Tutorijal: TBB, I deo

Sadržaj

- Uvod u TBB
- Paralelne petlje

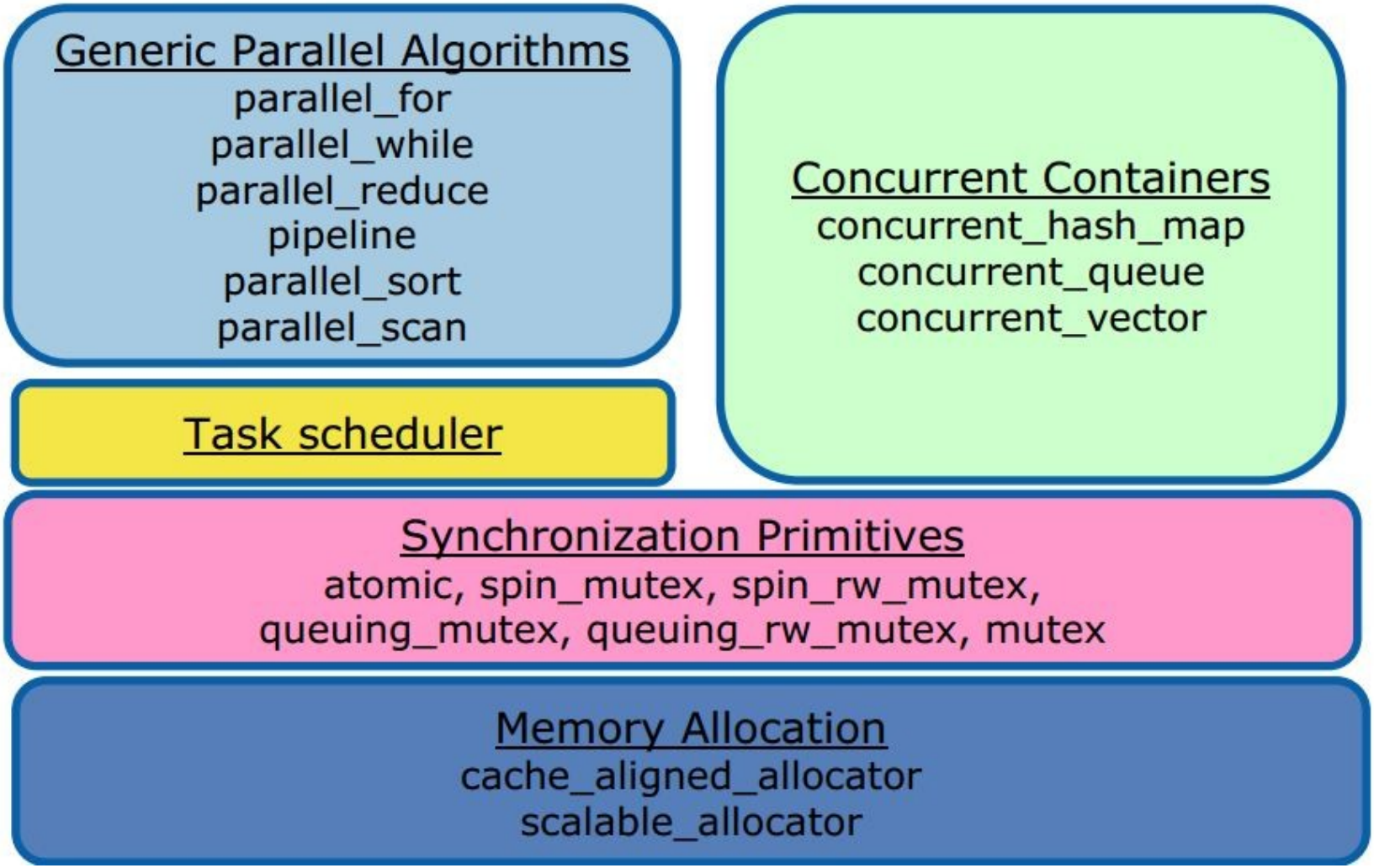
Uvod

- Intel Threading Building Blocks (TBB) je C++ biblioteka koja apstrahuje detalje rada sa nitima.
- Koristi C++ šablone (eng. template).
- U poređenju sa eksplicitnim korišćenjem niti, potrebno je manje linija programskog koda za postizanje paralelizma.
- Programi su portabilni na različite platforme.
- Biblioteka je skalabilna. Pri povećanju broja raspoloživih procesora, ne mora da se piše novi kod.

Ograničenja

- TBB se **ne** preporučuje za:
  - Obradu ograničenu U/I,
  - Obradu u realnom vremenu sa tvrdim ograničenjima (*eng. hard-real time* obradu).
- TBB nije alat koji nužno dovodi do optimalnog rešenja.

