; revised .

Desenvolupament i anàlisi d'una aplicació per al càlcul de costos computacionals en operacions de matrius amb arquitectura Model-Vista-Controladora

Dylan Canning Garcia[1], Antonio Marí González[2], Juan Marí González[3], Antoni Jaume Lemesev [4]

[1]Estudiant Enginyeria Informàtica UIB, 49608104W
[2]Estudiant Enginyeria Informàtica UIB, 46390336A
[3]Estudiant Enginyeria Informàtica UIB, 46390338M
[4]Estudiant Enginyeria Informàtica UIB, 49608104W

Alumne d'entrega: Dylan Canning (email: dcg750@id.uib.eu).

ABSTRACT

Aquest document descriu el disseny i la implementació del programa d'anàlisi i visualització del cost asimptòtic desenvolupat pel nostre equip per la primera pràctica del curs. Concretament, es detallen les tècniques emprades com són la programació concurrent i l'arquitectura Model-Vista-Controladora de l'estil de sistema distribuït, tot això seguit d'una explicació detallada dels algorismes implementats, Strassen i Suma Concurrent, juntament amb els seus respectius anàlisis de cost computacional i optimitzacions afegides per part de l'equip.

Finalitzarem el document amb un anàlisi detallat dels resultats obtinguts, juntament amb les gràfiques adients per a la comparació dels algorismes clàssics de suma i multiplicació enfront dels implementats per nosaltres, acabant amb una reflexió de les conclusions obtingudes d'aquesta pràctica, com també possibles millores davant del nostre disseny i implementació de la pràctica amb l'objectiu de millorar gradualment els nostres projectes.

I. Introducció

AQUESTA pràctica s'ha desenvolupat mitjançant una metodologia estructurada basada en l'arquitectura Model-Vista-Controlador (MVC), que ens ha permès treballar en paquets de feina disjunts. Per primera vegada hem implementat aquesta aproximació de desenvolupament, proporcionant-nos una agilitat i autonomia excepcionals, ja que cada component ha pogut evolucionar independentment dels altres.

La distribució de tasques entre el nostre equip de quatre membres s'ha realitzat seguint els components propis del patró MVC: un membre s'ha encarregat de la implementació de la interfície gràfica (Vista), dos membres han desenvolupat la part de dades i algorismes (Model), i un altre ha implementat la part controladora que coordina les interaccions entre els components anteriors. Aquesta divisió de responsabilitats ha estat definida meticulosament abans d'iniciar el desenvolupament, especificant amb claredat les interfícies d'interacció entre els diferents mòduls i els moments precisos en què aquestes s'havien d'utilitzar. Per facilitar la col·laboració i el seguiment del projecte, hem establert un entorn de treball comú basat en un repositori GitHub per al control de versions, permetent documentar i compartir els avenços de manera sistemàtica. Addicionalment, hem estandarditzat l'ús de l'IDE IntelliJ per a tot l'equip, assegurant així la compatibilitat dels formats de projecte i millorant l'eficiència del desenvolupament.

Un cop establertes aquestes bases metodològiques, hem procedit a definir els aspectes tècnics específics de cada secció (Algorismes, Estructures de Dades, Processos, etc.) necessaris per a la implementació. L'objectiu central del nostre disseny ha estat aplicar i maximitzar els coneixements adquirits a l'assignatura, especialment en relació amb les estratègies de Dividir i vèncer i Programació dinàmica,

que constitueixen els fonaments teòrics dels algorismes implementats per a l'anàlisi de costos computacionals en operacions matricials.

II. Metodologia de feina

Per aquest projecte hem seguint una metodologia estructurada i col·laborativa, que ha permès integrar diferents eines i tècniques per aconseguir un producte final coherent sense haver de fer intensius presencials amb tots els companys. En aquest apartat detallarem els aspectes relacionats amb l'entorn de programació i el sistema de control de versions que hem emprat al llarg del desenvolupament.

A. Entorn de Programació

El desenvolupament de l'aplicació l'hem dut a terme utilitzant l'IDE *IntelliJ IDEA*, que ha proporcionat un entorn robust i modern per a la programació en Java, bastant més amigable que *NetBeans*. S'ha treballat amb la versió *Java 11*, aprofitant les seves millores i estabilitat per garantir una execució eficient dels càlculs i processos implementats.

Per a la creació de la interfície gràfica s'han utilitzat les llibreries **Swing** i **JFreeChart**. Swing ha estat fonamental per al disseny i la gestió dels components interactius de la GUI, mentre que JFreeChart ha permès generar gràfics dinàmics i de qualitat per a la visualització dels resultats dels càlculs de costos computacionals asintòtics.

B. Control de Versions

Al llarg del projecte hem adoptat un rigorós sistema de control de versions basat en **Git**, amb un repositori centralitzat allotjat a **GitHub**. Aquesta estratègia ha facilitat:

- Mantenir un historial detallat de totes les modificacions realitzades al codi font.
- Coordinar el treball col·laboratiu entre els membres de l'equip, permetent la integració de diferents funcionalitats de manera ordenada.
- Assegurar una revisió contínua del codi i una resolució ràpida dels conflictes que sorgeixin durant el desenvolupament.

La utilització de GitHub ha estat clau per a la gestió de les versions, garantint que el projecte romangui coherent i actualitzat en tot moment, i afavorint la millora contínua i la transparència en el procés de desenvolupament. A més de l'esmentat, la IDE emprada ens ha permès dur a terme aquest control de versions de manera senzilla pels integrants que no han emprat abans la feina, ja que posseeix eines de gestió de versions sense haver d'utilitzar la terminal.

III. Fonaments Teòrics

En aquesta secció tractarem els diversos elements teòrics que hem emprat per a la nostra pràctica.

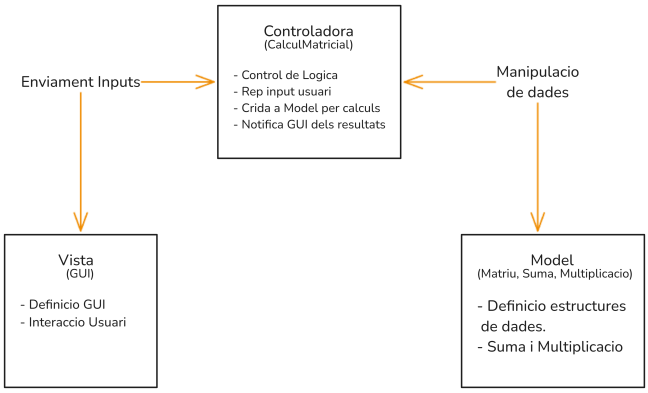

Figura 1. **Model Vista Controladora**

A. Model Vista Controlador

El Model-Vista-Controlador (MVC) és un patró de disseny de software emprat normalment per interfícies d'usuari que té com a objectiu separar la lògica del programa en tres elements:

- Model: La representació interna de la informació.
- Vista: La interfície gràfica que fa de portal entre l'usuari i la lògica interna del programa, es a dir, mostra informació i recull informació de l'usuari.
- Controladora: Fa de connexió entre el Model i la Vista.

