consultada el 2 de gener del 2024].

[36] https://numpy.org/doc/stable/user/whatisnumpy.html. [Consultada el 2 de gener del 2024].

[37] https://www.pypy.org/. [Consultada el 5 de gener del 2024].

[38] https://www.jetbrains.com/es-es/pycharm/. [Consultada el 2 de gener del 2024].

[39] https://www.techtarget.com/searchwindowsserver/definition/C. [Consultada el 2 de gener del 2024].

[40] https://www.guru99.com/c-programming-language.html. [Consultada el 3 de gener del 2024].

[41] https://es.wikipedia.org/wiki/C_(lenguaje_de_programaci%C3%B3n)#cite_note-K&R2_esp_1991-2. [Consultada el 3 de gener del 2024].

[42] https://www.timetoast.com/timelines/origen-de-los-lenguajes-de-programacion-d00b83c0-1c46-4d09-a47b-9583332a7933. [Consultada el 3 de gener del 2024].

[43] Kernighan, B. W., & Ritchie, D. M. (1991). *El lenguaje de programación C* (2ª edición). Prentice Hall Hispanoamericana.

[44] https://www.developer.com/news/c-language-drops-to-lowest-popularity-rating/. [Consultada el 3 de gener del 2024].

[45] https://www.tiobe.com/tiobe-index/. [Consultada el 13 de gener del 2024].

[46] https://en.wikipedia.org/wiki/C_(programming_language). [Consultada el 3 de gener del 2024].

[47] https://peter-jp-xie.medium.com/how-slow-is-python-compared-to-c-3795071ce82a#:~:text=It%20is%20450%20million%20loops,mode%20for%20a%20better%20performance.&text=Yes%2C%20it%20is%20unbelievable!,45%2C000%20times%20faster%20than%20Python. [Consultada l'11 de gener del 2024].

[48] https://www.jetbrains.com/es-es/clion/. [Consultada el 2 de gener del 2024].

[49] https://blog.mattjustice.com/2020/11/24/gprof-profiler/. [Consultada l'11 de gener del 2024].

[50] https://users.cs.duke.edu/~ola/courses/programming/gprof.html. [Consultada l'11 de gener del 2024].

[51] https://passlab.github.io/OpenMPProgrammingBook/openmp_c/1_IntroductionOfOpenMP.html. [Consultada el 3 de gener del 2024].

[52] https://carleton.ca/rcs/rcdc/introduction-to-openmp/. [Consultada el 3 de gener del 2024].

[53] https://carleton.ca/rcs/rcdc/introduction-to-mpi/. [Consultada el 3 de gener del 2024].

[54] https://www.amd.com/es/support/cpu/amd-ryzen-pro-processors/amd-ryzen-threadripper-pro-processors/amd-ryzen-threadripper. [Consultada el 10 de gener del 2024].

[55] https://www.intel.com/content/www/us/en/products/sku/64596/intel-xeon-processor-e52690-20m-cache-2-90-ghz-8-00-gts-intel-qpi/specifications.html [Consultada el 10 de gener del 2024]

[56] https://www.cpu-world.com/CPUs/K8/AMD-Second%20Generation%20Opteron%202210%20-%20OSA2210GAA6CQ%20(OSA2210CQWOF).html. [Consultada l'11 de gener del 2024] .

[57] https://www.intel.la/content/www/xl/es/products/sku/204448/intel-core-i910910-processor-20m-cache-up-to-5-00-ghz/specifications.html. [Consultada el 10 de gener del 2024].

[58] https://www.intel.la/content/www/xl/es/products/sku/199335/intel-core-i710700k-processor-16m-cache-up-to-5-10-ghz/specifications.html. [Consultada el 10 de gener del 2024].

[59] https://github.com/AreyFerreroRamos/MetodologiasProgramacion/tree/main/Practica2/. [Consultada el 5 de gener del 2024].

[60] https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html. [Consultada el 5 de gener del 2024].

[61] https://stackoverflow.com/questions/11607387/is-there-a-c-c-api-for-python-pandas. [Consultada el 5 de gener del 2024].

