

i Nytt dokument

Eksamen i INF3380 våren 2018

Hjelpemidler: Ett to-sidig A4 ark med håndskrevne notater pluss en kalkulator. Ingen andre hjelpemiddel er tillatt.

Alle oppgavene besvares med hjelp av tastatur og mus, det er ikke behov for å bruke digital håndtegning/skisseark.

Vekting av oppgavene:

Oppgaver 2.1, 2.2 (Processing inhomogeneous, independent tasks): 10%

Oppgaver 3.1, 3.2 (Processing homogeneous, dependent tasks): 10%

Oppgaver 4.1, 4.2 (All-to-all broadcast): 15%

Oppgaver 5.1, 5.2, 5.3 (OpenMP): 30%




Oppgaver 6.1, 6.2, 6.3 (Sorting): 35%

**2.1 Processing inhomogeneous, independent tasks;
two workers**

Skriv ditt svar her...

Maks poeng: 5

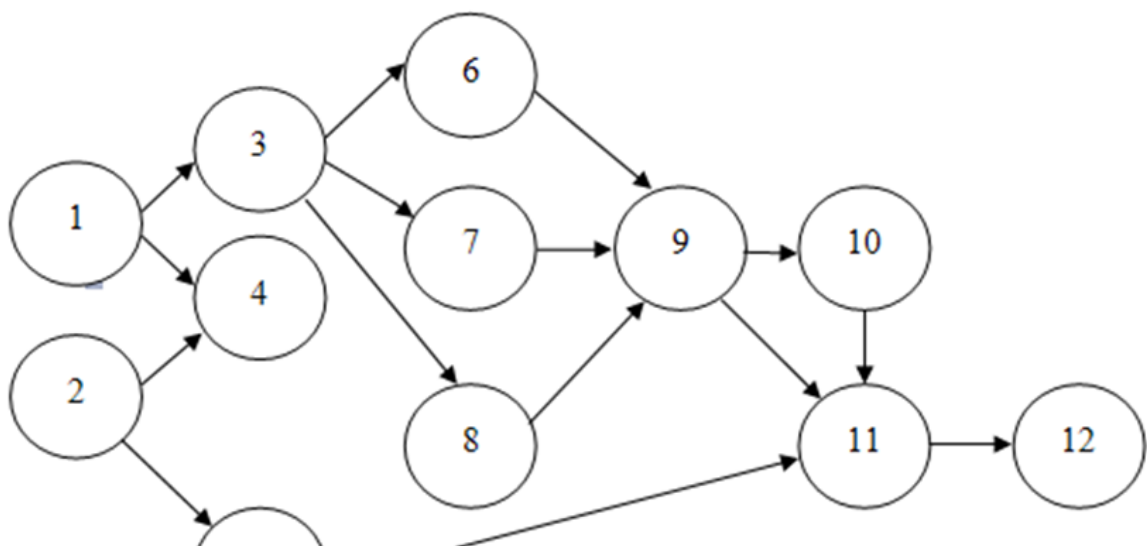
Skriv ditt svar her...

 |  | 

Words: 0

Maks poeng: 5

3.1 **Processing homogeneous, dependent tasks; two workers**



Her har vi en task-dependency graf som beskriver avhengighetsrelasjonen mellom 12 arbeidsoppgaver. (Oppavene er nummerert med tallene.) Alle oppgavene krever samme tidsbruk. Mer spesifikk trenger en arbeider 1 time å utføre hver oppgave. En oppgave kan kun utføres av én arbeider. Hver pil i grafen indikerer en avhengighetsrelasjon mellom en "destinasjonsoppgave" og en "kildeoppgave". Det betyr at en oppgave som er blitt pekt på ikke kan starte før alle sine kildeoppgaver er ferdig utført.

Hvis det er 2 arbeidere, hvor mange timer minst trenger de å fullføre alle de 12 arbeidsoppgavene? Begrunn svaret ditt.

Skriv ditt svar her...




Words: 0

Maks poeng: 5

3.2 Processing homogeneous, dependent tasks; three workers

Vi fortsetter med samme task-dependency grafen. Hvis det nå er tre arbeidere, hvor lang tid minimum trengs det? Begrunn svaret ditt.