Komponente


Generic Parallel Algorithms
parallel_for
parallel_while
parallel_reduce
pipeline
parallel_sort
parallel_scan
Concurrent Containers
concurrent_hash_map
concurrent_queue
concurrent_vector
Task scheduler
Synchronization Primitives
atomic, spin_mutex, spin_rw_mutex,
queuing_mutex, queuing_rw_mutex, mutex
Memory Allocation
cache_aligned_allocator
scalable_allocator

Sekvencijalna `for` petlja

- Pretpostavka: iteracije petlje su međusobno nezavisne.
- Sekvencijalni kod:

```
void SerialApplyFoo(float a[], size_t n) {
  for (size_t i = 0; i!=n; i++)
    Foo(a[i]);
}
```

parallel_for petlja

operator() mora da ima kvalifikator const, kao zaštitu od pokušaja akumuliranja ivičnih efekata, koji bi se izgubili zbog privatnih kopija svake niti

Telo petlje kao objekat funkcija

```
#include "tbb/tbb.h"
#include "tbb/parallel_for.h"
using namespace tbb;

class ApplyFoo {
  float *const my_a;
public:
  void operator()( const blocked_range<size_t>& range ) const {
    float *a = my_a;
    for ( size_t i=range.begin(); i!=range.end(); ++i )
      Foo(a[i]);
  }
  ApplyFoo( float *a ) :  my_a(a) {}
};

void ParalellApplyFoo(float a[], size_t n) {
  parallel_for ( blocked_range<size_t>(0, n),
                 ApplyFoo(a),
                 auto_partitioner());
}
```

Iteracioni prostor

Paralelni algoritam

Naznaka za podelu prostora

`parallel_for` petlja

- Iteracioni prostor ide od 0 do n
- Šablonska funkcija `parallel_for` deli taj prostor na delove nad kojima će se izvršavati operacije u odvojenim nezavisnim nitima
- Prvi korak u paralelizaciji je pretvaranje tela petlje u formu koja operiše nad delom iteracionog prostora (**body object**)
- Body object predstavlja ono šta će izvršavati nad ***svakim*** delom iteracionog prostora, pozivom operatora **()**

`parallel_for` petlja

- `Blocked_range<T>` je template klasa u TBB biblioteci.
- Opisuje jednodimenzionalni iteracioni prostor sa donjom i gornjom granicom
- `Parallel_for` algoritam može da radi sa različitim iteracionim prostorima i zato kroz argumente prihvata parametre `Range` i `Body`, odnosno iteracioni prostor i telo petlje

`parallel_for` petlja

- Pored granica, `blocked_range`-u se prosleđuju i tip elemenata, a opciono se može zadati i parametar `grain_size`
- Ovaj parametar predstavlja granulaciju iteracionog prostora i njega treba varirati kako bi empirijski utvrdili najbolju vrednost za dati računar
- Treba voditi računa da isuviše mala ili velika vrednost mogu drastično da smanje performanse jer dolazi do zasićenja

`parallel_for` petlja

- Npr. ukoliko imamo iteracioni prostor koji ima 10 elemenata i želimo da nad njim operišu tačno 2 niti/2 procesora, onda `grain_size` treba postaviti na vrednost 5
- Ukoliko kod napisan na taj način pustimo na mašini koja ima 4 ili 8 jezgara, maksimalno 2 će biti uposlena
- Kao 3. parametar `parallel_for`-a se može zadati `auto_partitioner()` koji automatski određuje "povoljnu" vrednost za `grain_size`

`parallel_for` petlja

- Zahtevi za body object:
  - Mora postojati konstruktor kopije (***može podrazumevani***) jer svaka nit treba da dobije svoju kopiju objekta
  - Mora postojati destruktor (***može podrazumevani***)
  - Preklapanje **operatora ()** mora biti `const` iz razloga što se mora obezbediti da neka od kopira body objekta ne bi u okviru svog poziva **operatora ()** menjala atribute klase i slično. Na ovaj način se sprečava zavisnost između iteracija

`parallel_for` petlja

- `parallel_for` je moguće “prevariti” korišćenjem globalnih promenljivih ali to nije dobra praksa
- for (i=0; i < n; i++)

```
    a[i] = c;
    if (i == 3)
        c = 5;
```

- Ako je c globalna promenljiva, sve će biti u redu ali će se `parallel_for` struktura narušiti; ako je c polje klase, `const operator()` će sprečiti da se ovakva petlja prevede

Sintaksa `parallel_for` petlje

```
template <typename Range, typename Body>
void parallel_for (const Range& range,
                   const Body& body
                   [,partitioner [, task_group_context]] );
```

- Zahtevi za Body B:
  - B::B(const B&) — *pravljenje kopije*
  - B::~B() — *uništavanje kopije*
  - void B::operator() (Range& subrange) const — *obrađivanje podopsega*

- `parallel_for` dodeljuje podopsege nitima radnika.
- `parallel_for` ne interpretira značenje opsega.

