

Párhuzamos és Grid rendszerek (7. ea)

szálak, openMP

Szeberényi Imre
BME IIT

<szebi@iit.bme.hu>



Áttekintés

- Eddig általános eszközöket láttunk, melyek SMP és Cluster környezetben is használhatók.
- SMP/NUMA környezet egyre gyakoribb a hétköznapiakban:
 - többmagos, többszál processzorok
 - 2, 4, 6, 8, ... mag
- Újabb architektúrák, ClearSpeed, GPGPU
- Újabb programozási modellek (CL, CUDA)

Szálak

- A processzek memóriaterületei egymástól teljesen elkülönítettek.
- A szálak olyan processzek, melyeknek csak a stack területe különül el.
- Így lényegesen gyorsabban lehet váltani közöttük, valamint könnyebben is kommunikálnak egymással.
 - pthread_create(), pthread_join(), pthread_exit()
 - pthread_mutex_..., pthread_cond_...

pthread példa

```
#include <stdio.h>
#include <pthread.h>
void* do_loop(void *id)
{
    int i, j;
    float f = 0;
    char me = *(char *)id;
    for (i=0; i<10; i++) {
        for (j=0; j<5000000; j++) f++;
        printf("Thread_%c: %d\n", me, i);
    }
    pthread_exit(NULL);
}
```

thread-et megvalósító függvény

thread egyedi adata

pthread példa/2

```
int main(int argc, char* argv[])
{
    pthread_t thread_a, thread_b;
    char a='a', b='b', c='c';

    pthread_create(&thread_a, NULL,
                  do_loop, &a);
    pthread_create(&thread_b, NULL,
                  do_loop, &b);

    do_loop(&c);
    printf("Ide nem jut!\n");
    return 0;
}
```

thread leírója

egyedi paraméter

megvalósító függvény

Kölcsönös kizárás

- mutex:
 - pthread_mutex_init
 - pthread_mutex_destroy
 - pthread_mutex_lock
 - pthread_mutex_trylock
 - pthread_mutex_unlock

pthread_mutex

```
....
pthread_mutex_t mutex_cnt;
....
pthread_mutex_init(&mutex_cnt, NULL);
....
pthread_mutex_lock(&mutex_cnt);

// kritikus régió

pthread_mutex_unlock(&mutex_cnt);
```

Feltétel változó

- condition:
 - pthread_cond_init
 - pthread_cond_destroy
 - pthread_cond_wait
 - pthread_cond_timedwait
 - pthread_cond_signal
 - pthread_cond_broadcast

pthread_cond

```
pthread_mutex_t mutex_cnt;
pthread_cond_t cond_cnt;
pthread_mutex_init(&mutex_cnt, NULL);
pthread_cond_init(&cond_cnt, NULL);
.... // egyik szál
pthread_mutex_lock(&mutex_cnt);
....
// kritikus régió, várni kell valamire
pthread_cond_wait(&cond_cnt,
&mutex_cnt);
....
pthread_mutex_unlock(&mutex_cnt);
```

pthread_cond (2)

```
....
.... // másik szál
pthread_mutex_lock(&mutex_cnt);
....
// a várt esemény bekövetkezett
pthread_cond_signal(&cond_cnt);
....
pthread_mutex_unlock(&mutex_cnt);
....
```

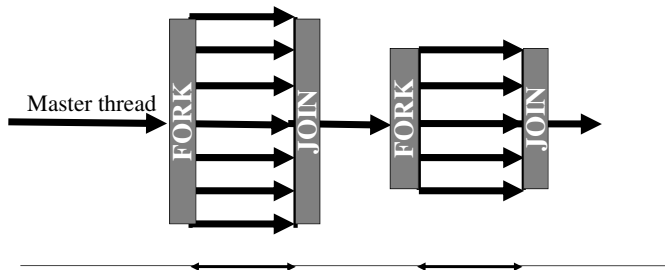
OpenMP motiváció

- Szálakkal történő párhuzamosítás macerás:
- OS függő API-k:
 - Windows: CreateThread
 - UNIX: pthread_create
- Még egy vektor elemeinek összeadásához is sok ismeret kell:
 - Kölcsönös kizárás (mutex)
 - Külön soros és párhuzamos kód keletkezik
 - Nehéz a skálázhatóság megoldása

OpenMP

- Nyelvi kiterjesztés
 - A programozó a funkcionalitásra koncentrálhat.
 - A párhuzamosítás csak lehetőség.
 - Shared memóriás párhuzamosítás
 - Ipari szabvány
 - 1997: 1.0
 - 2011: 3.1
 - 2013: 4.0 – jelenleg draft
- <http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf>

Végrehajtási modell



Shared memoria modell

- A szálak változók keresztül kommunikálnak.
- A megosztás nyelvi szinten definiált
- Versenyhelyzet kialakulhat
 - Szinkronizációs eszközök
 - Megosztás minimalizálása

Szintaxis

- `#pragma omp construct [clause [clause] ...]`
- Egy blokkra vonatkozik (egy belépés, egy kilépés)
- OpenMP konstrukciók:
 - Parallell régiók megadása
 - Munka elosztás (work sharing)
 - Adatelérés szabályozása
 - Szinkronizáció
 - Runtime függvények

Parallell régiók

```
double D[1000] = { 1, 2, 3, 4 };
#pragma omp parallel
{
    int i; double sum = 0;
    for (i=0; i<1000; i++) sum += D[i];
    printf("Thread %d computes %f\n",
           omp_get_thread_num(), sum);
}
// annyiszor fut, ahány thread van.
//OMP_NUM_THREADS
```

The code snippet shows a parallel region. A callout box labeled 'shared' points to the array `D`. Another callout box labeled 'private' points to the loop variable `i` and the local variable `sum`.

