Припреме за лабораторијске вежбе из предмета Системска програмска подршка у реалном времену II
- 2017-2018/ Вежба 8 -

Област вежби: *Паралелно програмирање*

ТВВ, ТУТОРИЈАЛ I

СЛОЖЕНЕ ПЕТЉЕ И КОНТЕЈНЕРИ

Садржај

- Увод
- Паралелне петље
- Контејнери

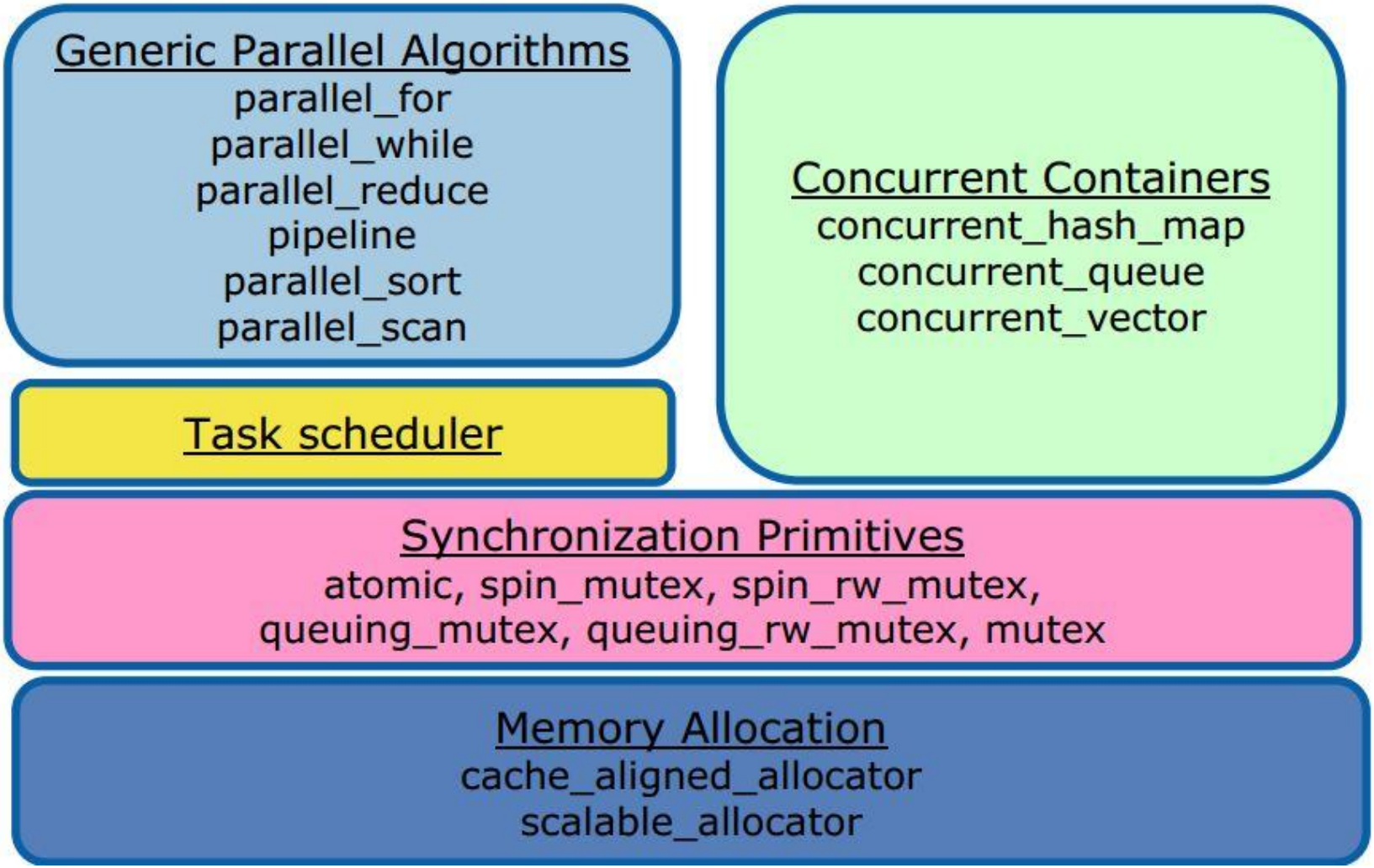
Увод

- Intel Threading Building Blocks (TBB) је C++ библиотека која апстрахује детаље рада са нитима.
- Користи C++ шаблоне (enг. template).
- У поређењу са експлицитним коришћењем нити, потребно је мање линија програмског кода за постизање паралелизма.
- Програми су портабилни на различите платформе.
- Библиотека је скалабилна. При повећању броја расположивих процесора, не мора да се пише нови код.

Ограничења

- ТВВ се **не** препоручује за:
  - Обраду ограничену У/И,
  - Обраду у реалном времену са тврдим ограничењима (енг. *hard-real time* obradu).
- ТВВ није алат који доводи до оптималног решења. Он нас само усмерава у правом смеру.