Primer 1: paralelno usrednjavanje

```
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
using namespace tbb;

struct Average {
  float* input;
  float* output;
  void operator()( const blocked_range<int>& range ) const {
    for (int i=range.begin(); i!=range.end(); ++i)
      output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.0f);
  }
};
// Note: The input must be padded such that input[-1] and
// input[n] can be used to calculate the first and last
// output values.
void ParallelAverage( float* output, float* input, size_t n) {
  Average avg;
  avg.input = input;
  avg.output = output;
  parallel_for( blocked_range<int>(0, n, 1000), avg);
}
```

Primer 1: paralelno usrednjavanje

- Program računa srednju vrednost svaka 3 člana ulaznog niza
- `output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.0f);`
- Potrebno je napraviti posebnu klasu ili strukturu čiji preklopljen `operator()` će biti telo `paralel_for` petlje
- Napraviti i posebnu funkciju u kojoj će biti pozvana `paralel_for` petlja sa prosleđenim odgovarajućim parametrima

Reduktori

- Reduktori se primenjuju prilikom primene funkcija kao što su *sum*, *max*, *min* ili logičko i na sve članove nekog niza.
- Operacije moraju biti ***asocijativne***
- Sekvencijalni kod:

```
float SerialSumFoo(float a[], size_t n) {
  float sum = 0;
  for( size_t i=0; i!=n; ++i)
    sum += Foo(a[i]);
  return sum;
}
```

- Ako su iteracije petlje nezavisne, petlja može da se paralelizuje.

parallel_reduce

operator() nije konstantan, pošto privatne kopije tela objekta treba da se spoje u jedno (osvežava SumFoo::sum)

```
class SumFoo {
  float* my_a;
public:
  float sum;
  void operator()( const blocked_range<size_t>& r ) {
    float *a = my_a;
    for (size_t i=r.begin(); i!=r.end(); ++i )
      sum += Foo(a[i]);
    }

  SumFoo ( SumFoo& x, split) : my_a(x.my_a), sum(0){}

  void join( const SumFoo &y ) {sum+=y.sum;}

  SumFoo(float a[]) : my_a(a), sum(0) {}
};
```

Razdvajajući konstruktor se razlikuje od konstruktora kopije pomoću *dummy* argumenta

Metoda *join* se poziva svaki put kada nit završi svoj zadatak i treba da spoji svoj rezultat sa telom osnovnog objekta.

`parallel_reduce`

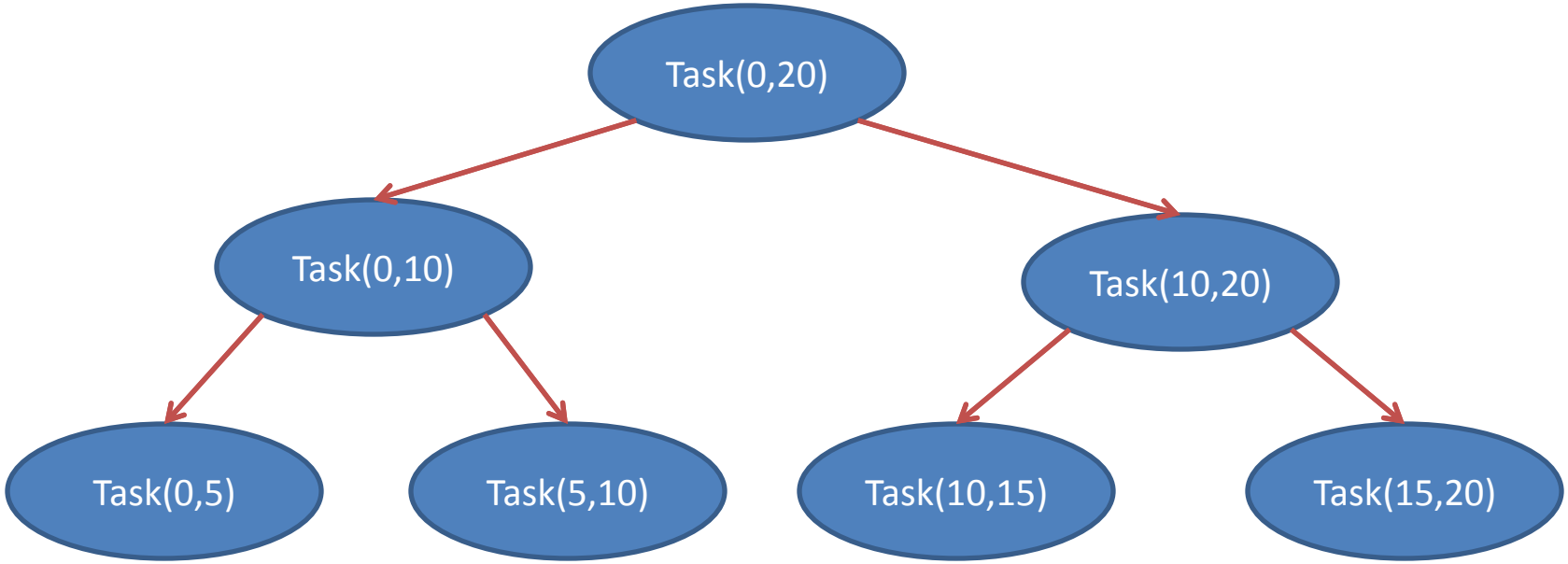
- Razlog zašto `parallel_reduce` treba i mora da osvežava javno polje **sum** je da bi došlo do operacije redukcije i akumuliranja
- Razdvajajuci konstruktor ima parametar **split** koji je dummy argument definisan od strane tbb-a uveden da bi se konstruktor kopije razlikovao od razdvajajuceg konstruktora.
- Razdvajajuci konstruktor prihvata kao parametar reference na originalni objekat a drugi parametar zanemaruje.

