Pràctica 4

Mapat a memòria i algorismes paral·lels.

Sistemes Operatius 1

Abril 2022

Índex

1 Introducció

Actualment, la majoria d'algorismes d'intel·ligència artificial fan un ús intensiu del càlcul matricial. De fet, a la branca de l'aprenentatge profund, les matrius amb les quals tractem usualment són matrius gegants amb milers o inclús milions de files i de columnes. Normalment aquestes matrius tenen certes propietats especials que fan que les estructures de dades usuals que utilitzem per representar matrius (*arrays*) no siguin recomanables. Tanmateix, per a casos generals, aquesta representació segueix essent una de les més populars, degut a la seva facilitat d'ús, ja que s'integra en gairebé tots els llenguatges de programació i és fàcil d'entendre i manipular.

En aquesta pràctica, realitzarem una de les operacions més bàsiques (i importants!) que es poden realitzar amb matrius, la multiplicació. Tanmateix, tractarem dos problemes diferents que es produeixen en el context del paràgraf anterior, és a dir, quan tenim matrius amb un gran nombre d'elements. El primer problema és la representació d'aquestes matrius: si una matriu es prou gran, la nostra memòria RAM no serà capaç de gestionar-la adequadament i llavors necessitarem una manera alternativa per treballar amb aquestes. El segon problema és el cost de la multiplicació de matrius: com ja sabeu de l'assignatura d'algorísmica, multiplicar dues matrius (per simplificar, quadrades amb n files i n columnes) te un cost $\mathcal{O}(n^3)$. Aquest cost per a n grans és massa alt i hem de trobar alguna manera de millorar l'execució d'aquest algorisme.

Per solucionar aquests problemes farem ús del mapat a memòria, que ens permet gestionar un fitxer a disc com si fos un element a la memòria principal, i el paral·lelisme, que ens permetrà paralel·litzar el nostre algorisme d'una manera fàcil.

2 Multiplicació de matrius

Donades dues matrius A (n files i m columnes) i B (m files i k columnes), el seu producte ve donat per la següent fòrmula:

$$
\begin{aligned}
A \cdot B &= \begin{pmatrix} a_{0,0} & \dots & a_{0,m-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \dots & a_{n-1,m-1} \end{pmatrix} \cdot \begin{pmatrix} b_{0,0} & \dots & b_{0,k-1} \\ \vdots & \ddots & \vdots \\ b_{m-1,0} & \dots & b_{n-1,k-1} \end{pmatrix} \\
&= \begin{pmatrix} \sum_{i=0}^{m-1} a_{0,i} b_{i,0} & \dots & \sum_{i=0}^{m-1} a_{0,i} b_{i,k-1} \\ \vdots & \ddots & \vdots \\ \sum_{i=0}^{m-1} a_{n-1,i} b_{i,0} & \dots & \sum_{i=0}^{m-1} a_{n-1,i} b_{i,k-1} \end{pmatrix}.
\end{aligned}
$$

on l'anterior matriu te n files i k columnes.

Per implementar aquest producte amb paral·lelisme, noteu que cada fila de A és utilitzada de forma independent a les altres files al producte $A \cdot B$. Per tant, si creem una matriu *buida* amb les dimensions adequades, $n \times k$, podem realitzar el producte de cada fila paral·lelament.

Tanmateix, crear un procés per cada fila és un procés costós e ineficient. Una opció molt millor és crear pocs processos (no més que el número de processadors de l'ordinador que executa el programa) i assignar-li un número particular de files que multiplicar. Per fer açò, podem assignar a tots els processos un número de files igual al quocient de la divisió entera $c = n//$ número de Processos. Com $n = c \cdot$ número de Processos $+ r$, amb $r <$ número de Processos, que és un número petit (menor que el nombre de processos), les r files no assignades se les assignarem a l'últim procés, de

manera que totes les files hauran estat distribuïdes quasi uniformement entre tots els processos, fent el procés de multiplicació més eficient.