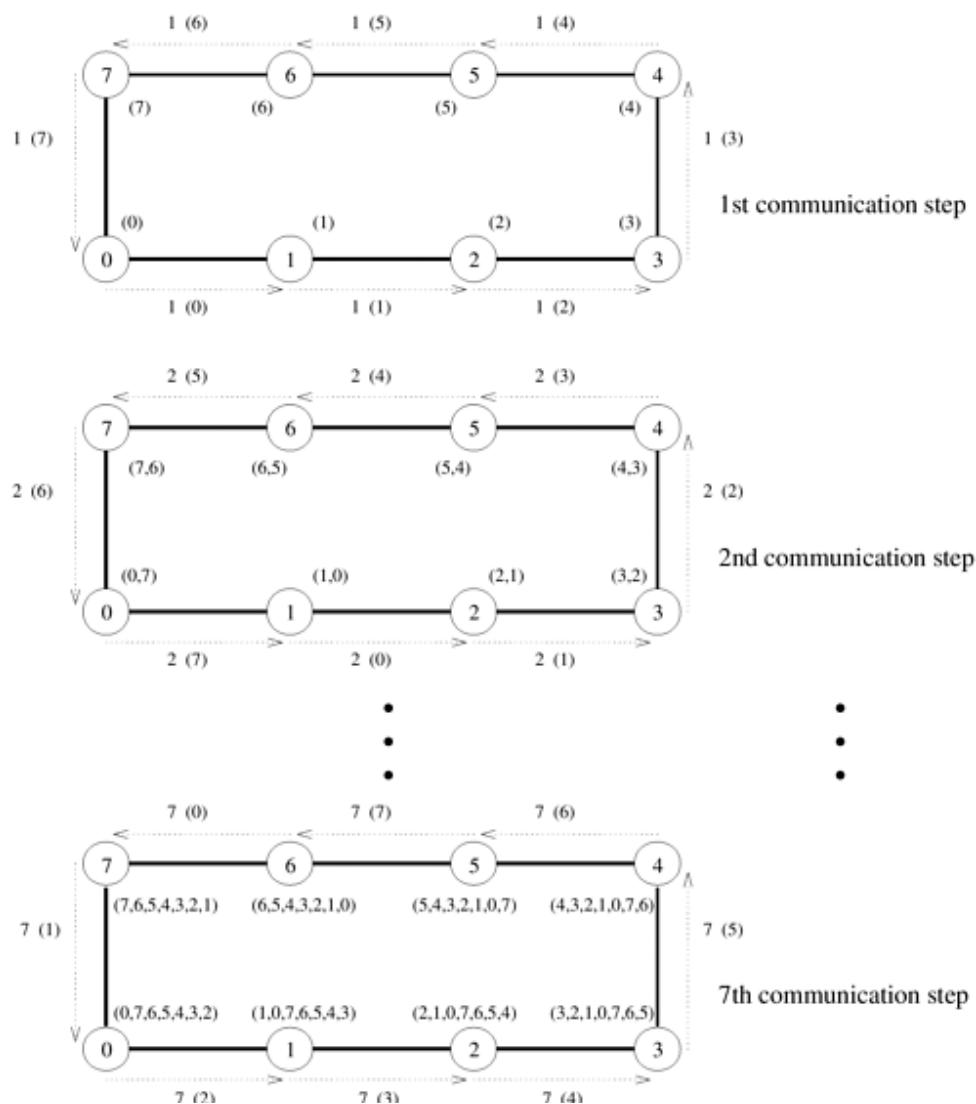
Skriv ditt svar her...

Format	▼				↺			
								

Maks poeng: 5

4.1 All-to-all broadcast on a 2D torus

Her har vi et eksempel av hvordan en all-to-all broadcast kan utføres, steg for steg, på en 1D ring som består av 8 noder.



Vennligst utvid ideen til en 2D torus, som er en 2D mesh av noder med "wraparound". (Et eksempel av 2D torus finner du nede.) Forklar hvordan en all-to-all broadcast kan utføres mest effektivt på en 2D torus med R rader og C søyler.

