Wigner GPU Lab

11. fejezet

Párhuzamosítás, Parallel STL Algoritmusok

Haladó Alkalmazott Programozás
ELTE, 2019

- Párhuzamos léptetés a tábla alapú szimulációkban

Párhuzamos Conway életjáték

Tekintsük a Conway életjátékot.

A lépetés során minden cellát frissíteni kell, amelyek függetlenül számolhatóak, így könnyű párhuzamosítani.

Ehhez átírjuk a Table osztály fill2 metódusát.

A számolást vízszintesen fogjuk sávokra osztani a szálak között.

Párhuzamos Conway életjáték

Eredeti kód:

```
template<typename F>
void fill2(F&& f)
{
  for(int i=0; i<w; ++i)
  {
    for(int j=0; j<h; ++j)
    {
      (*this)(i, j) = f(i, j);
    }
  }
}
```

Párhuzamos Conway életjáték

párhuzamosított kód:

```
template<typename F>
void parallel_fill2(F&& f)
{
  static const int n_threads = 2;
  if(h < 100){ fill2(f); return; }
  std::vector<std::future<void>> fs(n_threads);

...
```

Párhuzamos Conway életjáték

párhuzamosított kód:

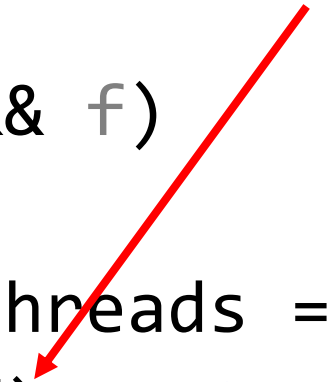
```
template<typename F>
void parallel_fill2(F&& f)
{
  static const int n_threads = 2;
  if(h < 100){ fill2(f); return; }
  std::vector<std::future<void>> fs(n_threads);

...
```

Szálak száma

Párhuzamos Conway életjáték

párhuzamosított kód:

Ha túl kicsi a tábla, nem éri meg párhuzamosítani, ilyenkor a korábbi kódot hívjuk.



```
template<typename F>
void parallel_fill2(F&& f)
{
  static const int n_threads = 2;
  if(h < 100){ fill2(f); return; }
  std::vector<std::future<void>> fs(n_threads);

...
```

Párhuzamos Conway életjáték

párhuzamosított kód:

Egy tömb a future-k számára

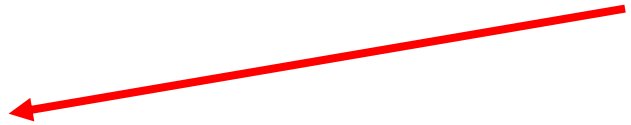
```
template<typename F>
void parallel_fill2(F&& f)
{
  static const int n_threads = 2;
  if(h < 100){ fill2(f); return; }
  std::vector<std::future<void>> fs(n_threads);

...
```

Párhuzamos Conway életjáték


```
template<typename F>
void parallel_fill2(F&& f)
{
...
  int ilow = 0;
  int idelta = h / n_threads;
  int ihi = idelta;
  for(int t=0; t<n_threads; ++t)
  {
    if(t == n_threads-1){ ihi = h; }
    fs[t] = std::async( ... );
    ilow = ihi;
    ihi += idelta;
  }
...
```

Párhuzamos Conway életjáték

```
template<typename F>
void parallel_fill2(F&& f)
{
...
  int ilow = 0;
  int idelta = h / n_threads;
  int ihi = idelta;
  for(int t=0; t<n_threads; ++t)
  {
    if(t == n_threads-1){ ihi = h; }
    fs[t] = std::async( ... );
    ilow = ihi;
    ihi += idelta;
  }
...
```



A függőlegeges méret szerint osztjuk fel a táblát a szálszám szerint.

`ilow` és `ihi` lesz a tartomány alsó és felső indexhatára.

Párhuzamos Conway életjáték

```
template<typename F>
void parallel_fill2(F&& f)
{
...
  int ilow = 0;
  int idelta = h / n_threads;
  int ihi = idelta;
  for(int t=0; t<n_threads; ++t)
  {
    if(t == n_threads-1){ ihi = h; }
    fs[t] = std::async( ... );
    ilow = ihi;
    ihi += idelta;
  }
...
```

A ciklus a szálak számán megy végig

Az utolsó szálnál az index tartomány felső határát a magassághoz igazítjuk.