En el cas del nostre projecte, hem definit els tres nuclis tal i com diu la definició, la part de la interfície gràfica és l'encarregada de dur a terme tota la interacció amb l'usuari, mostra les gràfiques de complexitat temporal, les constants multiplicatives i permet a l'usuari configurar les simulacions ajustant paràmetres com són les dimensions de les matrius, les operacions a realitzar, els "steps"entre càlculs i la situació dels números interns de les matrius, que explicarem més endavant. La controladora fa de funció de servidor, processa les peticions per part de la Vista/Usuari i acciona els "trigers"apropiats per dur a terme les accions especificades per l'usuari. Finalment, el Model engloba tot allò relacionat a estructures de dades i els mètodes de manipulació d'informació, en el cas d'aquesta pràctica són la classe Matriu que fa de plantilla pels objectes Matriu, Multiplicació i Suma que són les classes que encapsulen tots els procediments i funcions per dur a terme els càlculs/modificacions a les matrius.

Anam a especificar amb més detall cadascuna d'aquestes parts:

1) Interacció Vista $\Longleftrightarrow$ Controladora

En aquesta figura podem observar la interacció seqüencial entre la Vista i la Controladora. En el cas de la controladora, aquesta queda esperant el missatge de la GUI per a començar un càlcul, una vegada rebuda la notificació la controladora elimina les dades antigues inicialitza els nous fils d'execució per als càlculs i canvia el seu estat a "running"per indicar que

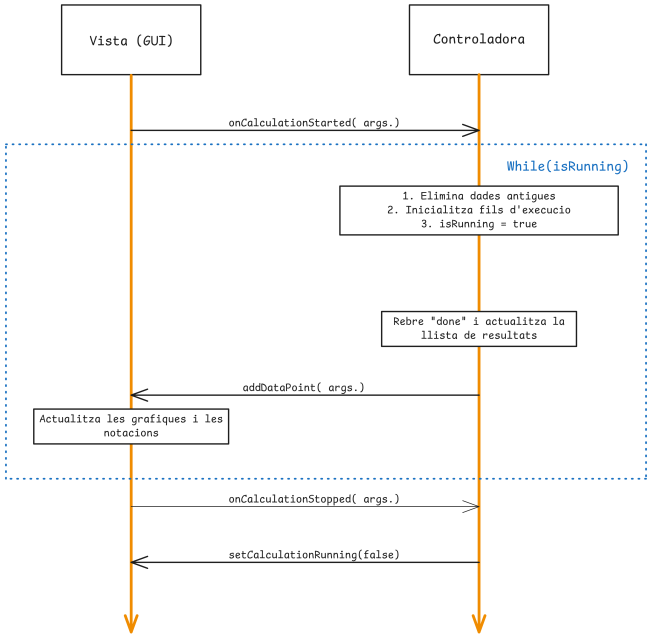

Figura 2. **Interacció Vista amb Controladora**

s'està realitzant una operació. Una vegada la controladora ha rebut el missatge de "done"per part del model, envia el punt calculat a la interfície gràfica i la vista procedeix a actualitzar les gràfriques.

2) Interacció Controladora $\Longleftrightarrow$ Model

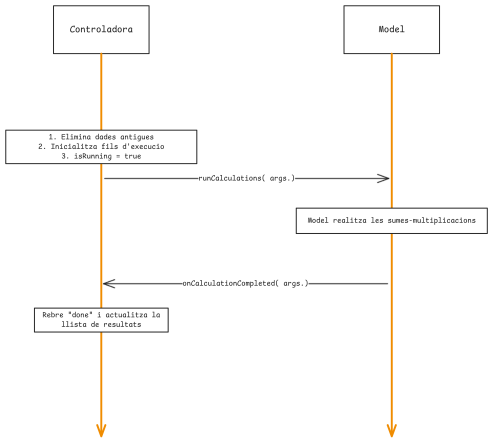

Figura 3. **Interacció Controladora amb Model**

En aquesta darrera figura podem observar la interacció entre la controladora i el model, igual que abans es comuniquen fent ús de mètodes que es comporten com a notificacions que creen esdeveniments/reaccions. Aquesta interacció és molt més simple, indicam al Model quin tipus de modificacions s'han de fer damunt les dades (Suma o Multiplicació) i el model les realitza i avisa a la controladora en haver-hi acabat. Això crea una independència total entre model i vista, que és un dels fonaments principals de la arquitectura MVC.

B. Esdeveniments

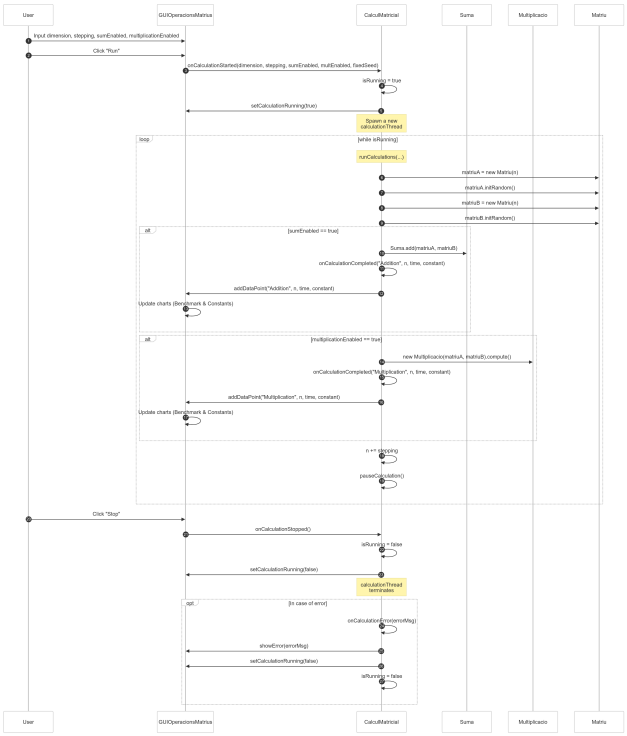

Figura 4. **Esdeveniments del programa**

Aquest diagrama, tot i estar molt estret representa el patró d'esdeveniments del nostre programa, conté els principals missatges de comunicació entre els distints components del codi, fent èmfasi en les funcions pròpies de cadascuna de les seccions, a més de representar els cicles interns que es produeixen quan s'executa les operacions de càlcul.

