Optimering av matrisbibliotek i C

Redaktör: Martin Söderén

Version 0.1

Status

Granskad	Martin Söderén	-
Godkänd	Andreas Runfalk	-

Innehåll

1 Inledning

Denna rapport går igenom vilka optimeringar som kunde göras på att befintligt matrisbibliotek. I slutändan så implementerades optimeringarna och jämförelser gjordes mellan den gamla och den nya versionen av biblioteket.

1.1 Syfte

Optimera ett matrisbibliotek som en lösare av kvadratiska konvexa optimeringsproblem ska använda.

1.2 Frågeställning

Finns det möjlighet till förbättringar vad gäller prestanda i den befintliga implementationen av matrisbiblioteket matLib som har tagits fram under kandidatprojektet?

1.3 Avgränsningar

Matrisbiblioteket kan körs på alla platformar som har en c kompilator men i denna rapport så körs det endast x86-64.
Koden kompileras utan optimeringsflaggor så att alla optimeringar sker i koden.
Bibilioteket kan användas med olika precisioner på flyttal. De möjliga precisionerna är:

- float
- double
- quad

Det kan även användas med bara heltal men de flesta av funktionerna fungerar inte med heltal. I denna rapport så göras alla jämförelser med precisionen satt till double.

2 Bakgrund

Under kandidatprojektet Prediktionsreglering så skulle en lösare av kvadratiska komplexa problem tas fram. För detta behövdes ett matrisbibliotek väljas men inget bibliotek uppfyllde kraven. Dessa var:

1. Lättanvänt api
2. Bra prestanda
3. Platformsoberoende
4. Lätt att kompilera
5. Tar upp lite minne
6. Bra dokumenterad kod så man själv kan implementera förbättringar

Inget bibliotek uppfyllde alla dessa krav. De som undersöktes var:

- GNU Scientific library
- LAPACK
- ATLAS
- NAG

Då togs beslutet att ta fram ett eget bibliotek som döptes till matLib. De matrisoperationer som behövdes var:

- addition
- subtraktion
- multiplikation
- beräkna determinat
- beräkna invers
- lösa linjära ekvationssystem
- gausselimination
- transponering
- skalärmultiplikation

Efter implementation ska nu eventuella optimeringar undersökas.

3 Teori

3.1 Allmän design av biblioteket

Alla operationer på matriser och vektorer bygger i grunden på matris structen som är definierad i matLib.h.

```
struct matrix {
        int columns;
        int rows;
        size_t size;
        value *start;
};
```

När en matris skapas så används följande funktion:

```
matrix* create_matrix(int row, int col) {
  if (row < 1 || col < 1) {
    return NULL;
  }

  matrix* mal = (matrix *) malloc(sizeof(matrix));
  mal->columns = col;
  mal->rows = row;
  mal->size = row * col;
  mal->start = (value *) malloc(col * row * sizeof(value));
  return mal;
}
```

Som skapar en *row x column* stor matris.

För att sätt in ett värde i matris används följande funktion:

```
bool insert_value(value insert,int row, int col, matrix* mat) {
  if (!check_boundaries(row, col, mat)) {
    return false;
  }
```

```
  value* start = mat->start + mat->columns * (row - 1) + (col - 1);
  *(start) = insert;
  return true;
}
```

Matriserna lagras row major så datan ligger lagrad enligt figur 1.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figur 1 – Ordning som data i matrisen lagras på

3.2 Tidskomplexitet på nuvarande implementationer operationer

Alla tidskomplexiteter beräknas på nxn matriser. Addition: $\mathcal{O}(n^2)$
Subtraktion: $\mathcal{O}(n^2)$
Multiplikation: $\mathcal{O}(n^3)$
Invers (crout och sedan lösa n ekvationssystem): $\mathcal{O}(n^3)$

3.3 Algoritmiska förbättringar

Multiplikation:
Istället för den naive algoritmen kan strassen algoritmen implementeras vilket reducerar tidskomplexiteten från $\mathcal{O}(n^3)$ till $\mathcal{O}(n^{2.807})$ [2].
Inverse:
Istället för crout så kan inversen beräknas med strassen algoritmen och på så sätt så sänks tidskomplexiteten från $\mathcal{O}(n^3)$ till $\mathcal{O}(n^{2.807})$. Denna algoritm har dock en stor nackdel som beskrivs i sektion 3.5.