Work sharing

```
#pragma omp sections
{
    #pragma omp section
    a = computation_1();
    #pragma omp section
    b = computation_2();
}
c = a + b;
```

Hosszú futási idejű valami

Kézi párhuzamosítás

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nThr = omp_get_num_threads();
    int istart = id*N/nThr, iend = (id+1)*N/nThr;
    for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
```

Automatikus párhuzamosítás

```
#pragma omp parallel
#pragma omp for schedule(static)
{
  for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
}
```

Csak egyszerű for ciklus lehet:

- 1 db int ciklusváltozó,
- cmp. op: <, >, <=, >
- Növelés/csökkentés: ++, --, értékadó op.
- Ciklusfüggetlen init, last, növelés

Párhuzamos for problémái

- Load balancing
 - Egyes részek futási ideje lehet, hogy nem azonos
 - A futási idők eltérése csak ritkán becsülhető előre
 - Dinamikus szétosztás kellene
- Granularitás
 - A szálak létrehozása, szinkronizálása akkor is idő, ha ezt eldugja a fordító.

Ütemezés

- schedule(static [, chunksize])
 - Statikus kiosztás
 - Default: azonos darabok
 - Round-robin (több darab, mint thread esetén)
- schedule(dynamic [, chunksize])
 - Dinamikus kiosztás
 - Default chunksize = 1
- schedule(guided [, chunksize])
 - Dinamikus, exponenciálisan csökken

Granularitás

- #pragma omp parallel if (expression)
 - Párhuzamosítás a feltétellel vezérelhető
- #pragma omp num_threads (expression)
 - A szálak száma módosítható

Adatok elérése

- Osztott változók
 - globális változók
 - int sum = 0;
 - #pragma omp parallel for
 - for (int i=0; i<N; i++) sum += i;
- Privát
 - Párh. blokk auto változói
 - Függvények auto változói
 - Explicit privát deklaráció esetén

Tárolási attribútumok megadása

- private:
 - Privát példány keletkezik, de nem másolódik le az eredeti adat
 - Ugyanaz, mintha {} között lenne
- firstprivate:
 - Kezdő érték lemásolódik
- lastprivate:
 - Legutolsó érték visszamásolódik
- threadprivate:
 - Perzisztens a párh. részek között (globális)

Tárolási attribútumok pl.

```
int i;  
#pragma omp parallel for private(i)  
for (i=0; i<n; i++) { ... }
```

```
int idx=1;  
int x = 10;  
#pragma omp parallel for firstprivate(x) \  
                        lastprivate(idx)  
for (i=0; i<n; i++) {  
    if (data[i]==x) idx = i;  
}
```

Tárolási attribútumok pl./2

```
int data[100];  
#pragma omp threadprivate(data)  
...  
#pragma omp parallel for copyin(data)  
for (int i=0; i<n; i++)  
    data[i]++;
```

Redukció

```
int sum = 0;  
#pragma omp parallel for reduction(+: sum)  
for (int i =0; i<N; j++)  
    sum = sum+a[i]*b[i];
```

- Csak skalár
- Másolat készül, a végén elvégzi a műveletet
- $x \text{ op expr}$
- $x++$, $++x$, $x--$, $--x$,
- op nem lehet túlterhelt

Szinkronizációs mechanizmusok

- Single/Master execution
#pragma omp single
#pragma omp master
- Critical sections, Atomic updates
#pragma omp critical [name]
#pragma omp atomic
update_statement
Csak skalár, a redukciónál megismert formában.

Szinkronizációs mech./2

- Ordered
#pragma omp ordered

```
int vec[100];  
#pragma omp parallel for ordered  
for (int i=0; i<100; i++) {  
    vec[i] = 2 * vec[i] + i;  
    #pragma omp ordered  
    printf("vec[i] = %d\n", vec[i]);  
}
```

Szinkronizációs mech./3

- Barriers
#pragma omp barrier
- Nowait
#pragma omp sections nowait
- Flush
#pragma omp flush (list)
- Reduction

Kontroll funkciók

- `omp_set_dynamic(int)/ omp_get_dynamic()`
- `omp_set_num_threads(int)/ omp_get_num_threads()`
– `OMP_NUM_THREADS` env.
- `omp_get_num_procs()`
- `omp_get_thread_num()`
- `omp_set_nested(int)/omp_get_nested()`
- `omp_in_parallel()`
- `omp_get_wtime()`
- `omp_init_lock(), omp_destroy_lock(),`
- `omp_set_lock(), omp_unset_lock(),`
- `omp_test_lock()`

Példa: integrálás

```
double integ(double low, double high, long n) {
    int num_pes = -1;
    double w, pew, sum = 0;
    int penum;           // private
    double sump, l, h;  // private

    #pragma omp parallel default (none) \
        shared(low, high, n, num_pes, w, pew) \
        private(penum, sump, l, h) \
        reduction(+:sum)
    { // Paralel régió kezdete
        if (num_pes < 0)
            num_pes = omp_get_num_threads();
        penum = omp_get_thread_num(); // private
    }
```

Példa: integrálás /2

```
w = (high - low) / n; // shared
pew = (high - low) / num_pes; // shared
l = pew * penum + low; // private
h = l + pew;           // private
sump = 0.0;
l += w / 2.0;
while (l < h) {
    sump += fx(l);
    l += w;
}
sum += sump;
} // Paralel régió vége
return sum * w;
}
// gcc -fopenmp prog.c -lgomp
```