Léptetjük az indexhatárokat

Párhuzamos Conway életjáték

```
fs[t] = std::async(std::launch::async,
[this, f](int lo, int hi)
{
  for(int i=0; i<w; ++i)
  {
    for(int j=lo; j<hi; ++j)
    {
      (*this)(i, j) = f(i, j);
    }
  }
}, ilow, ihi);
```

Párhuzamos Conway életjáték

```
fs[t] = std::async(std::launch::async,
[this, f](int lo, int hi)
{
  for(int i=0; i<w; ++i)
  {
    for(int j=lo; j<hi; ++j)
    {
      (*this)(i, j) = f(i, j);
    }
  }
}, ilow, ihi);
```

A szálindítás megkapja az alkalmazandó lambdát

És az adott szálra vonatkozó tartomány határokat.

Párhuzamos Conway életjáték


```
fs[t] = std::async(std::launch::async,
[this, f](int lo, int hi)
{
  for(int i=0; i<w; ++i)
  {
    for(int j=lo; j<hi; ++j)
    {
      (*this)(i, j) = f(i, j);
    }
  }
}, ilow, ihi);
```

A belseje lényegében a korábbi fill2, csak szűkített határokkal a belső ciklusban

Párhuzamos Conway életjáték


```
template<typename F>
void parallel_fill2(F&& f)
{
  std::vector<std::future<void>> fs(n_threads);
  ...
  std::for_each(fs.begin(), fs.end(),
                [](auto& fut){ fut.get(); });
}
```

Végül meg kell várnunk, hogy az elindított szálak befejeződjenek.

Párhuzamos Conway életjáték

Példa a léptetési idő skálázásra egy négy magos gépen:

Szálak száma	Átlagos léptetési idő [ms]
1	15.2
2	7.2
3	4.7
4	3.8
5	5.0
6	4.2
7	3.8
8	3.8

Ha sokkal több szálat indítunk, lassan növekszik a futásidő, pl. 64 szálnál már 4.1 ms.

Párhuzamos Conway életjáték

Érdemes megjegyezni, hogy a konkrét programban a szimuláció sebességét a léptetésen felül a kirajzolás is jelentősen befolyásolja, így sokat lehet gyorsítani még a `SoftwareRenderer::forall_pixels` és `plot_by_index` függvényeinek párhuzamosításával.

Párhuzamos Conway életjáték

A kód itt található:

https://github.com/u235axe/miniwnd

main_parallel_game_of_life.cpp

Parallel STL algoritmusok

Berényi Dániel – Nagy-Egri Máté Ferenc

Parallel STL

A C++17-ben a Standard Library a legtöbb algoritmusnak elérhetővé vált egy párhuzamosított változata.

Ezek a függvények abban különböznek a korábbiaktól, hogy első argumentumként egy `ExecutionPolicy` objektumot várnak, ez mondja meg, hogy legyen párhuzamosítva az eljárás.

```
auto sum = std::reduce(std::execution::par, v.begin(), v.end());
```

Parallel STL

`ExecutionPolicy` az `<execution>` headerből

3 féle policy lett eddig szabványosítva a C++17-ben, és 1 majd a 20-ban várható:

- sequenced_policy
- parallel_policy
- parallel_unsequenced_policy
- unsequenced_policy (C++20)

Parallel STL

- sequenced_policy
  Az algoritmus elemi lépései sorban hajtódnak végre nem meghatározott sorrendben, de egymás után.

- parallel_policy
  Az algoritmus elemi lépései párhuzamosan, másik szálakon hajtódhatnak végre, egymáshoz képest nem definiált sorrendben.

- parallel_unsequenced_policy
  Az algoritmus elemi lépései párhuzamosan hajtódhatnak végre, másik szálakon, egymáshoz képest nem definiált sorrendben és a vektorizáció is megengedett.

- unsequenced_policy (C++20)
  Az algoritmus elemi lépései vektorizáltan hajtódnak végre, de egyetlen szálon. A vektorizáció miatt a sorrend nem definiált.

Az utolsó két esetben (unseq) nem lehet lock-ot, mutexet, memória allokációt, vagy más vektorizációt nem támogató műveletet használni az adott eljárásban.

Parallel STL

Példakód: nagy vektor elemeinek négyzetösszege.

Vigyázat!