El patró d'esdeveniments és bastant senzill, però ens resulta bastant interessant el canvi de perspectiva que ens ha aportat fer-ho així, sobretot redueix molt l'esforç de crear interfícies gràfiques, és semblant a la implementació d'un servidor en canals, tens dos processos que s'han de comunicar fent ús del "servidor". Ara bé, aquest tipus d'estructura de programació posa molt de pes al seguiment d'un bon disseny a tots els nivells del programa, ja que s'ha de tenir en compte les comunicacions interblock que es donen dins del programa.

Sense sobre-explicar el diagrama farem una breu explicació del que ocorre al nostre codi, començant per els elements diferenciats del flow-chart:

- **Usuari**: L'usuari representa la persona que interactua de manera directa amb la interfície gràfica, realitzant accions com son la introducció de la dimensió, les passes i les operacions a inicialitzar; i clarament l'esdeveniment de "run", que indica a la controladora de començar les calculacions.
- **GUIOperacionsMatrius**: Aquest element representa la interfície gràfica la qual es emprada per l'usuari per realitzar totes les seves accions, com es veu al diagrama

es principal un receptor de missatges per part de la controladora, però també fa una comunicació cap a ella com son *onCalculationStarted* que representa l'inici de les calculacions per a que la controladora comenci les seves calculacions, o també *onCalculationsStopped()* per detenir les operacions.
- **CalculMatricial**: Aquest altre objecte representa la controladora, que com es pot observar es la que més interaccions comparteix amb la vista i amb les dades, a la gràfica apareix suficientment explicat les funcions i interaccions, per tant no farem un anàlisi d'aquesta.
- **Suma, Multiplicacio i Matriu**: Aquestes son les tres classes principals, com podem observar l'unica utilitat que tenen es la instanciació per part de la controladora per crear objectes de aquestes classes i en el cas de la suma al tenir els metodes Static simplement cridar a la operació add().

1) Per què implementació basada en esdeveniments?

L'aplicació ha estat dissenyada emprant un patró d'esdeveniments que optimitza la responsivitat de la interfície i millora la modularitat del codi. Aquest enfocament permet separar clarament les responsabilitats entre la Vista, el Model i el Controlador, de manera que cada component pugui actuar de forma independent i reaccionar a esdeveniments específics sense bloquejar el flux principal de l'aplicació.

Per exemple, es defineix una interfície `Notificar` que actua com a contracte per a la comunicació d'esdeveniments importants, com ara l'inici, la finalització o la detecció d'errors en els càlculs. Això permet que el Controlador notifiqui a la Vista en temps real sobre l'estat dels processos, millorant la resposta a les interaccions de l'usuari.

```
public interface Notificar {
    void onCalculationStarted();
    void onCalculationCompleted(Result result);
    void onCalculationError(String errorMessage);
}
```

A continuació, es mostra un fragment de codi del Controlador que utilitza aquesta interfície per gestionar els esdeveniments:

```
public void startCalculation() {
    notifier.onCalculationStarted();
    new Thread(() -> {
        try {
            Result result = model.computeResult();
            notifier.onCalculationCompleted(result)
                ;
        } catch (Exception e) {
            notifier.onCalculationError(e.
                getMessage());
        }
    }).start();
}
```

Amb aquesta implementació basada en esdeveniments, s'aconsegueixen dos avantatges clau:

- **Responsivitat:** El flux de càlcul es realitza de manera asíncrona, evitant bloquejar la interfície gràfica i permetent una experiència d'usuari fluida.
- **Modularitat:** Cada component (Vista, Model i Controlador) està desacoblat, facilitant la seva modificació, manteniment i expansió sense afectar el sistema global.

IV. Gestió del multi-threading i optimització de rendiment

Per aconseguir un rendiment òptim en els càlculs matricials sense afectar la responsivitat de la interfície gràfica, s'ha aplicat una estratègia multimodal de multi-threading que integra diferents tècniques segons la naturalesa de l'operació.

A. Paral·lelització de la Suma amb ExecutorService i CountDownLatch

En l'operació de suma de matrius, s'ha implementat un `ExecutorService` amb un nombre fix de fils, determinat a partir dels nuclis disponibles. La suma es divideix en blocs, amb la mida de cada bloc calculada per maximitzar l'ús de la cache (tenint en compte la constant `CACHE_LINE_SIZE`), i cada bloc es processa de forma concurrent. Per garantir que totes les tasques es compleixin abans de continuar, s'utilitza un `CountDownLatch` que es decrementa al finalitzar cada tasca. Aquesta solució permet:

- Al·liniar les dades amb la cache per obtenir un accés més ràpid.
- Sincronitzar la finalització de tots els blocs abans d'obtenir el resultat final.

B. Paral·lelització de la Multiplicació amb ForkJoinPool

Per a la multiplicació de matrius, s'ha adoptat l'algorisme de Strassen implementat mitjançant el model `ForkJoinPool`. Aquesta tècnica recursiva divideix la matriu en submatrius i crea tasques paral·leles per calcular els productes parcials. Algunes decisions clau en aquesta implementació són:

- Utilitzar un llindar (`UMBRAL_STRASSEN`) per determinar quan canviar a la multiplicació clàssica, evitant així la sobrecàrrega de crear fils per a submatrius petites.
- Controlar la profunditat de recursió (`MAX_PROFUNDITAT`) per mantenir un equilibri entre paral·lelisme i sobrecàrrega.
- Emprar els mètodes `fork()` i `join()` per gestionar la creació i combinació de tasques de manera eficient.

C. Gestió Global dels Fils al Controlador

La classe controladora, que implementa la interfície `Notificar`, crea un fil dedicat (`calculationThread`) per executar els càlculs de manera asíncrona. Això permet que la interfície gràfica romangui responsiva durant l'execució de processos intensius. A més, el controlador s'encarrega d'alliberar els recursos associats al `ExecutorService` i

al `ForkJoinPool` quan es deté el càlcul, assegurant una finalització segura i sense fuites de recursos.

D. Reptes i Solucions

Durant la implementació hem trobat diversos reptes:

- **Sincronització de Tasques:** Per a la suma, el repte[1] era assegurar que tots els blocs processats de forma[2] paral·lela finalitzessin abans de combinar els resultats.[3] La utilització de `CountDownLatch` ha estat clau per resoldre aquest problema.
- **Granularitat de les Tasques:** En la multiplicació, s'ha de trobar l'equilibri entre dividir massa la tasca (el que pot provocar una sobrecàrrega en la gestió de fils) i aprofitar el paral·lelisme. L'establiment d'un llindar per al canvi a l'algorisme clàssic ha permès optimitzar aquesta transició.
- **Gestió dels Recursos:** La creació d'un fil dedicat per als càlculs i la correcta finalització dels recursos (tant del `ExecutorService` com del `ForkJoinPool`) han estat essencials per mantenir la stabilitat de l'aplicació i evitar fuites de memòria o bloquejos.

