Újrakonfigurálható technológiák nagy teljesítményű alkalmazásai

OpenMP, MPI

Szántó Péter

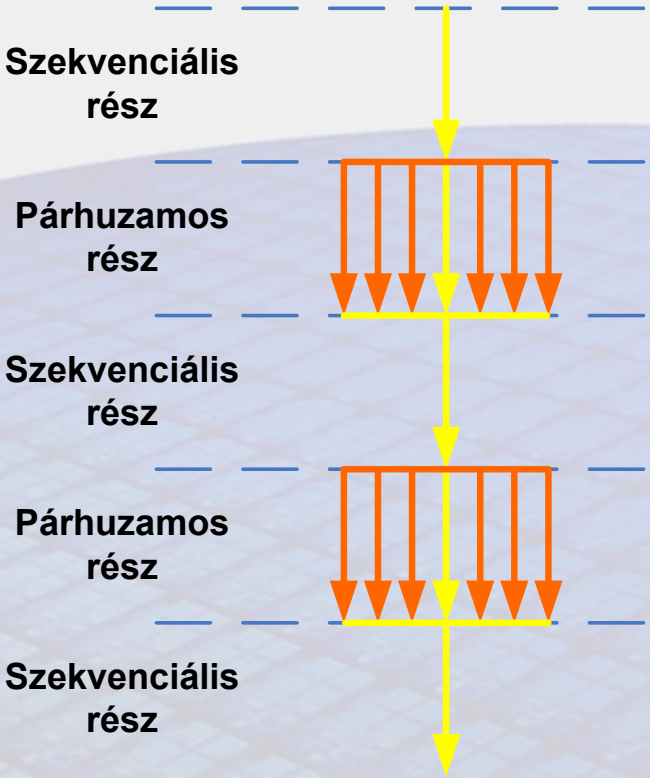
BME MIT, FPGA Laboratórium

Párhuzamos programozás

- **Szálkezelés könyvtárakkal**
  - Win32 API, POSIX Pthread library, Boost, ….
- **Fordító direktívák**
  - Osztott memória modell
  - OpenMP (pragma gyűjtemény)
- **Üzenetküldés alapú megoldások**
  - Az adatmegosztás explicit (a programozó feladata)
  - MPI: Message Passing Interface
  - PVM: Parallel Virtual Machine

OpenMP modell

- SPMD: Single Program Multiple Data
- Osztott memória: minden szál hozzáfér
- „Fork – join” modell



Nyelvi elemek

- **Párhuzamos programrészek**
  - #pragma omp parallel
- **Worksharing**
  - #pragma omp for, #pragma omp sections
- **Adatkörnyezet**
  - #pragma omp parallel shared/private (…)
- **Szinkronizáció**
  - #pragma omp barrier
- **Futási idejű, ill. környezeti változók**
  - int my_thread_id = omp_get_num_threads();
  - omp_set_num_threads(…);

Párhuzamos blokk

- **A létrehozott szálak számát az OMP_NUM_THREADS környezeti változó (vagy explicit megadás) adja meg**
- **Minden szál a párhuzamos blokkon belüli kódot hajtja végre**
- **NINCS szinkronizálás a szálak között**
  - Nem detirminisztikus hogy a szálak ugyanazt a kódrészletet mikor érik el
- **Miután minden szál végrehajtotta a párhuzamos blokkot a master szál kivételével megszűnnek**

Párhuzamos blokk

- **A #pragma hatóköre egy strukturális blokk: {……}**
- **Egyedi szál azonosító**
  - Különböző kód is végrehajtható

```
……
#pragma omp parallel
{
   myid = omp_get_thread_num();
   if (myid==0)
       master_function();
   else
       thread_function();
}
……
```

Párhuzamos blokk - végrehajtás

- **Dinamikus mód (alapértelmezett)**
  - A létrehozott szálak száma dinamikusan változik
  - A beállított szám felső limit
- **Statikus mód**
  - A szálak száma a programozó által beállított
  - #pragma omp parallel for num_threads(4)
- **A fordító létrehozhat szekvenciális kódot**
- **Végrehajtási mód beállítása**
  - OMP_DYNAMIC környezeti változó
  - omp_set_dynamic() függvény

Párhuzamos blokk paraméterek

- **shared(var1,var2,…)**
  - Szálak között megosztott változó
- **private(var1,var2,…)**
  - Minden szálnak saját példánya van a változóból
- **firstprivate(var1,var2,…)**
  - Inicializálás a párhuzamos blokkba lépés előtt, private változókba átmásolódik az inicializálási érték
- **lastprivate(var1,var2,…)**
  - Megtartja értékét a párhuzamos blokk után. Azaz a legutolsó iteráció értéke átmásolódik a szekvenciális (master) blokk változójába
- **if(expression)**
  - Feltételes párhuzamosítás
- **default(shared|private|none)**
  - Változók alaptípusa
- **schedule(type [,chunk])**
  - Ciklus iterációk ütemezése az egyes szálak között
- **reduction(operator|intrinsic:var1,var2…)**
  - Redukciós operátor/változók megadása

Ciklus párhuzamosítás

- **FOR ciklus többszálú végrehajtása**

```
int i;
float a[N],b[N],r[N];
#pragma omp parallel
{
   #pragma omp for
   for (i=0; i<N; i++){
       res[i]=a[i]*b[i];
   }
}
```

```
int i;
float a[N],b[N],r[N];
#pragma omp parallel for
for (i=0; i<N; i++){
   res[i]=a[i]*b[i];
}
```

- **Szinkronizáció**
  - Ciklus mag vége (kivéve: #pragma omp for nowait)
  - Teljes ciklus végrehajtása

