Tentamen Parallel Computing

Donderdag 3 juli 2008, 9:00-12:00 uur, Examenhal

docenten: R. de Bruin, A. Meijster

- Lees eerst een opgave volledig door alvorens deze te maken.
- Geef geen antwoorden zonder toelichting.
- Het tentamen bestaat uit 4 opgaven. Alle opgaven zijn evenveel punten waard.
- Succes.

Opgave 1: Architectuur

(a) Wat is het verschil tussen de *theoretical peak bandwidth* en de *bisection bandwidth* van een netwerk?

(b) De onderstaande figuren geven een aantal netwerktopologiën weer. Een vierkant representeert een processor en een lijn een verbinding.

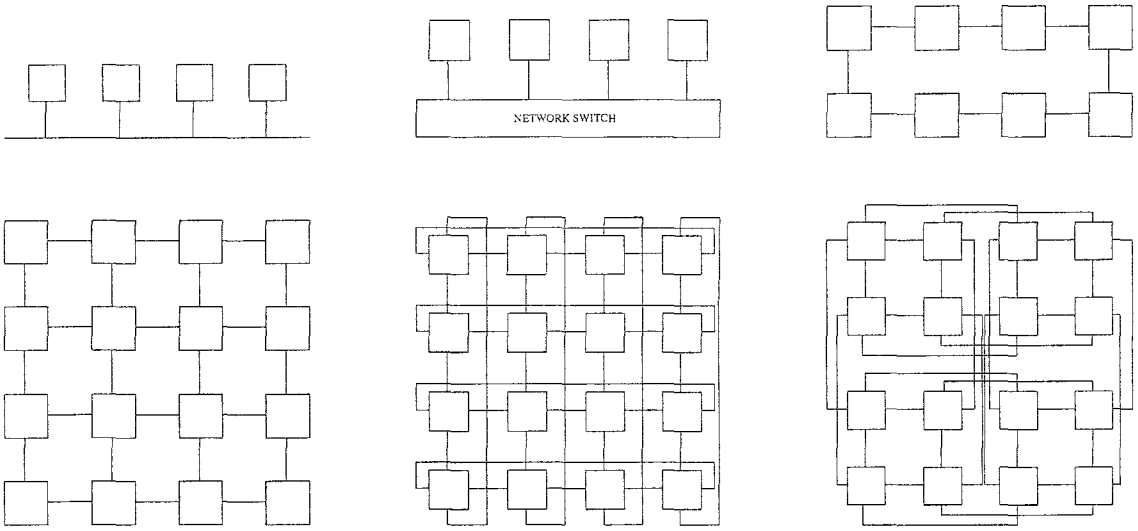

Figure 1: netwerken: (a) bus, (b) geswitched netwerk, (c) ring, (d) grid, (e) torus, (f) hypercube.

Geef van iedere toplogie aan wat de orde van de *bisection bandwidth* is. Laat n het aantal processoren zijn. Kies voor ieder netwerk uit:

(A) $O(1)$
(B) $O(\sqrt{n})$
(C) $O(n)$
(D) $O(n \log n)$
(E) $O(n^2)$

(c) De onderstaande twee programmafragmenten bepalen de som van de elementen in een groot array van doubles.

```
for (i=0; i<n; i++) {
  sum = a[i] + sum;
}
```

```
for (i=0; i<n; i+=8) {
  s0 += a[i+0]; s4 += a[i+4];
  s1 += a[i+1]; s5 += a[i+5];
  s2 += a[i+2]; s6 += a[i+6];
  s3 += a[i+3]; s7 += a[i+7];
}
sum = s0+s1+s2+s3+s4+s5+s6+s7;
```

Voor zekere `n` duurt op een vector-processor met een vectorregister ter lengte 8 het linker algoritme ongeveer 8000 klokcycli, terwijl het rechter algoritme ongeveer 1000 klokcycli duurt. Verklaar het verschil.