Aquest enfocament combinat en la gestió del multithreading ha permès accelerar els càlculs matricials tot mantenint una interfície d'usuari fluida i responsiva, assegurant al mateix temps una sincronització acurada i un ús eficient dels recursos disponibles.

V. Càlcul de complexitat asimptòtica

La suma i la multiplicació de matrius són operacions fonamentals en àmbits com l'Àlgebra Lineal, el processament d'imatges i molts algorismes científics. Aquests càlculs involucren un gran nombre d'operacions elementals, especialment per a matrius de mida considerable. Per tant, és crucial entendre el comportament d'aquests algorismes quan la quantitat de dades creix, és a dir, fer una anàlisi de la seva complexitat temporal asimptòtica.

L'anàlisi de cost asimptòtic permet comparar algorismes independentment de la implementació específica o de la plataforma, centrant-se en com l'ús de recursos (temps de CPU, operacions en memòria) escala amb la mida de la entrada. En el cas de la suma i la multiplicació de matrius, aquesta anàlisi és especialment important per a triar la millor estratègia en tractar conjunts de dades de gran dimensió o en optimitzar el codi per a un alt rendiment.

A. Transformació a Pseudocodi

Per tal d'analitzar clarament aquests algorismes, primer presentarem una descripció en pseudocodi de cadascuna de les operacions. Aquesta representació permet abstraure'n detalls d'implementació i centrar-nos en les operacions fonamentals que es duen a terme en cada cas.

1) Suma de matrius

La suma de dues matrius A i B de mida $n \times n$ es realitza sumant element a element les posicions corresponents. A continuació es mostra el pseudocodi de l'algorisme de suma de matrius, que recorre les dues matrius amb dos bucles imbricats:

```
per i = 1 fins a n:
    per j = 1 fins a n:
        C[i,j] = A[i,j] + B[i,j]
```

En finalitzar, la matriu C conté la suma d'A i B, és a dir, $C[i,j] = A[i,j] + B[i,j]$ per a tots els $1 \leq i,j \leq n$.

2) Multiplicació de matrius

La multiplicació de matrius es pot implementar de diverses maneres. A continuació es presenten dues versions en pseudocodi: l'algorisme clàssic (també anomenat *naïf*) i l'algorisme de Strassen, que utilitza una estratègia de *divideix i vençe* per reduir el nombre d'operacions multiplicatives necessàries.

Algorisme clàssic multiplicació

L'algorisme clàssic de multiplicació de matrius de mida $n \times n$ utilitza tres bucles niats per calcular cada element de la matriu resultant C com el producte escalar d'una fila de A per una columna de B:

```
per i = 1 fins a n:
    per j = 1 fins a n:
        C[i,j] = 0
        per k = 1 fins a n:
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

Després d'executar aquest pseudocodi, C contindrà el producte matricial $C = A \times B$, on cada element $C[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j]$.

Algorisme de Strassen per a la multiplicació

L'algorisme de Strassen 1969 aplica la tècnica de *divideix i vençe* per multiplicar matrius de manera més eficient que el mètode clàssic. En essència, divideix cada matriu en quatre submatrius de mida $n/2 \times n/2$ i calcula combinacions lineals que permeten reduir el nombre de multiplicacions de 8 (que seria el cas d'una divisió recursiva directa) a 7 multiplicacions recursives, a costa d'un increment en el nombre d'addicions. La idea clau és que, en combinar les submatrius adequadament, s'obtenen els productes parcials amb només 7 multiplicacions de submatrius, intercalant operacions de suma i resta de submatrius.

A continuació, es descriu en pseudocodi l'algorisme de Strassen per multiplicar dues matrius A i B de dimensió $n \times n$ (suposant n potència de 2 per simplicitat):