Skriv ditt svar her...

Format ▾

↺

↻

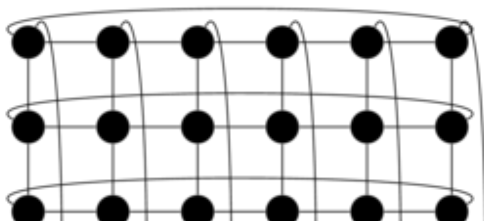
✖

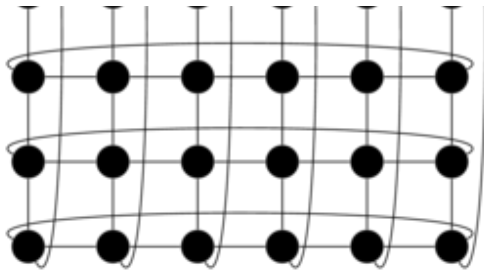
✎

Σ

✖

Words: 0





Maks poeng: 8

4.2 All-to-all broadcast on a 2D torus; performance model

Vi antar at følgende formel



$$t_s + t_w m$$

kan brukes til å beregne tidsbruken for å sende en melding av m bytes fra en node til en annen, hvor t_s og t_w er kjente konstantverdier. Utled en formel som beskriver total tidsbruk til en all-to-all broadcast som utføres på en 2D torus bestående av R rader og C søyler. (Initielt har hver node m bytes som egen data.)

Skriv ditt svar her...

Format ▾

↺

 | Σ | 

Maks poeng: 7

5.1 Private variable

Hvorfor trenger man private variabler i OpenMP kode? Gi et eksempel av bruk av en privat variabel.

Skriv ditt svar her...

Maks poeng: 10

5.2 Show the running result

Dersom følgende kodesnutt er kjørt med 4 OpenMP tråder. Hva vil utskriften bli?

```
int total_sum = 0;
int i;
#pragma omp parallel default(shared) reduction(+:total_sum)
{
    int my_id = omp_get_thread_num();
    int my_sum = 0;
    #pragma omp for schedule(static,10)
    for (i=1; i<=100; i++)
        my_sum += i;
    printf("From thread No.%d: my_sum=%d\n", my_id, my_sum);
    total_sum += my_sum;
}
printf("Total sum=%d\n",total_sum);
```

Skriv ditt svar her...

Maks poeng: 10

5.3 Correcting error(s)

Det er en eller flere feil i følgende OpenMP kodesnutt. Rett feilen(e).

```
int i;
```

```

double u[1000], v[1000];
for (i=0; i<1000; i++) {
    u[i] = 0.001*(i-500);
    v[i] = 0.0;
}

#pragma omp parallel default(shared)
{
    int time_step;
    double *tmp;

    for (time_step=0; time_step<100; time_step++)
    {
        #pragma omp for nowait
        for (i=1; i<999; i++)
            v[i] = u[i-1]-2*u[i]+u[i+1];

        tmp = v;
        v=u;
        u=tmp;
    }
}

```

Skriv ditt svar her...

Maks poeng: 10

6.1 Merging two sorted sublists

Følgende funksjon har til hensikt å flette sammen to sublister for å produsere en ny liste. (Begge input-sublistene er antatt til å allerede være sortert og er av lengde m . Som resultat skal output-listen være sortert og av lengde $2m$.) Den nåværende implementasjonen mangler litt på slutten. Skriv ferdig implementasjonen.

```
void merge (int m, const int *sorted_sublist1, const int *sorted_sublist2, int *merged_list)
{
    int i,j,k;
    i = 0;
    j = 0;
    k = 0;

    while (i<m && j<m)
    {
        if (sorted_sublist1[i] <=sorted_sublist2[j])
        {
            merged_list[k] = sorted_sublist1[i];
            i++;
        }
        else
```

```

    {
        merged_list[k] = sorted_sublist2[j];
        j++;
    }

    k++;
}

/* The remaining part of this function is missing, please complete.
   .....
*/
}