3.4 Strukturella förbättringar

När ett linjärt ekvationssystem på formen $Ax = b$ löses så bryts A ned till två matriser L och U där L är en undre triangulär matris och U är en övre triangulär matris. I matris structen skulle dessa matriser kunna sparas tillsammans med en boolean som säger om matrisen har modifierats så skulle man kunna undvika att beräkna U och L flera gånger för samma matris. Samma sätt skulle kunna användas för inversen till matriser.

3.5 Strassen algoritmen för beräkning av inverser

Matrisen man ska invertera delas upp i fyra submatriser på följande sätt:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Där inversen till A beräknas på följande sätt[1]:

$$\begin{aligned}
R_1 &= A_{11}^{-1} && (1)\\
R_2 &= A_{21}R_1 && (2)\\
R_3 &= R_1A_{12} && (3)\\
R_4 &= A_{21}R_3 && (4)\\
R_5 &= R_4 - A_{22} && (5)\\
R_6 &= R_1^{-1} && (6)\\
X_{12} &= R_3R_6 && (7)\\
X_{21} &= R_6R_2 && (8)\\
R_7 &= R_3X_{21} && (9)\\
X_{11} &= R_1 - R7 && (10)\\
X_{22} &= -R_6 && (11)\\
A^{-1} &= \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} && (12)
\end{aligned}$$

Här så kan A_{11}^{-1} och R_1^{-1} beräknas rekursivt med samma algoritm. Det stora problemet med denna algoritm är att samtliga submatriser måste vara inverterbara. Det vill säga följande måste vara uppfyllt:

$$\begin{aligned}
A_{11}A_{11}^{-1} &= I_n && (13)\\
A_{12}A_{12}^{-1} &= I_n && (14)\\
A_{21}A_{21}^{-1} &= I_n && (15)\\
A_{22}A_{22}^{-1} &= I_n && (16)\\
& && (17)
\end{aligned}$$

Ta till exempel identitetsmatrisen I_n som är invertbar då den är sin egna invers. Om denna delas upp i submatriser enligt strassen algoritmen så får man följande:

$$I_n = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

$$I_nI_n = I_n$$

Anta att n är ett jämnt tal så I_n kan delas upp i fyra lika stora submatriser.

$$I_{11} = I_{22} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} I_{12} = I_{21} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

Då I_{12} och I_{21} är nollmatriser och inte inverterbara så kan strassen algoritmen ej tillämpas på alla matriser även om de är inverterbara.

3.6 Strassen algoritmen för matrismultiplikation

Anta att vi vill multiplicera två matriser A och B till produkten C. Dessa delas upp i fyra submatriser på följande sätt:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Skulle radantalet vara ojämnt så fylls matriserna ut med en extra nollrad. Samma sak gäller kolumnerna. Vi definierar följande matriser:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \tag{18}$$
$$M_2 = (A_{21} + A_{22})B_{11} \tag{19}$$
$$M_3 = A_{11}(B_{12} - B_{22}) \tag{20}$$
$$M_4 = A_{22}(B_{21} - B_{11}) \tag{21}$$
$$M_5 = (A_{11} + A_{12})B_{22} \tag{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) \tag{23}$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \tag{24}$$

Resultatmatrisen C beräknas på följande sätt:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \tag{25}$$
$$C_{11} = M_1 + M_4 - M_5 + M_7 \tag{26}$$
$$C_{12} = M_3 + M_5 \tag{27}$$
$$C_{21} = M_2 + M_4 \tag{28}$$
$$C_{22} = M_1 - M_2 + M_3 + M_6 \tag{29}$$

Denna algoritm har tidskomplexitet $\mathcal{O}(n^{2.807})$ jämfört med den naiva algoritmen som har tidskomplexitet $\mathcal{O}(n^3)$. Denna kan även utvecklas till att köras parallel då beräkningen av matriser M_{1-7} kan utföras helt åtskilda. Matrismultiplikationerna i denna algoritmen kan sedan rekursivt använda samma algoritm och på så sätt kan processen använda många trådar. Antalet trådar beror på när man väljer att börja använda den naiva algoritmen. Vid små matriser kan strassen vara mindre lämpad då det tar tid att dela upp matriserna i mindre matriser samt använder upp mer minne. Väljer man att dela upp problemet till flera trådar så tar det tid att starta trådarna så man måste bestämma vid vilken storlek av matriser som man börjar använda den naiva algoritmen. Detta beror självklart på arkitekturen men tester kommer utföras senare pån x86-64 arkitekturen för att se när denna algoritm lämpar sig bäst.