`parallel_reduce`

- Kako redukcija funkcionise?
- Kada je neka nit slobodna i dostupna da radi neki posao, poziva se `parallel_reduce` razdvajajući konstruktor u kom se pravi podzadatak koji se dodeljuje toj niti.
- Kada nit završi svoj posao, poziva se metoda **join** kojom `parallel_reduce` akumulira rezultat podzadatka.

parallel_reduce

```
parallel_reduce(blocked_range<int>(0,20,5))
```



Reduktori

- Treba voditi računa da se prilikom paralelne redukcije može doći do drugačijih rezultata zbog zaokruživanja.
- Razlog za to je jer se operacije ne izvršavaju uvek istim redom, kao kod serijskog rešenja, pa zato može doći do greške prilikom zaokruživanja.
- Primer – sabiranje velikog niza u kome imamo i jako velike i jako male vrednosti

Primer 2: pronalaženje indeksa najmanjeg elementa niza

- Petlja pamti trenutno najmanju pronađenu vrednost i njen indeks. To su jedine informacije koje se prenose kroz iteracije petlje.

```
long SerialMinIndexFoo (const float a[], size_t n) {
  float value_of_min = FLT_MAX;   // FLT_MAX from <float.h>
  long index_of_min = -1;
  for( size_t i=0; i<n; ++i ){
    float value = Foo(a[i]);
    if (value < value_of_min) {
      value_of_min = value;
      index_of_min = i;
    }
  }
  return index_of_min;
}
```

parallel_sort

- Uporedni sort sa prosečnim vremenom složenosti O(n log n) na jednoprocesorskom sistemu, gde je n broj elemenata u nizu. Sa povećanjem broja procesora, složenost se smanjuje do O(n).
- Kada su niti radnici raspoloživi, parallel_sort stvara podzadatke, koji mogu da se izvršavaju konkurentno, i na taj način dovodi do smanjenja vremena izvršavanja.
- Sort je deterministički. Sortiranje istog niza će svaki put proizvesti isti rezultat.

Primer 3: `parallel_sort`

- Primer pokazuje dva sorta:
  - Prvi koristi podrazumevano poređenje, koje sortira u rastućem redosledu.
  - Drugi sortira u opadajućem redosledu, koristeći `std::greater<float>`.

```
void SortExample() {
  for (int i = 0; i < N; i++) {
    a[i] = sin((double)i);
    b[i] = cos((double)i);
  }
  parallel_sort(a, a + N);
  parallel_sort(b, b + N, std::greater<float>());
}
```

Merenje vremena u TBB-u

- Vrlo često postoji potreba za merenjem vremena izvršavanja programa ili nekog njegovog dela
- Iako postoje mnogi mehanizmi koji obezbeđuju ovu funkcionalnost, TBB ima svoj
- #include tbb/tick_count.h
- tick_count startTime = tick_count::now();
- (endTime - startTime).seconds()

Dodatni materijali

- Samostalno provežbati priložene primere
- Knjiga Paralelno programiranje
- TBB dokumentacija: vodiči za programiranje i priručnici