UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i	INF3380 — Parallellprogrammering for naturvitenskapelige problemer
Eksamensdag:	14. juni 2016
Tid for eksamen:	9.00 – 13.00
Oppgavesettet er på 5 sider.	
Vedlegg:	Ingen
Tillatte hjelpemidler:	Kalkulator Ett to-sidig A4 ark med håndskrevne notater

Kontroller at oppgavesettet er komplett før
du begynner å besvare spørsmålene.

Vekting av oppgavene

Oppgave 1: 10%
Oppgave 2: 20%
Oppgave 3: 25%
Oppgave 4: 20%
Oppgave 5: 25%

Oppgave 1 (vekt 10%)

Det er p prosessorer som har fått tildelt arbeid som er representert henholdsvis av $W_1, W_2, \ldots, W_p$. Vi kan da beregne *load imbalance ratio* som følgende:

$$\frac{\max_i(W_i) - \min_i(W_i)}{\min_i(W_i)}$$

Hvis en 100×100 matrise er partisjonert som p rektangulære blokker så jevnt som mulig, hvilke verdier skal load imbalance ratio henholdsvis være for følgende tre prosessor-grid: 8×1, 4×2, og 3×3?

Oppgave 2 (vekt 20%)

Figur 1 (på side 2) viser en "task dependency graph" som involverer seks oppgaver: $t_1, t_2, \ldots, t_6$. Tallene i parentsen representerer tidsbruk av oppgavene. Hver kant i grafen går fra en "start"-node til en "finish"-node, og tallet som tilhører hver kant representerer tidsbruk av kommunikasjonen mellom sin "start"-node og sin"finish"-node.

(Fortsettes på side 2.)

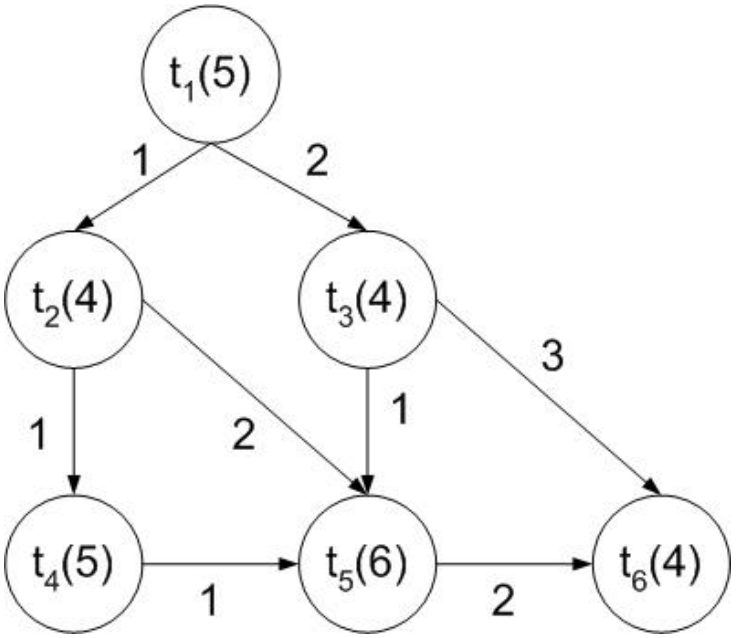

Figure 1: A task dependency graph

Vi antar at hver oppgave kan bare utføres av én arbeider. Videre antar vi at kommunikasjonen mellom et "start"-"finish" par vil bli utført *automatisk* (uten manuelle inngrep) så snart som arbeidet på "start"-noden er ferdig. Merk: Dersom en oppgave har en eller flere innkommende kommunikasjonskanter, kan oppgaven ikke starte før alle sine innkommende kommunikasjoner er ferdig utført.

2a (vekt 10%)

Hvis det bare er én arbeider, minimum hvor mye tid trengs det for å bli ferdig med alle oppgavene? Vennligst forklar svaret ditt.

2b (vekt 5%)

Hvis det er to arbeidere, minimum hvor mye tid trengs det for å bli ferdig med alle oppgavene? Vennligst forklar svaret ditt.

2c (vekt 5%)

Hvis det er tre arbeidere, minimum hvor mye tid trengs det for å bli ferdig med alle oppgavene? Vennligst forklar svaret ditt.

Oppgave 3 (vekt 25%)

Anta det er p prosessorer som har hver sin data. Hensikten med en *gather* operasjon er å la én prosessor få tak i alle dataene fra de andre

(Fortsettes på side 3.)

$p-1$ prosessorene.