[62] https://www.ibm.com/docs/es/i/7.5?topic=functions-fgets-read-string. [Consultada el 5 de gener del 2024].

[63] https://www.ibm.com/docs/es/i/7.5?topic=functions-strtok-tokenize-string. [Consultada el 5 de gener del 2024].

[64] https://stackoverflow.com/questions/10482974/why-is-stack-memory-size-so-limited. [Consultada el 5 de gener del 2024].

[65] https://www.ibm.com/docs/en/zos/2.1.0?topic=functions-malloc-reserve-storage-blocke. [Consultada el 5 de gener del 2024].

[66] https://www.ibm.com/docs/en/i/7.1?topic=functions-free-release-storage-blocks. [Consultada el 5 de gener del 2024].

[67] https://www.ibm.com/docs/es/i/7.5?topic=functions-calloc-reserve-initialize-storage. [Consultada el 5 de gener del 2024].

[68] https://www.ibm.com/docs/es/i/7.5?topic=functions-memset-set-bytes-value. [Consultada el 5 de gener del 2024].

[69] https://numpy.org/devdocs/user/basics.indexing.html. [Consultada el 4 de gener del 2024].

[70] https://docs.python.org/3.3/library/functions.html. [Consultada el 4 de gener del 2024].

[71] Amdahl, G. (1967). *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*. AFIPS Conference Proceedings.

[72] https://www.openmp.org/spec-html/5.0/openmpsu40.html. [Consultada el 16 de gener del 2024].

[73] https://www.openmp.org/spec-html/5.0/openmpsu112.html. [Consultada el 16 de gener del 2024].

[74] https://www.openmp.org/spec-html/5.0/openmpsu113.html. [Consultada el 16 de gener del 2024].

[75] https://github.com/AreyFerreroRamos/TFGInformaticsEngineering/pull/1. [Consultada el 19 de gener del 2024].

[76] https://www.intel.com/content/www/us/en/developer/articles/technical/cache-blocking-techniques.html. [Consultada el 19 de gener del 2024].

[77] https://blogs.nvidia.com/blog/what-is-cuda-2/. [Consultada el 19 de gener del 2024].

[78] http://www.johngustafson.net/pubs/pub13/amdahl.htm. [Consultada el 19 de gener del 2024].

[79] https://github.com/lithops-cloud/lithops. [Consultada el 19 de gener del 2024].

[80] https://github.com/AreyFerreroRamos/TFGInformaticsEngineering/pull/2. [Consultada el 19 de gener del 2024].

6 Annexes

6.1 Annex 1: Codi en Python per discretitzar una matriu

```
def discretize_matrix(matrix, threshold):
    rows = 0
    while rows < matrix.shape[0]:
        columns = 0
        while columns < matrix.shape[1]:
            if matrix[rows][columns] > threshold:
                matrix[rows][columns] = 1
            else:
                matrix[rows][columns] = 0
            columns += 1
        rows += 1
```

6.2 Annex 2: Codi en Python per realitzar el test de *nestedness*

```
def count_ones_binary_matrix(matrix):
    num_ones = 0

    for row in range(matrix.shape[0]):
        for column in range(matrix.shape[1]):
            if matrix[row][column] == 1:
                num_ones += 1

    return num_ones



def generate_nested_values_randomized(matrix, num_randomized_matrices):
    nested_values_randomized = []
    num_ones = count_ones_binary_matrix(matrix)

    for i in range(num_randomized_matrices):
       randomized_matrix = np.zeros((matrix.shape[0], matrix.shape[1]), dtype=int)
       randomized_matrix.ravel()[np.random.choice(matrix.shape[0]                *
       matrix.shape[1], num_ones, replace=False)] = 1
        nested_values_randomized.append(nestedness_optimized(randomized_matrix))
```

```
    return nested_values_randomized



def nestedness_test(matrix, num_randomized_matrices):
    if matrix.size == 0:
        return 0, 0


      # Generate as many randomized matrices from the real matrix as it is specified
      by parameter
      # and calculate their nestedness value.
    nested_values = generate_nested_values_randomized(matrix,
    num_randomized_matrices)


      # Calculate the nestedness value of the real matrix.
    nested_value = nestedness(matrix)
    nested_values.append(nested_value)


      # Sort the list of nestedness values.
    nested_values.sort()


      # Calculate the fraction of randomized matrices that have a nestedness value
      greater than that of the real matrix.
    p_value = (num_randomized_matrices - nested_values.index(nested_value)) /
    (num_randomized_matrices + 1)


    return nested_value, p_value
```

