Tweede deeltoets Concurrency

Dinsdag 8 november 2011, 8.30 – 10.30, Ruppert-Wit.

Motiveer je antwoorden, wees duidelijk en schrijf netjes.

1. **Alternatieve Prefix Sum.** De *prefix sum* van een rij $X[0..n-1]$ is de rij $Y[0..n-1]$, gedefinieerd door $Y[k] = \sum_{i=0}^{k} X[k]$. Als alternatief voor het behandelde algoritme bekijken we deze methode: (1) bereken de prefix sum van de linkerhelft en de rechterhelft; (2) tel bij de uitkomst van de rechterhelft, het totaal van de linkerhelft op.

   (a) Hoeveel rekenstappen kost het om de prefix sum met een *sequentieel* algoritme te bepalen?
   (b) Laat zien, hoe de alternatieve methode werkt, wanneer toegepast op de cijfers van je collegekaart (F=6 als die voorkomt).
   (c) Stel vergelijkingen op voor de *span* en het *work* van de alternatieve methode.
   (d) Wanneer is een parallel algoritme *efficient*? Is de alternatieve methode efficient?
   (e) Wanneer is een parallel algoritme *optimaal*? Is de alternatieve methode optimaal?

2. **Synchronisatiehulpmiddelen.** Een thread0 gebruikt een waarde `B`, die initieel `undef` is en wordt berekend door thread1:

```
Thread0:                Thread1:
  ..;                     ..;
  s = s+B;                B = pp+qq;
  ..;                     ..;
```

   (a) Thread0 mag de waarde van `B` *niet* gebruiken voordat die door Thread1 is uitgerekend en ingevuld. Welke statement(s) zou je hiervoor aan de code toevoegen?
   (b) Waarom wordt het gebruik van een `wait` statement binnen een loop van een Monitor, niet gezien als spinnen?
   (c) De berekening en het gebruik van `B` zijn nu onderdeel van een herhaling:

```
Thread0:                Thread1:
  s = 0;                  while (..)
  while (..)                { pp = "bereken iets";
    { ..                      qq = "bereken iets";
      s = s+B;                B = pp+qq;
    }                       }
```

   Elke nieuwe waarde van `B` moet exact eenmaal worden bijgeteld bij `s`. Wat kun je toevoegen aan de code om dat voor elkaar te krijgen?

3. **De atomaire Copy.** Een *CopyDog* heeft als toestandsruimte een rij van waarden, en naast operaties `write(i,x)` (schrijf `x` in locatie `i`) en `read(i)` (geef waarde van locatie `i`), een operatie `copy(i,j)` die de waarde van locatie `i` kopieert naar `j`. Een CopyDog met twee locaties kan 2-Consensus implementeren.

   (a) In de `propose(x)` operatie zal thread `i` (0 of 1) precies eenmaal `Arb.copy(i,1-i)` uitvoeren. Hoe kun je de initiële waarde van `Arb` kiezen om ervoor te zorgen dat alleen de *eerste* `copy`-operatie iets aan de inhoud verandert?
   (b) Hoe kan een thread na het uitvoeren van zijn `copy` zien, welke thread als eerste de `copy` deed? (Uitleg!)
   (c) Geef een wachtvrij algoritme dat een CopyDog gebruikt en 2-Consensus implementeert.
   (d) Kan een CopyDog wachtvrij worden geïmplementeerd met registers?
   (e) Wat is het Consensusgetal van een CopyDog?