3 La pràctica

La pràctica consisteix en tres apartats diferents. Durant tota la pràctica farem servir la estructura de dades següent:

```
struct Matrix {
    int numberOfRows;
    int numberOfColumns;
    float *grid;
    int fd;
}
```

A més a més, no està permès fer ús de l'operador `array[índex]`. Per accedir al valor d'un `array` farem servir l'operador estrella a punters, `*punter`. No és pot modificar el valor d'un array, ni accedir a ell, mitjançant el següent truc, que emula l'operador `array[fila*columnes + columna]`, `*(array + fila*columnes + columna)`. La raó per la qual no està permès fer aquests tipus d'operacions és perquè volem evitar fer operacions innecessàries (un producte i una suma) cada cop que vulguem accedir a un element de la matriu. Llavors, per iterar sobre els valors de la matriu farem servir aritmètica de punters, iterant sobre punters que apunten a la zona de memòria utilitzada per la matriu i utilitzant només l'operador accés `*` directament sobre aquests punters.

3.1 Exercici 1

El primer objectiu d'aquest exercici és dissenyar una funció que et permeti crear una matriu mapada a memòria del tipus `Matrix` especificada una ruta a un fitxer (assumirem que no existeix, per fer-ho més fàcil) i un número de files i de columnes. A més a més, aquesta funció ha d'emplenar la matriu amb els coeficients donats per una funció `float coeffFunction(int row, int column)`.

Aquesta matriu ha de poder ser modificada durant tota l'execució del programa. Per fer-ho, utilitzarem el mètode `mmap` amb mapat **no anònim**. El manual d'aquesta funció es pot trobar al següent enllaç. Per fer servir aquesta funció, necessitareu un descriptor de fitxer. El descriptor de fitxer pot ser aconseguit fent ús de la funció `open`, per obrir fitxers. El manual d'usuari de la funció `open` pot ser trobat en aquest enllaç. Recordeu que quan acabeu d'usar el mapat i el fitxer heu de *alliberar-los!* (fent ús de les funcions `munmap` i `close`).

El segon objectiu d'aquest exercici és veure que les matrius creades prèviament persisteixen a disc i poden ser reutilitzades. Per aquest exercici, s'ha d'implementar una funció

```
void exercici1()
```

que

1. Creï matrius A de 3 files i 3 columnes i B de 3 files i 2 columnes amb una funció que compleixi els requeriments anteriors. Aquestes matrius han d'estar inicialitzades amb coeficients $c_{i,j} = i^j$,

és a dir:

$$\begin{pmatrix} c_{0,0} & c_{0,1} & \dots & c_{0,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n-2,0} & c_{n-2,1} & \dots & c_{n-2,m-1} \\ c_{n-1,0} & c_{n-1,1} & \dots & c_{n-1,m-1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & n-2 & \dots & (n-2)^{m-1} \\ 1 & n-1 & \dots & (n-1)^{m-1} \end{pmatrix}.$$

2. Alliberi els recursos utilitzats per les estructures d'ambdues matrius, tancant els seus fitxers i desmapant-les amb la funció `munmap`.

3. Creï noves matrius utilitzant els fitxers existents de les matrius A i B.

4. Imprimeixi les noves matrius per pantalla.

5. Alliberi tots els recursos emprats per a les noves matrius i elimini els fitxers associats a les matrius (pots utilitzar la funció `unlink`).

Quan executis aquest exercici, comprova que els valors de les matrius són els mateixos per a les matrius originals i per a les carregades de disc.

Per implementar aquest exercici, es recomana seguir els següents passos:

1. Implementar una funció:

```
struct Matrix createEmptyMatrix(int numberOfRows,
int numberOfColumns, char* filepathPersistence),
```

que creï una matriu buida a la ruta `filepathPersistence`.

2. Implementar una funció:

```
float VandermondeCoefficients(int row, int column) {
    if(column == 0)
        return (float)1.0;
    return (float)pow(row, column);
}
```

3. Implementar una funció:

```
void fillMatrix(struct Matrix *matrix,
FILL_MATRIX_FUNC func),
```

on

```
typedef float (*FILL_MATRIX_FUNC)(int, int);
```

on la definició de tipus nou representa un punter a una funció que retorna un valor `float` i te com a entrada dos paràmetres `int`. En el nostre cas, la funció que passarem per paràmetre a `fillMatrix` serà la funció, definida prèviament, `VandermondeCoefficients`.