```
funcio StrassenMultiplicar(A, B, n):
    si n = 1:
        retornar A[1,1] * B[1,1]
    altrament:
        m = n/2
        Dividir A en submatrius A11, A12, A21, A22
            (mida m)
        Dividir B en submatrius B11, B12, B21, B22
            (mida m)
        P1 = StrassenMultiplicar(A11 + A22, B11 +
            B22)
        P2 = StrassenMultiplicar(A21 + A22, B11)
        P3 = StrassenMultiplicar(A11, B12 - B22)
        P4 = StrassenMultiplicar(A22, B21 - B11)
        P5 = StrassenMultiplicar(A11 + A12, B22)
        P6 = StrassenMultiplicar(A21 - A11, B11 +
            B12)
        P7 = StrassenMultiplicar(A12 - A22, B21 +
            B22)
        C11 = P1 + P4 - P5 + P7
        C12 = P3 + P5
        C21 = P2 + P4
        C22 = P1 + P3 - P2 + P6
        Combinar C11, C12, C21 i C22 en una matriu
            C
        retornar C
```

En aquest pseudocodi, $A11, \ldots, A22$ i $B11, \ldots, B22$ representen les quatre submatrius de A i B obtingudes en dividir-les en blocs quadrats iguals. Les operacions $A11 + A22$, $A21 - A11$, etc., són sumes o restes de submatrius. Els càlculs $P1, \ldots, P7$ corresponen a multiplicacions recursives de submatrius (cada una de dimensió $m \times m$). Finalment, les expressions per $C11, C12, C21, C22$ combinen aquests resultats parcials per obtenir els quadrants de la matriu final C. Aquest esquema redueix el nombre de multiplicacions de submatrius (7 en lloc de 8), aconseguint una millora en cost teòric a canvi de més operacions de suma i resta de matrius.

B. Anàlisi del Cost Asimptòtic

A continuació s'analitza el cost asimptòtic (complexitat temporal) de cadascun dels algorismes presentats, mesurat en termes del nombre d'operacions elementals requerides en funció de la mida n de les matrius. També es discuteix com afecten el cost algunes tècniques de paral·lelisme i optimització.

1) Suma de matrius

La suma de dues matrius $n \times n$ requereix iterar per totes les n^2 posicions i realitzar una operació de suma per a cadascuna. Per tant, el cost total (en nombre d'operacions aritmètiques bàsiques) creix proporcionalment a n^2. En notació de complexitat, la suma de matrius és un procés de cost $\Theta(n^2)$. Això significa que, asimptòticament, si la mida de la matriu es duplica, el nombre d'operacions (i aproximadament el temps de càlcul) s'incrementa per un factor de quatre.

2) Multiplicació de matrius clàssica

L'algorisme clàssic de multiplicació de matrius realitza, per a cada una de les n^2 posicions de la matriu resultant, un producte escalar de longitud n. En concret, per a cada parella d'índexs (i, j), es calcula $C[i, j] = \sum_{k=1}^{n} A[i, k] \cdot B[k, j]$. Això implica n multiplicacions i $n-1$ sumes per al còmput de cada element $C[i, j]$. En total, el nombre d'operacions elementals (sumes i multiplicacions) creix en l'ordre de n^3. En termes de complexitat, el mètode clàssic té un cost $\Theta(n^3)$. Per a valors grans de n, el temps d'execució creix de forma cúbica; per exemple, multiplicar una matriu de dimensió $2n$ requerirà en principi unes vuit vegades més operacions que multiplicar una de dimensió n (ja que $(2n)^3 = 8n^3$).

3) Algorisme de Strassen

L'algorisme de Strassen millora la complexitat reduint el nombre de multiplicacions recursives. Si denotem per $T(n)$ el nombre d'operacions (o el temps) per multiplicar dues matrius de mida $n \times n$ amb Strassen, es pot aproximar mitjançant la recurrència:

$$T(n) = 7\,T(n/2) + c\,n^2,$$

on la part $c\,n^2$ prové de les operacions de suma i resta de submatrius (per alguna constant c específica). Aplicant el Teorema Master o resolent aquesta recurrència recursivament, s'observa que el terme que domina és $n^{\log_2 7}$. Concretament, $\log_2 7 \approx 2.8074$, de manera que la complexitat asimptòtica de Strassen es pot expressar com $O(n^{2.8074})$. Aquest exponent és inferior a 3, la qual cosa indica que per a n prou grans, l'algorisme de Strassen serà més eficient (en termes de nombre d'operacions) que l'algorisme clàssic, asimptòticament parlant.

Cal notar, però, que la millora asimptòtica comporta un cost constant addicional considerable: Strassen realitza més operacions de suma de matrius que l'enfocament clàssic, i requereix més memòria per emmagatzemar submatrius temporals. Això significa que, en mides moderades, l'algorisme clàssic altament optimitzat pot ser més ràpid en la pràctica. Amb tot, des d'un punt de vista teòric, per a n suficientment gran el terme $n^{2.8074}$ creix més lentament que n^3, marcant un avantatge eventual per a l'algorisme de Strassen.

4) Paral·lelisme i optimitzacions

Més enllà de l'anàlisi en mode seqüencial, és important considerar com el paral·lelisme i altres optimitzacions afecten el cost efectiu d'aquests algorismes.

La **suma de matrius** és altament paral·lelitzable, ja que la suma de cada posició és independent de les altres. En una implementació paral·lela ideal amb p processadors, el temps d'execució podria reduir-se aproximadament a

$O(n^2/p)$, distribuint els n^2 càlculs entre els p nuclis. En la pràctica, factors com la comunicació i la coordinació entre fils introdueixen un cert sobrecost, però la velocitat de càlcul pot escalar de forma quasi lineal amb el nombre de nuclis. Per exemple, utilitzant un conjunt de fils (pool de *threads*) es pot repartir el càlcul de la suma de manera que cada fil sumi un bloc contigu de la matriu. Una tècnica emprada és agrupar els elements en blocs alineats amb les línies de memòria cau per maximitzar la localitat (accedint seqüencialment a la memòria) i fer servir *loop unrolling* (desenrotllament del bucle) per reduir l'overhead de control de bucle. Aquestes optimitzacions no canvien la complexitat asimptòtica ($\Theta(n^2)$ en la suma de matrius), però milloren notablement els factors constants de temps, aconseguint un ús més eficient de l'arquitectura de maquinari.

De forma similar, la **multiplicació de matrius** es pot beneficiar del paral·lelisme. En l'algorisme clàssic, es poden assignar diferents parts del càlcul (per exemple, blocs de files o submatrius) a diferents fils o nuclis. Això permet, en teoria, reduir el temps a l'entorn de $O(n^3/p)$ utilitzant p processadors, tot i que l'acceleració real sovint es veu limitada per la necessitat d'accedir a la mateixa memòria (possible congestió a la memòria cau) i pel cost de combinar els resultats parcials. Una tècnica clau d'optimització per a millorar el rendiment de la multiplicació (encara en mode seqüencial) és la multiplicació per blocs (*tiling*), on es divideixen les matrius en sub-blocs més petits (*tiles*) que es processen completament abans de passar al següent. Aquest *blocking* millora la localitat de referència a la memòria cau, reduint errors de cau (*cache misses*) i aprofitant millor la jerarquia de memòria. El nombre total d'operacions computacionals segueix sent $O(n^3)$, però l'execució és més eficient en termes de temps real, especialment en arquitectures modernes on l'accés a memòria és un factor limitant.