6.3 Annex 3: Implementació original de l'algorisme per calcular el valor de *nestedness* d'una matriu binària en Python

```
def nestedness(matrix):
    first_isocline = 0
    second_isocline = 0
    third_isocline = 0
    fourth_isocline = 0


      # Calculate the sum of the number of shared interactions between rows.
    for first_row in range(matrix.shape[0]):
        for second_row in range(matrix.shape[0]):
            if first_row < second_row:
```

```
            for col in range(matrix.shape[1]):
                if matrix[first_row][col] == 1 and matrix[second_row][col] == 1:
                    first_isocline += 1

    # Calculate the sum of the number of shared interactions between columns.
for first_col in range(matrix.shape[1]):
    for second_col in range(matrix.shape[1]):
        if first_col < second_col:
            for row in range(matrix.shape[0]):
                if matrix[row][first_col] == 1 and matrix[row][second_col] == 1:
                    second_isocline += 1

    # Calculate the sum of the number of interactions of rows.
for first_row in range(matrix.shape[0]):
    for second_row in range(matrix.shape[0]):
        if first_row < second_row:
            first_acum = second_acum = 0
            for col in range(matrix.shape[1]):
                first_acum += matrix[first_row][col]
                second_acum += matrix[second_row][col]
            third_isocline += min(first_acum, second_acum)

    # Calculate the sum of the number of interactions of columns.
for first_col in range(matrix.shape[1]):
    for second_col in range(matrix.shape[1]):
        if first_col < second_col:
            first_acum = second_acum = 0
            for row in range(matrix.shape[0]):
                first_acum += matrix[row][first_col]
                second_acum += matrix[row][second_col]
            fourth_isocline += min(first_acum, second_acum)

    # Calculate and return the nestedness value of the matrix.
return (first_isocline + second_isocline) / (third_isocline + fourth_isocline)
```

6.4 Annex 4: Implementació optimitzada de l'algorisme per calcular el valor de *nestedness* d'una matriu binària en Python

```
def nestedness_optimized(matrix):
    sum_rows = []
    sum_cols = []

    # Calculate and save the number of interactions of every row.
    for row in range(matrix.shape[0]):
        sum_rows.append(sum(matrix[row, :]))

    # Calculate and save the number of interactions of every column.
    for col in range(matrix.shape[1]):
        sum_cols.append(sum(matrix[:, col]))

    first_isocline = second_isocline = third_isocline = fourth_isocline = 0

    # Calculate the sum of the number of shared interactions between rows
    # and the sum of the minimum of pairs of interactions of rows.
    for first_row in range(matrix.shape[0] - 1):
        for second_row in range(first_row + 1, matrix.shape[0]):
            for col in range(matrix.shape[1]):
                if matrix[first_row, col] == 1 and matrix[second_row, col] == 1:
                    first_isocline += 1
            third_isocline += min(sum_rows[first_row], sum_rows[second_row])

    # Calculate the sum of the number of shared interactions between columns
    # and the sum of the minimum of pairs of the number of interactions of columns.
    for first_col in range(matrix.shape[1] - 1):
        for second_col in range(first_col + 1, matrix.shape[1]):
            for row in range(matrix.shape[0]):
                if matrix[row, first_col] == 1 and matrix[row, second_col] == 1:
                    second_isocline += 1
            fourth_isocline += min(sum_cols[first_col], sum_cols[second_col])

    # Calculate and return the nestedness value of the matrix.
    return (first_isocline + second_isocline) / (third_isocline + fourth_isocline)
```