Az `std::accumulate`-ben adott sorrendben történik a redukció, ez bizonyos optimalizációkat megakadályozhat a párhuzamosítás során.

A megfelelő parallel algoritmus erre célra az `std::transform_reduce`

Ez hasonlóan működik a korábban látott std::inner_product-hoz (de egyetlen tároló az argumentuma), de a redukáló műveletnek asszociatívnak és kommutatívnak kell lennie!

Parallel STL

Példakód: nagy vektor elemeinek négyzetösszege.

```
std::vector<double> v, w;

auto add = [](auto x, auto y){ return x+y; };
auto sq = [](auto x){ return x*x; };

std::transform_reduce(        v.begin(), v.end(), 0.0, add, sq);

auto policy = std::execution::par;

std::transform_reduce(policy, w.begin(), w.end(), 0.0, add, sq);
```

Parallel STL

Példakód: nagy vektor elemeinek négyzetösszege.

```
std::vector<double> v, w;

auto add = [](auto x, auto y){ return x+y; };
auto sq = [](auto x){ return x*x; };

std::transform_reduce(        v.begin(), v.end(), 0.0, add, sq);

auto policy = std::execution::par;

std::transform_reduce(policy, w.begin(), w.end(), 0.0, add, sq);
```

Kezdő érték

Párhuzamos változat

Parallel STL

Példakód: nagy vektor elemeinek négyzetösszege.

Példa futásidőkre 4 magos gépen:

Elemszám	Szekvenciális [us]	Párhuzamos [us]
10 000	13.1	79.9
100 000	127	215
1 000 000	1 355	696
10 000 000	12 720	4 025
100 000 000	135 391	36 662

Parallel STL

Példakód: nagy vektor elemeinek négyzetösszege.

Példa futásidőkre 4 magos gépen:

Kevés elemre a szekvenciális gyorsabb!

Elemszám	Szekvenciális [us]	Párhuzamos [us]
10 000	13.1	79.9
100 000	127	215
1 000 000	1 355	696
10 000 000	12 720	4 025
100 000 000	135 391	36 662

Parallel STL

A párhuzamosítás nem triviális feladat, és attól, hogy egy algoritmusnak van párhuzamos változata, egyáltalán nem biztos, hogy az gyorsabb.

Például az MSVC fejlesztői azt írják (2018 október), hogy az alábbi algoritmusoknál nem párhuzamosítanak, mert az ő tesztjeik szerint egyiknél sem sikerült gyorsulást elérni semmilyen paraméterek esetén:

copy, copy_n, fill, fill_n, move, reverse, reverse_copy, rotate, rotate_copy, swap_ranges

Parallel STL

A Parallel STL jelenleg a Visual Studio 2017-es toolset-től érhető el, illetve a g++ 9-es verziója támogatja (pl. Ubuntu 19.04-en).

Megjegyzés: g++/clang alatt az Intel Thread Building Blocks könyvtárára dependál az implementáció. g++ alatt vannak még nehézségek a használatával.

Házi feladat

Feladat:

1) Mérjük le a léptetési időket abban az esetben, ha függőlegesen osztanánk fel a táblát a Conway szimulációban.

2) Hogyan lehetne átírni az egyenes illesztő kódot úgy, hogy a lehető legkevesebb algoritmus hívást használja és így a párhuzamosítás a leghatékonyabb legyen?

Szorgalmi: Ha van Parallel STL-ünk, hasonlítsuk össze ennek a változatnak a teljesítményét többi változattal.

Szorgalmi feladatok a jobb jegyért

Szorgalmi feladatok:

1) Írjuk át az eredeti házi feladatban megírt egyenesillesztő kódot úgy, hogy a parallel STL-el legyen párhuzamosítva, hasonlítsuk össze a futásidőt az eredetivel.

2) Írjuk át úgy a Conway szimulációt, hogy a szálak egyszer legyenek létrehozva a program elején és amikor számolni kell, legyenek felébresztve, léptessék a tábla adott részét, majd legyenek elaltatva. (ehhez az előző órai párhuzamos primitíveket kell használni)

A helyes működést mindig ellenőrizzük a szekvenciális változattal való összehasonlítással!

Akit bővebben érdekelnek a szimulációk és programozás:

Következő félévben: Grafikus Processzorok Tudományos Célú Programozása

Július 11-12 GPU Nap gpuday.com