(d) De onderstaande programmafragmenten voeren de matrix-vermenigvuldiging `C=A*B` uit. Ga uit van grote `n` (bijvoorbeeld: `n=500`). Veronderstel dat initieel de matrix `C` overal nul is en dat de matrices `A` en `B` gegeven zijn. In de praktijk blijkt op een scalaire machine met een cache dat

```
1 for (i=0; i<n; i++) {
2   for (j=0; j<n; j++) {
3     for (k=0; k<n; k++) {
4       C[i][j] += A[i][k] * B[k][j];
5     }
6   }
7 }
```

```
1 for (k=0; k<n; k++) {
2   for (i=0; i<n; i++) {
3     for (j=0; j<n; j++)
4       C[i][j] += A[i][k] * B[k][j];
5     }
6   }
7 }
```

```
1 for (j=0; j<n; j++) {
2   for (k=0; k<n; k++) {
3     for (i=0; i<n; i++)
4       C[i][j] += A[i][k] * B[k][j];
5     }
6   }
7 }
```

Figure 2: Matrix-matrix vermenigvuldiging: lus-volgordes (a) ijk, (b) kij, (c) jki.

de drie algoritmen aanzienlijk verschillen in snelheid (let op: we veronderstellen dat de compiler geen optimalisaties verricht!). Welke algoritme is het snelste en welke is het langzaamste? Leg uit waarom.

(e) Een programma bestaat uit een deel dat parallel uitgevoerd kan worden, en een deel dat sequentieel is. Het sequentiele deel van de executietijd op één processor bedraagt 10%. Wat is de maximale speedup die verkregen kan worden door gebruik te maken van:

1. twee processoren
2. tien processoren
3. onbeperkt aantal processoren.

Opgave 2: OpenMP
Het onderstaande programma bestaat uit twee threads. De threads delen de *shared variable* x. Je mag veronderstellen dat iedere toekenning atomair is.

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
  int x = 1;
  #pragma omp parallel sections shared(x)
  {
    #pragma omp section
    {
      int y = x+1;
      x = y;
    }
    #pragma omp section
    {
      int y = 2*(x+1);
      x = y;
    }
  }
  printf ("x=%d\n", x);
  return EXIT_SUCCESS;
}
```

(a) Leg uit wat het betekent dat de toekenningen atomair zijn.

(b) Twee mogelijke uitkomsten zijn `x=5` of `x=6`. Welke uitkomst(en) is(zijn) nog meer mogelijk? Geef voor iedere uitkomst (ook voor `x=5` en `x=6`) aan in welke volgorde de toekenningen zijn uitgevoerd.

(c) Geef aan hoe je m.b.v. OpenMP er voor kunt zorgen dat alleen de uitkomsten `x=5` of `x=6` voor kunnen komen. Je mag expliciet de juiste OpenMP-commando's geven, maar het is ook toegestaan om de oplossing te beschrijven (de syntax is dus onbelangrijk).

(d) Het onderstaande programmafragment implementeert een deel (de voorwaartse pass) van het Gaussische eliminatie algoritme voor het oplossen van een stelsel lineaire vergelijkingen. Geef aan hoe je deze code met OpenMP kan paralleliseren.

```
1   double s, a[N][N];
2   int i, j, k;
3   for (i=0; i<N; i++) {
4     for (j=i+1; j<N; j++) {
5       s = a[j][i] / a[i][i];
6       for (k=i; k<N; k++) {
7         a[j][k] = a[j][k] - s*a[i][k];
8       }
9     }
10  }
```

Opgave 3: Pthreads
Het onderstaande programma simuleert twee threads die (vijf maal) om de beurten respectievelijk `ping` en `pong` afdrukken.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <pthread.h>
 4
 5 volatile int turn=0;
 6
 7 void *thread(void *arg) {
 8   int i, whoami=(int)arg;
 9   for (i=0; i<5; i++) {
10     while (turn != whoami);
11     printf (whoami == 0 ? "ping\n" : "pong\n");
12     turn = 1 - turn;
13   }
14   return NULL;
15 }
16
17 int main (int argc, char **argv) {
18   pthread_t th0, th1;
19
20   pthread_create(&th0, NULL, thread, (void *)0);
21   pthread_create(&th1, NULL, thread, (void *)1);
22
23   pthread_join(th0, NULL);
24   pthread_join(th1, NULL);
25   return EXIT_SUCCESS;
26 }
```

(a) Wat betekent *volatile* in regel 5? Waarom is dit nodig?