6.5 Annex 5: Script en bash per paral·lelitzar el codi en Python per realitzar un test de *nestedness*

```
#!/bin/bash


# Author: Arey Ferrero Ramos.

# Date: June 9, 2023. Version: 3.

# Description: This script runs a command that executes the section of the project
made in SEES:lab that performs nestedness assessment using the python3 interpreter,
but applying external parallelism.

#      Input:

#             -The type of abundance matrix.

#             -The number of randomized matrices for the assessment.

#      Output:

#             -The value of nestedness of the abundance matrix.

#             -The p-value of the nestedness value of the abundance matrix.


num_matrices=$2

num_procs=$(nproc --all)

num_procs_parallel=$(($num_procs-1))

num_matrices_proc=$(( num_matrices / num_procs_parallel ))


>executable_parallel_tmp.sh

for (( num_proc=1; num_proc<=$num_procs_parallel; num_proc++ )); do

       echo    "python3    nestedness_test.py    ../input_files/count_Genus_all.tsv
../input_files/metadata.csv           ../input_files/sp_code.txt           \"$1\"
\"$num_matrices_proc\"" >> executable_parallel_tmp.sh

done


parallel --eta --bar --jobs $num_procs_parallel :::: executable_parallel_tmp.sh


exit 0
```

6.6 Annex 6: Implementació original de l'algorisme per calcular el valor de *nestedness* d'una matriu binaria en C

```
double calculate_nested_value(int **matrix, int num_rows, int num_cols)

{

    int first_isocline, second_isocline, third_isocline, fourth_isocline;

    int first_row, second_row, row, first_col, second_col, col,
```

```
int first_acum, second_acum;


first_isocline = second_isocline = third_isocline = fourth_isocline = 0;


    /* Calculate the sum of the number of shared interactions between rows. */
for (first_row = 0; first_row < num_rows; first_row++) {
    for (second_row = 0; second_row < num_rows; second_row++) {
        if (first_row < second_row) {
            for (col = 0; col < num_cols; col++) {
                if ((matrix[first_row][col] == 1) && (matrix[second_row][col]
                == 1)) {
                    first_isocline++;
                }
            }
        }
    }
}


    /* Calculate the sum of the number of shared interactions between columns.*/
for (first_col = 0; first_col < num_cols; first_col++) {
    for (second_col = 0; second_col < num_cols; second_col++) {
        if (first_col < second_col) {
            for (row = 0; row < num_rows; row++) {
                if ((matrix[row][first_col] == 1) && (matrix[row][second_col]
                == 1)) {
                    second_isocline++;
                }
            }
        }
    }
}


    /* Calculate the sum of the number of interactions of rows. */
for (first_row = 0; first_row < num_rows; first_row++) {
    for (second_row = 0; second_row < num_rows; second_row++) {
        if (first_row < second_row) {
            first_acum = second_acum = 0;
            for (col = 0; col < num_cols; col++) {
                first_acum += matrix[first_row][col];
```

```
                    second_acum += matrix[second_row][col];
                }
                if (first_acum < second_acum) {
                    third_isocline += first_acum;
                }
                else {
                    third_isocline += second_acum;
                }
            }
        }
    }

        /* Calculate the sum of the number of interactions of columns. */
    for (first_col = 0; first_col < num_cols; first_col++) {
        for (second_col = 0; second_col < num_cols; second_col++) {
            if (first_col < second_col) {
                first_acum = second_acum = 0;
                for (row = 0; row < num_rows; row++) {
                    first_acum += matrix[row][first_col];
                    second_acum += matrix[row][second_col];
                }
                if (first_acum < second_acum) {
                    fourth_isocline += first_acum;
                }
                else {
                    fourth_isocline += second_acum;
                }
            }
        }
    }

        /* Calculate and return the nested value of the matrix. */
      return ((double)(first_isocline + second_isocline) / (double)(third_isocline
      + fourth_isocline));
}
```