4. Implementar una funció

```
void printMatrix(struct Matrix *matrix)
```

5. Crear un mètode

   ```
   struct Matrix loadSavedMatrix(int numberOfRows,
   int numberOfColumns, char* filepath)
   ```

   que retorni una matriu de tipus `Matrix` que estigui carregada d'un fitxer existent `filepath` creat (i possiblement modificat) prèviament amb el mètode `createEmptyMatrix`.

6. (A la funció `exercici1`) Crear les matrius A (3 files i 3 columnes) i B (3 files i dos columnes) fent servir les funcions anteriors juntament amb la funció `float VandermondeCoefficients(int row, int column)`. Utilitzar la funció `printMatrix` per imprimir per pantalla les dues matrius A i B.

7. (A la funció `exercici1`) Per a les matrius A i B, tancar els descriptors de fitxers i *desfer* (`munmap`) el mapat a memòria de les matrius, però no eliminar els fitxers generats al crear les matrius.

8. (A la funció `exercici1`) Fer servir el mètode `loadSavedMatrix` per crear noves matrius amb els fitxers de les anteriors.

9. (A la funció `exercici1`) Imprimir les matrius i observar que són iguals.

3.2 Exercici 2

El objectiu d'aquest exercici és implementar el producte de matrius estàndard entre dos matrius de tipus `Matrix` amb paral·lelisme, tal i com està explicat a la secció 2. Aquest producte serà una funció que retornarà una altra matriu mapada a memòria `Matrix` especificada una ruta a un fitxer que, com abans, assumirem que no existeix.

Per a assolir aquest objectiu i testejar el correcte funcionament de les funcions implementades, implementa una funció

```
void exercici2()
```

que

1. Creï matrius A de 3 files i 3 columnes i B de 3 files i 2 columnes com a l'exercici anterior.

2. Multipliqui les dues matrius A i B (amb paral·lelisme).

3. Mostri per pantalla les dues matrius A i B i el seu producte $A \cdot B$.

4. Alliberi els recursos utilitzats i elimini els arxius associats a les tres matrius.

Aquest mètode hauria d'imprimir per pantalla:

```
Vandermonde matrix A with rows = 3 and columns = 3:
1.000000 0.000000 0.000000
1.000000 1.000000 1.000000
1.000000 2.000000 4.000000
```

```
Vandermonde matrix B with rows = 3 and columns  = 2:
1.000000 0.000000
1.000000 1.000000
1.000000 2.000000
A*B:
1.000000 0.000000
3.000000 3.000000
7.000000 10.000000
```

Per fer aquest exercici, es recomana:

1. Implementar una funció de producte de matrius sense paral·lelisme.

   ```
   struct Matrix naiveMultiplyMatrices(struct Matrix *A,
   struct Matrix *B, char* filepathPersistence)
   ```

2. Un cop funcioni la implementació anterior, implementar la funció de producte amb paral·lelisme

   ```
   struct Matrix naiveMultiplyMatricesParallel(struct Matrix *A,
   struct Matrix *B, char* filepathPersistence,
   int numberOfProcesses)
   ```

   Per crear una matriu buida, com està especificat a la secció 2, es pot utilitzar la funció `ftruncate`, el manual de la qual es pot trobar aquí. Es recomana seguir els següents passos:

   (a) Crear una funció

   ```
   void multiplyMatricesProcessDelegation(int initialRow,
   int finalRow, struct Matrix *A, struct Matrix *B,
   struct Matrix *resultMatrix)
   ```

   que donades matrius `A` i `B` existents, una matriu buida `resultMatrix` i unes files inicial i final calculi els coeficients corresponents als productes relacionats amb les files de la matriu `A` compreses entre `initialRow` i `finalRow` i els escrigui a la matriu `resultMatrix`.

   (b) A la funció `naiveMultiplyMatricesParallel` crear tants processos com `numberOfProcesses` on cada procés executa la funció creada al punt anterior amb les files que li corresponguin, tal com està explicat a la secció 2.

3. (A la funció `exercici2`) Crear matrius A i B com les de l'exercici 1. A més a més, fer el seu producte i imprimir les dues matrius i el seu producte (amb paral·lelisme, amb la funció `naiveMultiplyMatricesParallel`) per pantalla.

4. (A la funció `exercici2`) Netejar els arxius temporals, tancar i alliberar memòria dels recursos que ho necessitin.