3.6.1 Nackdelar med strassen

I den nuvarande implementationen så skapar det nya matriser i varje rekursivt anrop till strassen. Detta kan leda till att den använder mycket minne. Detta skulle kunna återgärdas med att inga nya matriser skapas utan varje rekursivt anrop får de ursprungliga matriserna och en mängd som de ska arbeta med. Alternativt sätta rekursionsdjupet beroende på storleken av matriserna och mängden tillgängligt ram-minne.

4 Metod

Först undersöktes de strukturella förbättringarna som har föreslagits i biblioteket och sedan de algoritmiska.

4.1 Strukturella förbättringar

Tidigare så föreslogs att matrisernas L och U matriser kunde sparas i matrix-structen så dessa inte behövdes beräknas varje gång. Detta fungerar bra så länge matrisen inte ändras och då detta behövs då kollas i alla funktioner som ändrar på matrisens innehåll. Så structen skulle isåfall innehålla två extra matriser och en booleank variabel som höll koll på om matrisen ändrades. Att testa i alla funktioner bestämdes till att vara väldigt överflödigt och om någon lägger till en funktion och glömmer lägga in testet så faller hela konceptet. Samma sak gäller om någon väljer att manipulera datan i matrisen direkt utan att använda bibliotekets funktioner så detta alternativ gick bort.

4.2 Algoritmiska förbättringar

Den förbättringen som valdes till den bästa för biblioteket var att använda strassen istället för den naiva algoritmen när man utförde matrismultiplikationer. Först så implementerades algoritmen och sedan så gjordes tester för att bestämma när det blev mer optimerat att använda den istället för den naiva. I den nuvarande implementation så används funktionen multiply_matrices som innehåller den naiva algoritmen. Detta gjordes om så att den funktionen kallar på multiply_matrices_naive om storleken på den resulterande matrisen understiger gränsvärdet som mättes upp , annars kallar funktionen på strassen_matrices för matrismultiplikation.

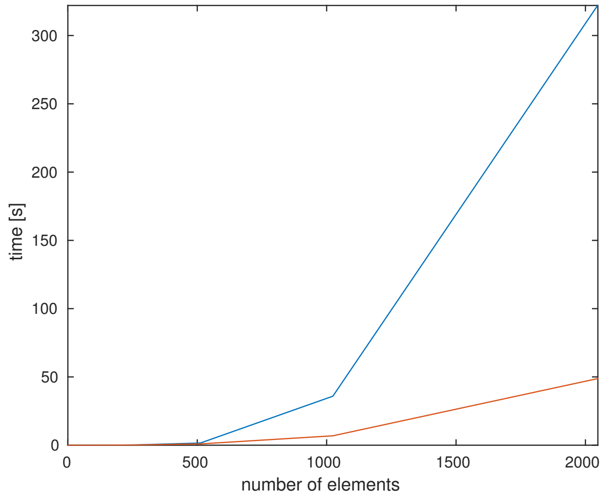
För att sedan testa om biblioteket kunde utnyttja en paralleliserad version av Strassen så implementerades strassen_matrices_parallel. Dock för att bibehålla kompabiliteten med alla platformen så kan all denna kod tas bort av preprocessorn genom att inte sätta PARALLEL flaggen i matLib.h

4.3 Implementation av Strassen algoritmen

Implementationen använder algoritmen beskriven i 3.6. Dock beroende på utformning av bibliotektet så måste en ny matris skapas för varje operation man gör. Detta leder till att under en multiplikation utan rekursivt anrop så skapas och frigörs 38 matriser.

4.4 Jämförelser mellan algoritmer

I figur 2 kan man se jämförelsen mellan Strassen och naive. Upplösningen mellan punkter är rätt dålig men det tar lång tid att beräkna punkter över 500 element. Datan är dock genomsnittet av 5 tester per algoritm. Man kan tydligt se att över 500 element så har Strassen ett stort övertag över naive.