L'algorisme de **Strassen** també es pot paral·lelitzar aprofitant la seva naturalesa de divideix i venç. Com que calcula set multiplicacions de submatrius de forma independent a cada nivell recursiu, es disposa de treball que es pot fer en paral·lel. Per exemple, es podrien llançar fins a 7 fils per calcular $P1, \ldots, P7$ simultàniament en la fase descrita al pseudocodi. En la implementació amb `ForkJoinPool` de Java, s'utilitza un esquema de tasques recursives: cada crida recursiva pot fer *fork* de subtasques per calcular en paral·lel part dels P_i i després fer *join* per combinar els resultats. Amb prou nuclis disponibles, el temps d'execució teòric de Strassen es podria reduir aproximadament a $O(n^{2.8074}/p)$ (fins al límit de saturar 7 tasques paral·leles per nivell recursiu). No obstant això, a la pràctica cal limitar la granularitat de la paral·lelització per tal que el cost de gestionar fils no superi el benefici. Per això, sovint s'estableix un llindar de mida (per exemple, matrius de 64x64) a partir del qual l'algorisme deixa d'aplicar Strassen i reverteix a la multiplicació clàssica seqüencial, ja que per a submatrius petites la sobrecàrrega de l'algorisme de Strassen no compensa. També es pot limitar la profunditat recursiva paral·lela (per exemple, un màxim de 3 nivells de subdivisió en paral·lel) per controlar el nombre total de tasques en execució. Aquestes mesures busquen un equilibri entre l'explotació del paral·lelisme i el cost de sincronització, permetent obtenir acceleracions importants en entorns multinucli sense perdre l'eficiència.

C. Demostracions i Justificacions

En aquest apartat es proporcionen proves i justificacions formals del cost de cada operació i algorisme, seguint el comportament asimptòtic analitzat, així com una comparació teòrica entre les diferents implementacions esmentades.

1) Cost de la suma de matrius (demostració)

Formalment, per calcular la suma de dues matrius A i B de dimensió $n \times n$, s'han de realitzar:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} 1 = n \times n = n^2$$

operacions de suma de nombres. Aquest sumatori doble reflecteix que per a cadascun dels n valors de i i cadascun dels n valors de j efectuarem exactament una operació elemental (una suma). D'aquí es dedueix directament que el cost total és n^2. Aquest resultat també es pot demostrar per inducció sobre n: pel cas base $n = 1$ és trivial ($1^2 = 1$ operació); suposem cert per a $n - 1$ (és a dir, sumar dues matrius de $(n-1) \times (n-1)$ requereix $(n-1)^2$ operacions) i considerem la suma de dues matrius de $n \times n$. Afegir la fila i la columna n-èsima comporta sumar $2n - 1$ elements extra (n elements a la nova fila i $n - 1$ a la nova columna, ja que l'element de la cantonada ja es compta a la fila). En total tindríem $(n-1)^2 + 2n - 1 = n^2$ operacions. Per tant, per inducció, la fórmula es compleix per a tot n.

2) Cost de la multiplicació clàssica (demostració)

En l'algorisme clàssic de multiplicació, per a cada posició (i, j) de la matriu resultat C, es calcula:

$$C[i, j] = \sum_{k=1}^{n} A[i, k] \times B[k, j].$$

Això suposa n multiplicacions i $n - 1$ sumes per obtenir un sol element $C[i, j]$. Com hi ha n^2 elements a C, podem comptabilitzar el nombre total d'operacions de la manera següent. El nombre de multiplicacions és:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1_{\text{(mult)}} = n^3,$$

ja que es realitza una multiplicació per a cada tríplet (i, j, k). El nombre de sumes (per acumular els productes per a cada (i, j)) és:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} (n-1) = n^2 \cdot (n-1) = n^3 - n^2.$$

Sumant ambdós tipus d'operacions s'obté un total de $n^3 + (n^3 - n^2) = 2n^3 - n^2$ operacions elementals. En notació $O(\cdot)$, això és $O(n^3)$, ja que per a $n \to \infty$ el terme dominant és el n^3. També podem verificar el caràcter cúbic comparant el cost per una matriu de mida n i una de mida $2n$:

$$\frac{T(2n)}{T(n)} \approx \frac{2(2n)^3 - (2n)^2}{2(n^3) - n^2} \approx \frac{16n^3}{2n^3} = 8,$$

és a dir, doblar la dimensió de la matriu multiplica per vuit el nombre d'operacions (ignorem termes de menor grau), confirmant el creixement cúbic esperat.

3) Cost de la multiplicació amb Strassen (demostració)

Per a l'algorisme de Strassen, analitzem formalment la seva complexitat resolent la recurrència que el defineix. Com hem vist, el temps de càlcul $T(n)$ satisfà aproximadament:

$$T(n) = 7\,T(n/2) + a\,n^2,$$

per alguna constant a (que correspon al cost de les sumes i restes de submatrius a cada nivell). Per simplicitat en la demostració, suposem que $n = 2^m$ és una potència de 2, de manera que la recursió pot aplicar-se repetidament fins al cas base $T(1)$. Desenvolupem la recurrència pas a pas:

$$\begin{aligned}
T(n) &= 7\,T(n/2) + a\,n^2, \\
&= 7\Big(7\,T(n/4) + a\,(n/2)^2\Big) + a\,n^2, \\
&= 7^2\,T(n/4) + 7a\,(n/2)^2 + a\,n^2, \\
&= 7^2\Big(7\,T(n/8) + a\,(n/4)^2\Big) + 7a\,(n/2)^2 + a\,n^2, \\
&= 7^3\,T(n/8) + 7^2 a\,(n/4)^2 + 7a\,(n/2)^2 + a\,n^2.
\end{aligned}$$

Si continuem aquest procés recursiu m nivells (fins que la mida de les submatrius sigui 1), obtindrem:

$$T(n) = 7^m\,T(1) + a\sum_{i=0}^{m-1} 7^i \left(\frac{n^2}{4^i}\right),$$

on hem utilitzat que $n = 2^m$ i, en conseqüència, després de m nivells recursius tenim subproblemes de mida 1 (cas base). Notem que $7^m = 7^{\log_2 n} = n^{\log_2 7}$. El segon terme del costat dret representa la suma de les operacions de suma/resta realitzades a cada nivell de la recursió. Simplifiquem aquesta sumatòria per acotar-la:

$$\begin{aligned}
\sum_{i=0}^{m-1} 7^i \frac{n^2}{4^i} &= n^2 \sum_{i=0}^{m-1} \left(\frac{7}{4}\right)^i \\
&= n^2 \cdot \frac{\left(\frac{7}{4}\right)^m - 1}{\frac{7}{4} - 1} \\
&< n^2 \cdot \frac{\left(\frac{7}{4}\right)^m}{\frac{3}{4}} \\
&= \frac{4}{3}\,n^2 \left(\frac{7}{4}\right)^m \\
&= \frac{4}{3}\,n^2 \left(\frac{2^{\log_2 7}}{2^{\log_2 4}}\right)^m \\
&= \frac{4}{3}\,n^2 \left(2^{\log_2 7 - \log_2 4}\right)^m \\