```

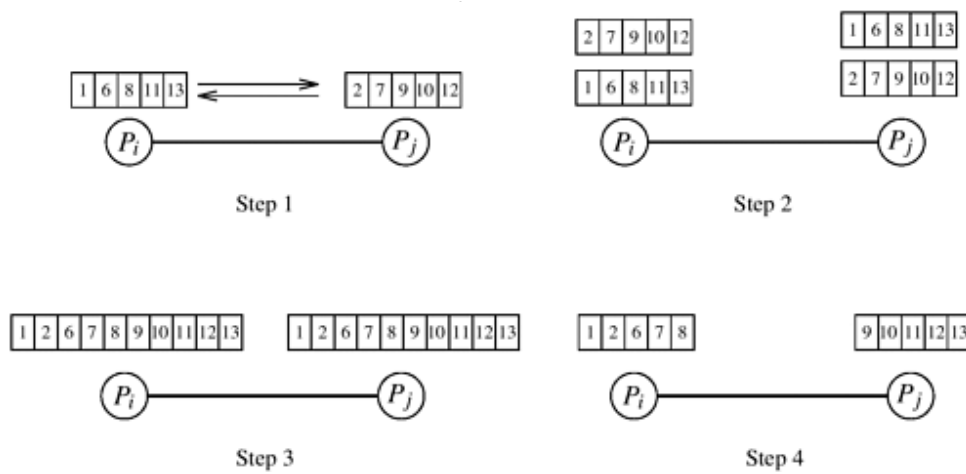
Skriv ditt svar her...

1	
---	--

Maks poeng: 10

6.2 Compare-split

Den såkalte compare-split operasjonen, som involverer to parallelle prosesser, er en viktig byggekloss for å implementere en parallell sorteringsalgoritme. Som utgangspunkt for en compare-split operasjon er det to prosesser som har hver sin subliste som allerede er sortert. Et konkret eksempel av compare-split er skissert i følgende figur. (We antar også at Rank P_i er lavere enn Rank P_j .)



Implementer compare-split operasjonen som en funksjon:

```
void compare_split (int m, int *my_sublist, int my_MPI_rank, int other_MPI_rank)
```

ved å bruke passende MPI kommando(er) og å kalle merge-funksjonen fra forrige oppgave.

Hint: Funksjonen "compare_split" er ment for å bli kalt av to MPI prosesser samtidig i samarbeid, som i utgangspunkt har hver sin sorterte subliste.

Skriv ditt svar her...

Syntaks for noen av de viktigste MPI funksjoner:

```
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)
```

6.3 Block version of parallel odd-even transposition

Parallel Odd-Even Transposition

```
1.  procedure ODD-EVEN_PAR(n)
2.  begin
3.      id := process's label
4.      for i := 1 to n do
5.          begin
6.              if i is odd then
7.                  if id is odd then
8.                      compare-exchange_min(id + 1);
9.                  else
10.                     compare-exchange_max(id - 1);
11.              if i is even then
12.                  if id is even then
13.                      compare-exchange_min(id + 1);
14.                  else
15.                     compare-exchange_max(id - 1);
16.          end for
17.  end ODD-EVEN_PAR
```

Parallel formulation of odd-even transposition.

Figuren ovenfor skisserer en pseudokode for å utføre såkalte parallell odd-even transposition, som har til hensikt å sortere en liste med tall. Denne pseudokoden har antatt at antall prosesser er lik listelengden.

Du er nå bedt om å skrive en C funksjon som er basert på samme ideen av parallell odd-even transposition. Forskjellen er at du nå skal tillatte listelengden n til å være lik $P \cdot m$, altså, n er et multiplum av antall MPI prosesser P . Dette betyr at hver prosess initielt er tildelt m tall som danner sin subliste.

Implementer følgende C funksjon:

```
void odd_even_block_parallel (int m, int *sublist)
```

Merk: Denne funksjonen skal bli kalt samtidig av alle MPI prosessene. Du kan anta at "sublist", som er av lengden m , allerede er sortert på hver MPI prosess før funksjonen "odd_even_block_parallel" er kalt.

Hint: Du skal erstatte alle "compare_exchange_min" og "compare_exchange_max" med compare_split funksjonen fra forrige oppgave, på en passende måte.

Skriv ditt svar her...

--	--

Maks poeng: 10