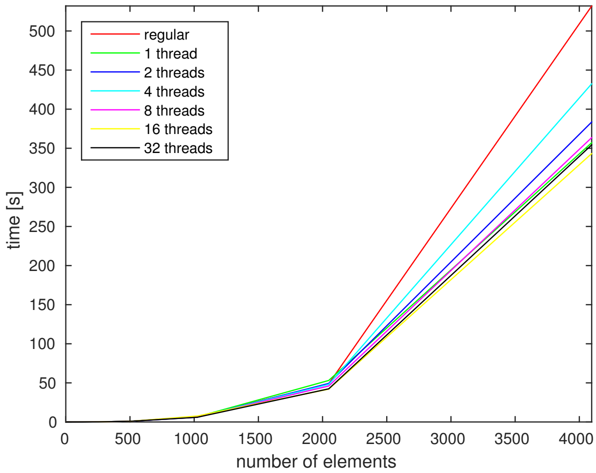
Figur 2 – Jämförelse mellan naive och Strassen.

4.5 Implementation av Strassen paralelliserad

Implementation använder algoritmen beskriven i 3.6. Det som sker nu är att varje gång någon av matriserna M_{1-7} ska beräknas och om det finns en ledig tråd så körs detta i den. Annars beräknas den i den nuvarande tråden. För att hålla koll på antalet trådar som körs så finns den globala räknaren thread_counter som skyddas av ett lås för att försäkra ömsesidig uteslutning. Maximala antalet trådar sätts i den statiska variabeln number_of_cores. När någon nuvarande tråd kommer till ett tillfälle den den ska beräkna någon av M_{1-7} så låser den thread_counter och jämför denna med number_of_cores och om dessa är olika så beräknas multiplikationen med strassen_matrices_parallel. Detta skapar en ny tråd för beräkningen. Annars så utförs beräkning med strassen_matrices som då sker i den nuvarande tråden.

4.6 Jämförelse mellan paralleliserad och oparalleliserad Strassen

Jämförelsen är svår att göra då man kan variera båda antalet element och antalet trådar som processen använder. När tidigare tidsmätningar har gjorts så har c-bibliotket time.h användts men detta mäter den effektiva cpu-tiden. När man använder fler trådar så är detta missvisande då det intressanta är reela tiden det tar. För att lösa detta så används bash kommandot time för att mäta tiden istället. Resultatet från jämförelsen kan ses i figur 3. Detta test utfördes på en bärbar dator med en tvåkärning I7 processor med Hyper threading vilket gör att det kan jämföras med en fyrakärnig processor. Helst skulle testet utförts på en dator med fler kärnor. I figur 3 kan man se en tydlig prestande ökning från vanliga algoritmen till algoritmen när den använder en extra tråd. Därefter avtar prestandeökning. Att multiplicera 2 matriser som var 4096x4096 tog 582 sekunder med den vanliga och som bäst 343 sekunder när den använda 16 trådar. Samma beräkning med den naiva algoritmen tog 2688.527 sekunder.



Figur 3 – Jämförelse mellan vanliga Strassen och parallel version.

5 Resultat

Den enda förbättringen som gjordes i matrisbiblioteket var att för matrismultiplikation där storleken på den resulterande matrisen överstiger 512x512 element så används Strassen algoritmen istället för beräkningen. Det leder till att beräkningen utförs med tidskomplexitet $\mathcal{O}(n^{2.807})$ istället för $\mathcal{O}(n^3)$. I praktiken så gör detta stor skillnad vad gäller prestande. Redan vid 512*512 element så är Strassen 5 gånger snabbare. Den är snabbare redan vid 128 element men inte så mycket så att det väger upp för det extra minnet den kräver. Vid multiplikation av två matriser med 4096*4096 element så tog Strassen 582 sekunder medan den naiva algoritmen tog 2689 sekunder. Påverkan på lösaren som utnyttjar detta bibliotek blir dock minimal då det troligtvis inte kommer använda matriser som är större än 512x512 men om man vill göra det i framtiden så blir körtiden troligvis mindre med denna implementation.

Den parallel versionen av Strassen är snabbare än den vanliga Strassen när man överstiger 2048 element. För att verkligen testa detta skulle man behöva en dator med lika många kärnor som trådar man testar.

6 Diskussion

6.1 Resultat

Resulatet är rimligt och var inte oväntat.

6.2 Metod

Det skulle behövas tillgång till bättre hårdvara för att verkligen testa körtiderna i den parallela versionen.

7 Slutsatser

Strassen är snabbare än naive.

Referenser

[1] Marko D. Petković and Predrag S. Stanimirović. Block recursive computation of geneeralized inverses. *Electronic journal of Linear Algebra*, 26:394–405, 2013.

[2] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.