(b) Het programma maakt gebruik van busy-waiting in regel 10. Wat is busy-waiting en waarom is dit in het algemeen niet een goede techniek?

(c) De prototypes van de functies `pthread_cond_wait` en `pthread_cond_signal` zijn:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

Leg uit wat deze functies doen. Leg met name uit waar de mutex toe dient.

(d) Verander het programma zodanig dat er niet langer gebruik wordt gemaakt van busy-waiting. Maak gebruik van mutexen en conditie-variabelen. Merk op dat de exacte syntax van de pthread-calls niet van belang is (bijv. `lock(m)` i.p.v. `pthread_mutex_lock(&m)` is voldoende).

Opgave 4: MPI

(a) Wat wordt bedoeld met *Master-Slave computing* en wat is het voordeel van deze techniek?

(b) De MPI-routines `MPI_Send` en `MPI_Recv` hebben de volgende protoypes:

```
int MPI_Send(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);
```

Wat is de functie van de parameter `tag`?

Het onderstaande MPI-programma fragment heeft de structuur van een Master-Slave programma. Het Master-proces (proces met rank 0) dient als een eenvoudige server die een array van 1024 integers administreert. Dit array wordt gebruikt om virtueel een stukje gemeenschappelijk (shared) memory te simuleren. Client processen (slave processen) kunnen elementen van het array veranderen of opvragen door middel van de operaties `init()`, `store()` en `load()`.

- `store(a, b)` zal in het array op plaats (index) `a` de waarde `b` plaatsen.
- `init(a, b)` is een speciaal geval van `store`. Voor een index `a` gedraagt deze routine zich bij eerste aanroep als `store`. Bij iedere volgende aanroep met dezelfde index `a` wordt de operatie genegeerd (skip-operatie). `load(a, &b)` kopiert uit het array de waarde op plaats (index) `a` in `b`.
- `stop()` Een proces dat deze routine aanroept geeft aan niet langer gebruik te willen maken van het gesimuleerde shared memory.

(c) waarom is het nodig om in de `server` gebruik te maken van wildcards voor de source-rank en de message-tag (regels 20-21)?

(d) Schrijf de bodies van de routines `init()`, `store()`, `load()` en `stop()`.

(e) Leg uit hoe je het pingpong-algoritme (regels 7-15 van opgave 3) met behulp van het gesimuleerde gemeenschappelijke array zou kunnen implementeren (je mag dit in woorden uitleggen en/of de betreffende code schrijven). Waarom is *busy-waiting* op een cluster meestal geen bezwaar?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4
5 #define STOPTAG   0
6 #define INITTAG   1
7 #define STORETAG  2
8 #define LOADTAG   3
9
```

```
11 void server() { /* MPI-rank of server is 0 */
12   int np, idx, src, tag;
13   int msg[2];
14   int data[1024], init[1024];
15   char str[64];
16   MPI_Status status;
17   for (idx=0; idx<1024; idx++) init[1024]=0;
18   /* get number of processes in the process group */
19   MPI_Comm_size(MPI_COMM_WORLD, &np);
20   while (np > 1) {
21     MPI_Recv (&msg, 2, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
22               MPI_COMM_WORLD, &status);
23     tag = status.MPI_TAG;
24     src = status.MPI_SOURCE;
25     idx = msg[0];
26     switch(tag) {
27     case STOPTAG:
28       np--;
29       break;
30     case INITTAG:
31       if (init[idx] == 0) {
32         data[idx] = msg[1];
33         init[idx]=1;
34       }
35       break;
36     case STORETAG:
37       data[idx] = msg[1];
38       break;
39     case LOADTAG:
40       MPI_Send(&data[idx], 1, MPI_INT, src, 99, MPI_COMM_WORLD);
41       break;
42     }
43   }
44
45 void init(int idx, int data) {
46     ....
47 }
48
49 void store(int idx, int data) {
50   ....
51 }
52
53 void load(int idx, int *data) {
54   ....
55 }
56
57 void stop() {
58   ....
59 }
```