Ütemezés

```
for (i=0; i<1100; i++) { res[i]=a[i]*b[i] }
```

- #pragma omp parallel for schedule (static, 250)

250	250	250	250	100
t0	t1	t2	t3	t0

- #pragma omp parallel for schedule (dynamic, 200)

200	200	200	200	200	100
t3	t1	t2	t0	t1	t3

- #pragma omp parallel for schedule (guided, 100)

137	120	105	100	100	100	100	100	100	100	38
t0	t3	t0	t1	t2	t3	t0	t1	t2	t3	t0

- #pragma omp parallel for schedule (auto)

Critical

- Egyszerre egy szál hajtja végre

```
sum = 0;
#pragma omp parallel private (lsum)
{
   lsum = 0;
   #pragma omp for
   for (i=0; i<N; i++){
      lsum = lsum + A[i];
   }
   #pragma omp critical{
      sum += lsum;
   }
}
```

Redukciós műveletek

- Redukciós operátorok biztonságos végrehajtása
- Pl. MAC:

```
#pragma omp parallel for        \
  default(shared) private(i)    \
  schedule(static,chunk)        \
  reduction(+:result)
  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
```

Párhuzamos függvények

- Függvények párhuzamos végrehajtása

```
omp_set_num_threads(THREAD_NUM);
#pragma omp parallel
{
    int thid = omp_get_thread_num();
    function0( …… );
    function1( …… );
    function2( …… );
}
```

Sections

```
#pragma omp parallel default(none) shared(val,thread){
#pragma omp sections
{
    #pragma omp section
    {
        for(int i=0;i<3;++i)
            val[i]=1<<i;
    }
    #pragma omp section
    {
        for(int i=3;i<6;++i)
            val[i]=1<<i;
    }
}
```

1. szál hajtja végre

2. szál hajtja végre

Szinkronizáció

- **#pragma omp master {…….}**
  - Csak a master szál hajtja végre
- **#pragma omp critical {…….}**
  - Egy időben csak egy szál hajtja végre
- **#pragma omp barrier**
  - Szálak bevárják egymást
- **Egyéb**
  - ATOMIC
  - ORDERED – a ciklus iterációk végrehajtási sorrendje ugyanaz, mint szekvenciális esetben
  - FLUSH – cache kezelés

MPI

- **Üzenetküldés API elosztott memóriás rendszerekhez**
- **Nagy teljesítmény**
- **Jól skálázható, portolható**
- **C, C++, Fortran**
- **Az adatcsere explicit**
- **MPI-1**
  - Az adatcserére fókuszál
- **MPI-2**
  - Párhuzamos I/O
  - Dinamikus processz menedzsment
  - Távoli memoria műveletek
  - >500 eljárás

Legfontosabb függvények

- **MPI_Init( )**
  - MPI futatókörnyezet inicializálása
- **MPI_Comm_size ( )**
  - Visszaadja a résztvevő processzek számát
- **MPI_Comm_Rank( )**
  - A hívó rank-ja (~processz azonosító)
- **MPI_Send( )**
  - Egyszerű üzenetküldés
- **MPI_Recv**
  - Egyszerű üzenet fogadás
- **MPI_Finalize( )**
  - Befejezés

MPI „Hello world!”

- MPI_COMM_WORLD: az összes processzt tartalmazó kommunikátor

```
#include<stdio.h>
#include"mpi.h"
int main(intargc,char**argv)
{
   int rank, size;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   printf("Hello world!I am %d of %d\n",rank,size);
   MPI_Finalize();
   return 0;
}
```

Üzenetküldés

- **MPI_SEND(start, count, datatype, dest, tag, comm)**
  - Buffer kezdőcím
  - Az elküldendő elemek száma
  - Adattípus (pl. MPI_INTEGER)
  - Címzett (rank)
  - Üzenet azonosító (0…32767)
  - Communicator (pl. MPI_COMM_WORLD)

MPI példa (1)

```
#include<stdio.h>
#include<mpi.h>
int main(intargc,char**argv)
{
   int i,rank,size,dest;
   int to,src,from,count,tag;
   int st_count,st_source,st_tag;
   double data[100];
   MPI_Status status;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   printf("Process %d of %d is alive\n",rank,size);
   dest=size-1;
   src=0;
   ……
```

MPI példa (2)

```
if(rank==src)
{
   to=dest;
   count=100;
   tag=2001;
   for(i=0;i<100;i++)
      data[i]=i;
   MPI_Send(data,count,MPI_DOUBLE,to,tag,
            MPI_COMM_WORLD);
}
elseif(rank==dest)
{
   tag=MPI_ANY_TAG;
   count=100;
   from=MPI_ANY_SOURCE;
   MPI_Recv(data,count,MPI_DOUBLE,from,tag,
            MPI_COMM_WORLD,&status);
```

MPI példa (3)

```
    MPI_Get_count(&status,MPI_DOUBLE,&st_count);
    st_source=status.MPI_SOURCE;
    st_tag=status.MPI_TAG;
    printf("Status info: source=%d, tag=%d,
       count=%d\n", st_source, st_tag, st_count);
    printf("%d received:",rank);
    for(i=0;i<st_count;i++)
       printf("%lf",data[i]);
    printf("\n");
    }
    MPI_Finalize();
    return0;
}
```

- **MPI_Bcast( )**
  - Egy processz üzenetét továbbítja az összes többinek
- **MPI_Reduce( )**
  - Minden processz végrehajtja a bemeneti bufferén a redukciós műveletet (pl. max), majd visszaadja az értéket a „root” processz kimeneti bufferében
- **Blokkoló/nem blokkoló üzenetküldés (MPI_Isend)**
  - MPI_Wait, MPI_Test