&= \frac{4}{3}\,n^2 \cdot 2^{\,m(\log_2 7 - 2)} \\
&= \frac{4}{3}\,n^2 \cdot 2^{\log_2 n\,(\log_2 7 - 2)} \\
&= \frac{4}{3}\,n^2 \cdot n^{\log_2 7 - 2} \\
&= \frac{4}{3}\,n^{\log_2 7}.
\end{aligned}$$

Hem acotat la sumatòria pel valor d'una progressió geomètrica de raó $7/4$ (que és major que 1). Com que $\frac{4}{3}n^{\log_2 7}$ és de l'ordre $O(n^{\log_2 7})$, podem concloure que:

$$T(n) = 7^m\,T(1) + O(n^{\log_2 7}).$$

Tenint en compte que el terme $7^m T(1) = T(1)\,n^{\log_2 7}$ també creix com $n^{\log_2 7}$, la part $O(n^{\log_2 7})$ domina el comportament per a grans n. Per tant, en termes asimptòtics,

$$T(n) = \Theta(n^{\log_2 7}),$$

la qual cosa ratifica que la complexitat de l'algorisme de Strassen pertany a l'ordre de $n^{2.8074}$ (aproximadament).

4) Comparació teòrica entre implementacions

Des del punt de vista teòric, podem comparar les complexitats asimptòtiques i el nombre d'operacions dels diferents algorismes i implementacions analitzats:

- **Suma de matrius**: requereix exactament n^2 operacions (totes sumes). Complexitat $\Theta(n^2)$. Es tracta d'un procediment òptim en el sentit que és lineal en el nombre d'elements de les matrius (no es pot fer millor que llegir i sumar tots els elements).
- **Multiplicació clàssica**: realitza aproximadament $2n^3$ operacions (al voltant de n^3 multiplicacions i n^3 sumes). Complexitat $\Theta(n^3)$. Cada increment modest en la dimensió de la matriu provoca un creixement significatiu en el temps de càlcul (creixement cúbic).
- **Multiplicació amb Strassen**: realitza $O(n^{2.8074})$ operacions en el model asimptòtic, la qual cosa és significativament millor per a n molt grans. En comptatge exacte, per cada nivell recursiu es fan 7 multiplicacions

de submatrius i diverses sumes (en concret 18 sumes de submatrius per recombinar els resultats, segons el pseudocodi), però asimptòticament el cost de les multiplicacions domina i marca l'exponent global.

- **Implementació optimitzada (tiling)**: manté el mateix ordre $\Theta(n^3)$ però amb factors constants més petits gràcies a una millor explotació de la jerarquia de memòria. Teòricament no canvia la classe de complexitat, però en termes de temps d'execució pot suposar una acceleració notable en comparació amb l'algorisme clàssic sense optimitzar, especialment per a dimensions grans on l'ús eficient de la memòria cau és crític.
- **Implementació paral·lela**: amb p processadors, el nombre total d'operacions es reparteix entre ells, però la complexitat vista com a funció de n continua essent $\Theta(n^2)$ o $\Theta(n^3)$ (segons sigui suma o multiplicació). La millora és en el temps de càlcul efectiu: en el millor dels casos, el temps pot disminuir aproximadament en un factor p. Per exemple, una multiplicació clàssica paral·lela tindria un temps d'execució $T_{\text{par}}(n) \approx \Theta(n^3/p)$, i l'algorisme de Strassen paral·lel tindria $T_{\text{par}}(n) \approx \Theta(n^{2.8074}/p)$. En ambdós casos, l'acceleració teòrica màxima és lineal en p, tot i que en la pràctica pot veure's limitada per les sobrecàrregues de sincronització i comunicació.

En resum, l'avantatge teòric de Strassen es manifesta únicament per a mides molt grans de matriu, ja que les constants i requeriments addicionals fan que per a dimensions moderades l'algorisme clàssic (especialment si està optimitzat amb técniques com *tiling*) pugui ser més ràpid. A més, existeixen altres algorismes de multiplicació de matrius amb complexitats encara millors (per exemple, algorismes basats en tècniques algebraiques que assoleixen exponents propers a 2), però aquests són majoritàriament d'interès teòric, ja que són extremadament complicats d'implementar i només són avantatjosos per a mides descomunalment grans. Pel que fa a la *suma de matrius*, el seu cost $\Theta(n^2)$ és òptim i no hi ha espai per a millorar l'ordre asimptòtic (tots els elements s'han de processar); no obstant, es pot escalar horitzontalment de forma quasi perfecte aprofitant paral·lelisme massiu, degut a la seva independència de dades entre elements.

D. Reflexió del cost

Hem analitzat dos dels algorismes implementats per a l'operació amb matrius (la suma i la multiplicació) i les optimitzacions afegides. La **suma de matrius** presenta un cost polinòmic de segon grau ($\Theta(n^2)$), mentre que la **multiplicació de matrius clàssica** té un cost de tercer grau ($\Theta(n^3)$). L'**algorisme de Strassen** suposa una fita important en reduir l'exponent de la multiplicació fins a ≈ 2.8074, demostrant que és possible multiplicar matrius de forma més eficient que el mètode elemental mitjançant tècniques de *divideix i vençe*. Hem vist que aquesta millora teòrica comporta una major complexitat en la implementació i una penalització en termes de constants, però obre la porta a algorismes encara més ràpids per a dimensions molt grans.

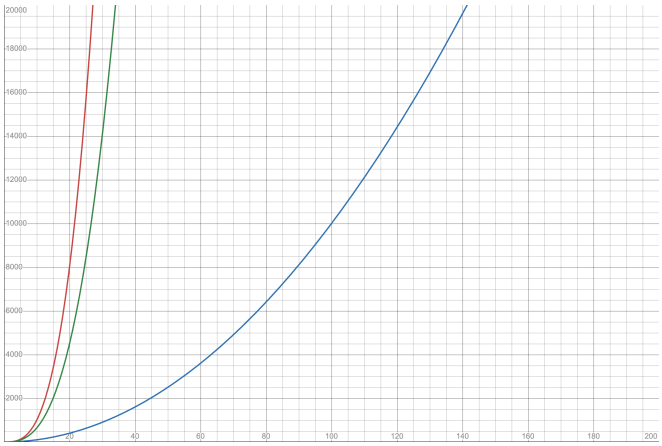

Figura 5. **Gràfica cost computacions. Blau = Suma, Vermell = Multiplicació Clàssica, Verd = Strassen.**

A nivell pràctic, les implementacions paral·leles i les optimitzacions específiques de maquinari (com el *tiling* per a millorar l'ús de la memòria) juguen un paper crucial en el rendiment real. Tot i que no alteren la classe de complexitat asimptòtica fonamental, permeten aprofitar al màxim els recursos físics disponibles i acostar l'execució al màxim teòric de rendiment. En el nostre anàlisi hem constatat que una estratègia adequada de paral·lelisme pot reduir dràsticament el temps d'execució, especialment per a la suma de matrius (que escala de forma gairebé perfecta) i per a la multiplicació quan es divideix el treball entre nuclis de manera equilibrada.

Tot i que la nostra implementació intenta optimitzar la multiplicació al màxim, cal esmentar que existeixen algorismes de multiplicació de matrius fins i tot més eficients en teoria que el de Strassen (per exemple, algorismes basats en l'algoritme de Coppersmith-Winograd i millores modernes que assoleixen exponents propers a 2). Tanmateix, aquests algorismes són de gran complexitat i el seu benefici pràctic és limitat a casos molt especials. Mentrestant, la tendència en l'àmbit de l'alt rendiment és combinar algorismes clàssics i millores com Strassen amb implementacions multinucli i fins i tot multiplataforma (GPU, computació distribuïda) per accelerar el càlcul matricial. En conclusió, comprendre el cost asimptòtic i les oportunitats de paral·lelisme ens permet dissenyar i triar millor els algorismes per a l'operació amb matrius, optimitzant-ne tant l'eficiència teòrica com el rendiment pràctic en diversos entorns de computació.

