Ukeoppgaver uke 3 INF2440 – v2017

Denne uka skal vi se på det å multiplisere to nxn matriser A og B sammen og regne ut resultatmatrisen C, altså C = AxB , først sekvensielt (koden for en rett fram implementasjon av den sekvensielle versjonen finner du nederst i tips) og så parallelliser dette. Dette kan se ut som en veldig matematisk oppgave, og matriser brukes mye i løsning av store ingeniør-beregninger, men for å løse denne oppgaven trenger men ikke vite hva matriser brukes til. Det holder å vite at en matrise er simpelthen en todimensjonal double-array (eks. double [][] a = new double[n][n]; hvor n er et passe heltall – i vårt tilfelle mindre enn 1000).

Resultatet av AxB er selv en nxn matrise C, og vi har at element *c*[i][j] i matrisen C er da definert som:

$$c[\mathrm{i}][\mathrm{j}] = \sum_{k=0}^{n} a[\mathrm{i}][\mathrm{k}] * b[\mathrm{k}][\mathrm{j}]$$

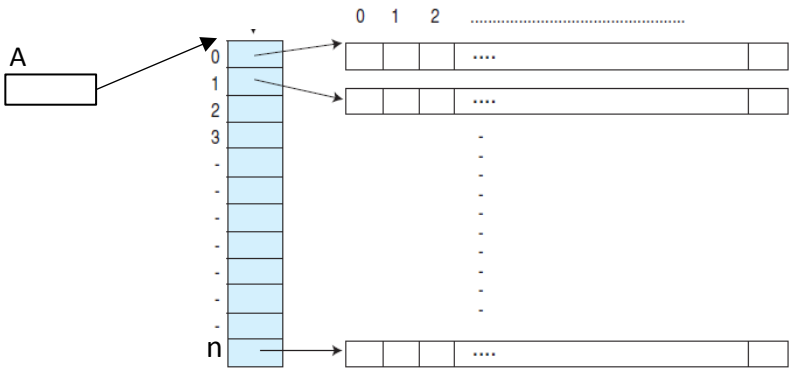
Vi multipliserer altså hvert element i rad_i fra A parvis med hvert element fra $\mathrm{kolonne}_j$ fra B og summerer de n multiplikasjonene for å få *c*[i][j] (se koden).

Sekvensiell implementasjon

a) Du skal før initiere matrisene A og b med tilfeldige double-tall fra klassen Random med metoden nextDouble(). Du får da et tilfeldig tall mellom 0,0 og 1.0. (Hvis du først vil teste koden din, kan du initiere A og B med 1.0, og da vil alle elementene i C være n.)

   Test kjøretiden for n=100, 500 og 1000 og lag en liten tabell om kjøretiden som funksjon av n. Forklar hvorfor kjøretiden 8-dobles når n dobles.

Her du ser på hvordan to-dimensjonale arrayer er lagret i Java:



Da ser du at når vi beregner ett element i C: *c*[i][j], så leser vi fra A tall fra en rad som ligger etter hverandre i lageret (raskt), mens elementene fra B hentes fra hver sin rad, og vi hopper rundt i lageret. Det er forventet at du vil få problemer med cache-ene, og da langsommere lesing av B-elementene. Du skal altså prøve ut en ide at før du multipliserer sammen A og B, så bytter du om på

elementene i B, slik at kolonnene i B ligger lagret radvis som rader (på matematisk kalles dette å **transponere** B). Dette oppnår du hvis du bytter alle elementer B[i][j] med B[j][i] for alle i <n og j<i (grunnen til dette siste j < i er å unngå å bytte om hvert element to ganger og dermed ende opp med at B ikke er endret) Du skal altså nå forsøke å multiplisere AxB med først å transponere B. Kall den B', og da er :

$$c[\mathrm{i}][\mathrm{j}] = \sum_{k=0}^{n} a[\mathrm{i}][\mathrm{k}] * \mathrm{b}'[\mathrm{j}][\mathrm{k}]$$

Ta også tider her (hvor tidene nå inkluderer først transponering av B og så multiplisering AxB' og se om du får raskere tider enn rett fram multiplisering som definert i pkt a).

Parallell implementasjon

Dette er ett av de problemene hvor parallellisering er meget lett, en av de trivielt parallellbare problemene. La bare tråd 0 beregne de n/k første kolonnene i C, tråd 1 de neste n/k kolonnene,,…, og den siste tråden alle de siste kolonnene i C.

Siden dette ikke består i å skrive samtidig på felles variable, bare lese A og B (eller B') , trenger vi ikke noen synkronisering, bare en skikkelig avslutning av trådene – for eksempel med join().

Inkluder din parallelle løsning med den beste av løsningene a) eller b) ovenfor og parallelliser denne. Hvis løsning b) med transponering var raskest, er det tilsvarende enkelt å parallellisere transponeringen av B. Skriv en liten tabell over kjøretidene og speedup for din parallelle løsning og kommenter den.

Tips

Koden for beregning av C = AxB:

```
double  elem ;

for (int i=0;i < n;i++) {
  // for each row i C
        for (int j=0;j < n;j++) {
                // for each collumn in C
                elem =0.0;
                for (int k=0;k < n;k++){
                        elem += a[i][k]*b[k][j];
                }
               c[i][j] = elem;
        } // end j
} // end i
```