3a (vekt 10%)

Hvis det er 16 prosessorer som er koblet sammen via en ring, vennligst forklar steg for steg hvordan en gather operasjon kan mest effektivt realiseres som en serie av en-til-en kommunikasjoner. (Du kan anta at det er prosessor nummer 0 som skal samle alle dataene.)

3b (vekt 10%)

Anta at følgende modell for tidsbruk gjelder for å sende en melding med m ord fra en prosessor til en annen:

$$t_s + t_w m,$$

hvor t_s og t_w er to konstanter. Utledd tidsbruk-modellen for utførelse av en gather operasjon på en ring av p prosessorer. Hver prosessor initielt har m ord som sin data. Du kan anta at p er en potens av 2.

3c (vekt 5%)

Hvordan vil tidsbruk-modellen endre seg dersom p ikke er en potens av 2? Vennligst diskuter to spesifikke situasjoner hvor vi henholdsvis har $p = 7$ og $p = 12$.

Oppgave 4 (vekt 20%)

4a (vekt 10%)

Følgende pseudo-kode beskriver den såkalte bubble-sort algoritmen, som skal sortere en liste med n tall: a_1, a_2, ..., a_n.

```
1.    procedure BUBBLE_SORT(n)
2.    begin
3.        for i := n − 1 downto 1 do
4.            for j := 1 to i do
5.                compare-exchange(a_j, a_{j+1});
6.    end BUBBLE_SORT
```

Vennligst skriv en seriell C funksjon

```
void serial_bubble_sort (int n, int* a);
```

som implementerer pseudo-koden. Vennligst også sørg for at løkken med `i`-indeksen kan avslutte tidligere dersom det er mulig. Du kan anta at n er et partall.

(Fortsettes på side 4.)

4b (vekt 10%)

Følgende pseudo-kode beskriver metoden som heter odd-even transposition. Den tillater parallell prosessering:

```
1.    procedure ODD-EVEN(n)
2.    begin
3.        for i := 1 to n do
4.        begin
5.            if i is odd then
6.                for j := 0 to n/2 − 1 do
7.                    compare-exchange(a_{2j+1}, a_{2j+2});
8.            if i is even then
9.                for j := 1 to n/2 − 1 do
10.                   compare-exchange(a_{2j}, a_{2j+1});
11.       end for
12.   end ODD-EVEN
```

Vennligst skriv en OpenMP-parallelisert C funksjon

```
void para_oddeven_sort (int n, int* a);
```

som implementerer pseudo-koden. Vennligst også sørg for at løkken med `i`-indeksen kan avslutte tidligere dersom det er mulig. Du kan anta at n er et partall.

Oppgave 5 (vekt 25%)

5a (vekt 10%)

Vi skal parallelisere matrise-vektor multiplikasjon $y = Ax$ ved hjelp av MPI programmering, hvor A er en $n \times n$ matrise, mens både x og y er en vektor av lengde n. Vennligst implementer følgende C funksjon som skal bli kalt av hver MPI prosess:

```
void para_matvec (int n, double** A, double* x, double* y);
```

Du kan anta at antall MPI prosesser, p, går opp i n. Videre kan du anta at matrisen A allerede er partisjonert på forhånd, ved hjelp av "row-wise block 1D partitioning". Det vil si, 2D-array `A` som input innholder den matchende delen av A for hver MPI prosess. (`A` har n/p som sin første dimensjon og n som sin andre dimensjon.) Når det gjelder 1D-array `x` som input, er det kun hos prosess 0 hvor hele x-vektoren er lagret i `x`-arrayen, mens de andre MPI prosessene har dummy verdier i sin `x`-array som input til `para_matvec`. Når funksjonen returnerer, skal hele y-vektoren være lagret kun i `y`-arrayen som tilhører prosesses 0, mens de andre MPI prosessene har dummy verdier i sin `y`-array som output.

(Fortsettes på side 5.)

5b (vekt 10%)

Vennligst utledd en teoretisk modell for den parallelle tidsbruken av `para_matvec`. (Du kan bruke konstantene t_s and t_w som var definert i Oppgave 3b. Videre kan du innføre en ny konstant t_c som representerer tidsbruken av hver operasjon av multiplikasjon eller addisjon.)

5c (vekt 5%)

Vennligst utfør den assosierte isoefficiency-analysen.

Appendix: The syntax of some commonly used MPI functions

```
int MPI_Comm_size( MPI_Comm comm, int *size )

int MPI_Comm_rank( MPI_Comm comm, int *rank )

int MPI_Barrier( MPI_Comm comm )

int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm )

int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)

int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)

int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```