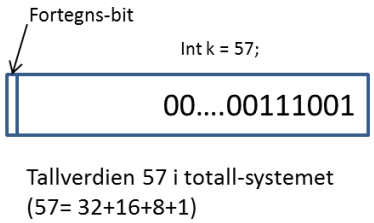
Ukeoppgaver i uke 4, INF2440 – v2017

1) Shift, bitvis AND (&), bitvis inklusiv eller (|), eksklusiv eller(^) mellom heltall (int, long, short, byte)

På heltall i Java kan man utføre operasjoner på de enkelte bit et heltall består av (en *int* har 32 bit (nummereres fra venstre mot høyre: 31, 30,29..1,0); hvorav det første (øverste, venstre) bit, dvs bit nr. 31 brukes til fortegn (1 = negativt tall, 0 = positivt tall). Det at det øverste bit nyttes til fortegn gjelder også long(da bit 63), short(bit 15) og byte (bit 7).



De bitene et heltall består av, kan vi endre på med flere operatorer, som da endrer tallverdien. Du skal nå lage et program som har deklarert int *a* =57 og som tester de ulike operatorene.

a) Skift operasjoner:
   a. *a*<<j , skifter alle bit i *a* venstre-over j plasser (fyller på med 0-bit nedenfra).
      i. Skriv en linje i programmet som skifter *a* 2 plasser venstre-over og som skriver ut 'gamle a' og resultatet med System.out.println().
   b. *a*>>>k, skifter alle bit i *a* k plasser høyreover (fyller på med 0-bit i de k øverste bit-ene).
      i. Skriv en linje i programmet som skifter (>>>) (-*a*) tre plasser høyreover og skriver ut resultatet sammen med 'gamle' (-*a*).
   c. *a*>>k, skifter alle bit i *a* k plasser høyreover (fyller på med verdien av fortegns-bitet i de k øverste bit-ene i *a*)
      i. Skriv en linje i programmet som dividerer *a* med 8 og som skriver ut 'gamle *a*' og resultatet.(Hint hvor mange bit må du skifte ned for å dele med 8)

b) Logiske bit-operatorer (de to operandene må være av samme lengde – eks. begge int).
   a. *AND: a* & m , hvert bit i *a* blir sammenlignet med motsvarende bit i m, og hvis begge disse bit er == 1, blir dette bitet i resultatet = 1, 0 ellers.
      i. Skriv en løkke i programmet som først AND-er *a* med 0..001, så med 0..000011 , så 0…000111 og skriver ut resultatet sammen med 'gamle' *a*.
   b. Inklusiv OR: *a* | m, hvert bit i *a* blir sammenlignet med motsvarende bit i m, og hvis mist en av disse bit er == 1, blir dette bitet i resultatet = 1, 0 hvis begge == 0.
      i. Skriv en løkke i programmet som først OR-er *a* med 0..001, så med 0..000011 , så 0…000111 Skriv ut resultatet sammen med 'gamle' *a*.

   c. Eksklusiv eller, XOR: *a* ^ *b*, hvert bit i *a* blir sammenlignet med motsvarende bit i *b*, og hvis ett og bare ett av disse bit er == 1, blir dette bitet i resultatet = 1, 0 hvis begge eller ingen av bitene == 0 (tilsvarer addisjon uten mente).

i. Skriv en kode i programmet som først XOR-er *a (=57)* med et vikårlig tall b (f.eks 10236) og skriver ut resultatet sammen med 'gamle' *a*. Så tar du det 'rare' tallet rr du fikk ut av denne XOR operasjoner og XORer omigjen med b og skriv ut. Kommenter resultatet. Ta så rr og XOR med a(=57) – hvilket tall får du da?

XOR er meget brukt i kryptering – mellomresultatet er uforstålig, men du får fram opprinnelig resultat ved å ta XOR en gang til med samme nøkkel (her b= 10236).

2) Fortsettelse av oppgaven med matrisemultiplisering

Denne uka skal vi også 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. Denne gangen skal vi dele opp data på en annen måte for trådene enn i Uke3. Anta at du har n rader og n kolonner i matrisen din og k tråder på din maskin. I uke3 skulle hver tråd beregne n/k ***kolonner*** i C, nå i uke4 skal vi se hva som skjer når trådene skal beregne n/k ***rader*** i C (husk at den siste tråden må regne ut alle de siste radene hvis n ikke er delbar med k).

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

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

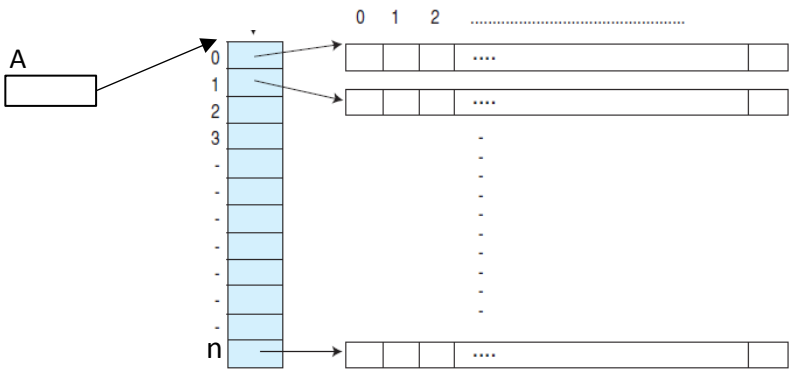
Vi multipliserer altså hvert element i rad i fra A parvis med hvert element fra kolonne j fra B og summerer de n multiplikasjonene for å få c[i][j] (se koden).

Sekvensiell implementasjon – som i Uke3 – du kan bruke samme kode.

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 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 mellom alle radene. Det er forventet at du vil få problemer med cache-ene, og meget langsommere lesing av B-elementene. Du skal altså prøve ut en idé at du 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 :

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

Du skal også etter å ha multiplisert slik, transponere B tilbake slik at du ikke 'ødelegger'(dvs. endrer) innholdet av B.

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

3) Parallell implementasjon – nå radvis

Dette er ett av de problemene hvor parallellisering er meget lett, et av de pinlig parallelliserbare problemene. La bare tråd 0 beregne de n/k første ***radene*** i C, tråd 1 de neste n/k radene,,…, og den siste tråden tar alle de siste radene 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 ovenfor og parallelliser denne. Siden løsningen med transponering var raskest, er det tilsvarende enkelt å parallellisere transponeringen av B. Begrunn hvorfor en transponering tar mye kortere tid enn selve multipliseringen. Det holder derfor bare i begge tilfellene å ikke parallellisere transponeringen. Skriv en liten tabell over kjøretidene og speedup for din parallelle løsning og kommenter den. ***Kommenter også ved å sammenligne resultatene fra Uke3 med disse resultatene:*** Hva var best: Dele opp C radvis (uke4) eller kolonnevis(uke3)? Forklar forskjellene du fant.

Tips

Koden for sekvensiell beregning av C = AxB:

```
double  elem ;

for (int i=0;i < n;i++) {
  // for hver rad  i C
        for (int j=0;j < n;j++) {
              // for hver kolonne i 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
```