3.3 Exercici 3

En aquest exercici compararem les implementacions amb i sense paral·lelisme implementades a l'exercici 2. Nota: Aquest exercici s'haurà d'executar **fora** de la màquina virtual!

Per fer-ho, crea una funció

```
void exercici3()
```

que

1. Creï matrius A i B, ambdues 900 files i 900 columnes amb els mateix coeficients utilitzats als exercicis 1 i 2, és a dir, matrius $A, B = (c_{i,j})_{i,j \in \{0,\dots,899\}}$ on $c_{i,j} = i^j$.
2. Calculi el producte de les matrius 10 cops amb i sense paral·lelisme (amb 4 processos). Enregistreu el temps en segons (representats amb `float`, i no amb `int`) que triguen totes les crides a les funcions de producte. Imprimiu aquest valor per pantalla.
3. Imprimiu per pantalla la mitjana de temps d'execució per a les implementacions amb i sense paral·lelisme.

A la memòria, respon a les següent preguntes:

La implementació amb paral·lelisme millora el temps d'execució? Com canvien aquestes mitjanes en funció del nombre de processos? És cert que si augmentem el nombre de processos sempre obtindrem millors resultats? Raona les respostes i aporta dades numèriques que la validin.

3.4 Depuració del codi

Utilitzeu `Valgrind` per assegurar-vos de que allibereu correctament la memòria. A més a més, heu de fixar-vos en tancar correctament tots els fitxers oberts prèviament durant el mapat a memòria i de desfer el mapat a memòria quan calgui (per exemple, després de cada experiment a l'exercici 3 seria convenient alliberar els recursos dels productes realitzats). Per utilitzar Valgrind amb múltiples processos, heu d'utilitzar l'opció `-trace-children=yes`. Per a més informació es pot consultar el manual de Valgrind.

4 Entrega

El lliurament consisteix en dues parts: **el codi font** (80% de puntuació) i **l'informe** (20% de puntuació). Les dues parts s'han de comprimir en un **únic fitxer ZIP** i lliurar tal com s'indica a la tasca del campus.

4.1 Codi font

Es demana lliurar els següents fitxers:

- El codi font que implementi la funcionalitat descrita a la secció 3. Tot el codi es pot implementar en un únic fitxer C si així es vol. La funció `main` estarà implementada **exactament** com:

```
void main() {
    exercici1();
    exercici2();
    exercici3();
}
```

- Un `Makefile` que permeti compilar el codi. L'estructura d'aquests fitxers està descrita en un document del campus, a la secció de teorico-pràctica.

El codi font no tindrà arguments d'execució, i s'executarà com:

```
$ ./practica4
```

4.2 Informe

L'informe a lliurar ha d'estar en **format PDF o equivalent (no s'admeten formats com odt, docx,...)**. Una bona manera d'escriure un informe tècnic és dividir-ho en tres parts: **introducció, informe/proves realitzades i conclusions.**

A la part d'introducció es descriu breument el problema solucionat (i.e. un resum del que proposa aquesta pràctica). A la part de l'informe/proves es demana:

- Mostrar les proves que s'han fet per assegurar el bon funcionament del codi.
- Donar resposta a les preguntes de l'exercici 3.

Sigueu breus i clars en els comentaris i experiments realitzats, no cal que us estengueu al text escrit. No es necessari que expliqueu el codi font llevat que hi hagi alguns detalls que vulgueu esmentar respecte la vostra implementació.

Finalment, a la part de conclusions es descriuen unes conclusions tècniques de les proves realitzades. Per acabar, es poden incloure conclusions personals.

En cas que vulgueu incloure captures de pantalla en comptes d'incloure els resultats dels experiments en format text, assegureu-vos que el text de la captura es pot llegir bé (és a dir, que tingui una mida similar a la resta del text del document) i que totes les captures siguin uniformes (és a dir, que totes les captures tinguin la mateixa mida de text).

El document ha de tenir una longitud màxima de 3 pàgines (sense incloure la portada). El document s'avaluarà amb els pesos següents: proves realitzades i comentaris associats, un 60%; escriptura sense faltes d'ortografia i/o expressió, un 20%; paginació del document feta de forma neta i uniforme, 20%.