VI. Conclusions

En aquesta pràctica hem analitzat el cost computacional de les operacions de suma i multiplicació de matrius, tant amb algorismes clàssics com amb versions optimitzades. La suma de matrius presenta un cost polinòmic de segon grau, $\Theta(n^2)$, que ès òptim, ja que cal processar obligatòriament tots els n^2 elements de les matrius. La multiplicació de matrius

amb l'algorisme clàssic té un cost de tercer grau, $\Theta(n^3)$, segons l'anàlisi teórica estàndard. No obstant això, l'algorisme de Strassen va demostrar que ès possible multiplicar matrius de forma més eficient que el mètode elemental: redueix l'exponent de la complexitat fins aproximadament $O(n^{2.8074})$. Aquesta fita, introduïda per Volker Strassen el 1969, confirma que es pot millorar el cost asimptòtic de la multiplicació més enllà de $\Theta(n^3)$. Els nostres resultats reflecteixen aquesta millora teòrica: per a dimensions grans de matriu l'algorisme de Strassen assoleix temps d'execució inferiors al clàssic, confirmant el seu avantatge asimptòtic.

Tanmateix, també hem observat que l'avantatge de Strassen només es manifesta per a mides de matriu molt grans. Per a dimensions moderades, els factors constants i la sobrecàrrega addicional de Strassen (més operacions de suma i combinació de subresultats) fan que l'algorisme clàssic optimitzat pugui ser més ràpid en la pràctica. En particular, tècniques d'optimització com la multiplicació per blocs (*tiling*), que explota millor la jerarquia de memòria cau, milloren dràsticament el rendiment de la multiplicació clàssica sense alterar la seva complexitat base. Nosaltres hem implementat aquesta tècnica i hem constatat una millora notable: l'ús de blocs incrementa la localitat de les dades i redueix els temps d'accés a memòria, aconseguint accelerar l'algorisme clàssic en comparació amb la versió naïf. Aquesta observació concorda amb altres treballs d'alt rendiment, on s'ha demostrat que un bon ús de la memòria cau pot fer que l'algorisme clàssic superi Strassen fins que la mida de les matrius ès prou gran.

Un altre aspecte estudiat han estat les implementacions paral·leles, que han evidenciat millores substancials de rendiment. En la nostra aplicació concurrent, repartir el càlcul entre diversos fils d'execució threads ha permès reduir dràsticament el temps d'execució efectiu. Per a la suma de matrius, que té independència de dades entre elements, hem assolit una *escala gairebé perfecta*: amb p processadors, el temps s'aproxima a $\Theta(n^2/p)$. Aquest resultat era previsible donada la descomposició trivial de la suma en tasques completament independents. Per a la multiplicació de matrius, la paral·lelització també ha millorat el temps de càlcul de forma notable. Distribuir els productes de submatrius entre fils permet, en el millor dels casos, dividir aproximadament el temps per un factor p (per exemple, una multiplicació clàssica paral·lela tindria cost aproximadament $\Theta(n^3/p)$). En el cas de l'algorisme de Strassen, gràcies a les 7 multiplicacions de submatrius independents per nivell recursiu, hem pogut executar fils en paral·lel en cada pas, obtenint teòricament un temps aproximadament $\Theta(n^{2.8074}/p)$. Així, l'acceleració teòrica màxima és lineal en funció de p tant per a la suma com per a la multiplicació.

Finalment, cal esmentar que existeixen algorismes més avançats de multiplicació de matrius amb complexitats asimptòtiques encara millors que la de Strassen. Diverses investigacions en algorismes algebraics han aconseguit reduir l'exponent teòric de la multiplicació: per exemple, l'algorisme de Coppersmith-Winograd va establir $\omega < 2.376$ el 1990, i treballs més recents han arribat fins a exponents propers a 2.37 o fins i tot menors. Un dels resultats actuals més destacats ès el de V. Vassilevska Williams (2012), que va obtenir un algorisme amb $\omega \approx 2.3727$. Tot i aquests avenços teòrics, *cap* d'aquests algorismes complicadíssims es fa servir en aplicacions pràctiques convencionals. En problemes del món real, els algorismes clàssics ben optimitzats (potser combinats amb tècniques com Strassen en certs rangs de mida) solen oferir el millor equilibri entre eficiència i simplicitat.

Resumint els resultats, podem afirmar que el cost asimptòtic obtingut per a cada operació s'ajusta al *esperat teòricament: la suma de matrius creix amb n^2 i la multiplicació clàssica amb n^3, però hem comprovat que ès possible reduir l'exponent de la multiplicació mitjanç ant estratègies de divideix-i-venceràs (Strassen) i que, a la pràctica, aquesta millora només es manifesta en escenaris de dades massius. Les optimitzacions de baix nivell i el paral·lelisme han demostrat tenir un impacte decisiu en el rendiment real: sense canviar la classe de complexitat, permeten aprofitar al màxim els recursos disponibles i acostar l'execució als límits teòrics de rendiment. En conjunt, les implementacions desenvolupades (suma concurrent i multiplicació de Strassen concurrent) han assolit acceleracions notables respecte als algorismes base seqüencials, validant així els beneficis tant de millorar l'algorisme (reduir operacions) com de paral·lelitzar el càlcul en múltiples nuclis.

Referències

[1] V. Strassen, *"Gaussian Elimination is not Optimal."* Numerische Mathematik, vol. 13, no. 4, 1969, pp. 354–356.

[2] D. Coppersmith and S. Winograd, *"Matrix Multiplication via Arithmetic Progressions."* Journal of Symbolic Computation, vol. 9, no. 3, 1990, pp. 251–280.

[3] V. Vassilevska Williams, *"Multiplying matrices faster than Coppersmith-Winograd."* In *Proceedings of the 44th ACM Symposium on Theory of Computing (STOC 2012)*, ACM, 2012, pp. 887–898.

[4] G. M. Amdahl, *"Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities."* AFIPS Conference Proceedings (30), 1967, pp. 483–485.

[5] K. Goto and R. A. van de Geijn, *"Anatomy of high-performance matrix multiplication."* ACM Transactions on Mathematical Software, vol. 34, no. 3, 2008, Article 12.

[6] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn, *"Strassen's algorithm reloaded."* In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*, IEEE Press, 2016, pp. 690–701.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. 3rd ed., MIT Press,

2009.
[8] P. Bürgisser, M. Clausen, and A. Shokrollahi, *Algebraic Complexity Theory*. Springer-Verlag, 1997.