Компоненте


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

Садржај

- Увод
- Паралелне петље
  - Једноставне петље
  - Сложене петље
- Контејнери

Секвенцијална `for` петља

- Претпоставка: итерације петље су међусобно независне.
- Секвенцијални код:

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

parallel_for петља

operator() мора да има квалификатор const, као заштиту од покушаја акумулирања ивичних ефеката, који би се изгубили због приватних копија сваке нити

Тело петље као објекат функција

```
#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());
}
```

Итерациони простор

Паралелни алгоритам

Назнака за поделу простора

Синтакса `parallel_for` петље

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

- Захтеви за Body B:
  - B::B(const B&) *— прављење копије*
  - B::~B() *— уништавање копије*
  - void B::operator() (Range& subrange) const *— обрађивање подопсега*

- `parallel_for` додељује подопсеге нитима радника.
- `parallel_for` не интерпретира значење опсега.

Пример 1: паралелно усредњавање

```
#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);
}
```

Редуктори

- Редуктори се примењују приликом примене функција као што су *sum*, *max*, *min* или логичко I на све чланове неког низа.
- Секвенцијални код:

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

- Ако су итерације петље независне, петља може да се паралелизује.

parallel_reduce

operator() није константан, пошто приватне копије тела објекта треба да се споје у једно (освежава 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) {}
};
```

Раздвајајући конструктор се разликује од конструктора копије помоћу *dummy* аргумента

Метода *join* се позива сваки пут када нит заврши свој задатак и треба да споји свој резултат са телом основног објекта.

Пример 2: проналажење индекса најмањег елемента низа

- Петља памти тренутно најмању пронађену вредност и њен индекс. То су једине информације које се преносе кроз итерације петље.

```
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;
}
```

- Покренути паралелни пример.

Садржај

- Увод
- Паралелне петље
- Контејнери

Конкурентни контејнери

- Intel TBB обезбеђује контејнере који се могу безбедно конкурентно користити.
  - Конкурентне операције нису безбедне над STL (Standard Template Library) контејнерима.
  - Обично се STL контејнери закључавају искључивим приступом, што смањује паралелизам.
- TBB контејнери имају лошије перформансе од STL ако их користи једна нит, али имају бољу скалабилност.
- Могу да се користе са TBB-ом, OpenMP-ом или обичним (pthread) нитима.

Конкурентни ред `concurrent_queue<T>`

- Задржава локални FIFO поредак.
  - Ако једна нит стави, а друга нит извади из реда две вредности, оне излазе истим редом којим су стављене. Ако више нити стављају и ваде вредности конкурентно, FIFO поредак није загарантован.
- Операције за вађење из реда:
  - Неблокирајућа: `bool try_pop(T&)`
- Уграђена подршка за итерирање кроз ред приликом отклањања грешака (debugging).

Пример 3: `concurrent_queue`

- Пример прави ред са целобројним вредностима 0..9, а потом их исписује на стандардни излаз.

```
int main() {
  concurrent_queue<int> queue;
  for (int i=0; i<10; ++i)
    queue.push(i);
  for (concurrent_queue<int>::const_iterator
i=queue.begin(); i!= queue.end(); ++i)
    cout << *i << " ";
  cout << endl;
  return 0;
}
```

Конкурентни вектор `concurrent_vector<T>`

- Динамички проширириви низ типа T.
  - `grow_by(n)`
  - `grow_to_at_least_(n)`
- Елементи се не померају када се вектор проширује.
- Могућ је конкурентни приступ и проширивање.
  - Методе за брисање и уништавање вектора нису безбедне за конкурентно извршавање са методама за приступ или прошривање.

Конкурентна мапа

`concurrent_hash_map<Key,T,HashCompare>`

- Асоцијативна табела која пресликава кључ *Key* на елемент типа *T*.
- `HashCompare` је класа која одређује како се кључеви праве и упоређују.
- Дозвољава конкурентни приступ за читање и упис:
  - `bool insert(accessor &result, const Key &key)` за додавање или измену,
  - `bool find(accessor &result, const Key &key)` за измену,
  - `bool find(const_accessor &result, const Key &key)` за приступ,
  - `bool erase(const Key &key)` за уклањање.

Пример 4: `concurrent_hash_map`

- Пример прави конкурентну мапу, где су кључеви стрингови, а одговарајући подаци представљају број појављивања сваког од стрингова у низу Data.

```
typedef concurent_hash_map<string,int,MyHashCompare> StringTable;

// Function object for counting occurrences of strings
struct Tally {
  StringTable& table;
  Tally(StringTable& table_) : table(table_) {}
  void operator()(const blocked_range<string*> range) const {
    for(string* p=range.begin(); p!=range.end(); ++p) {
      StringTable::accessor a;
      table.insert(a, *p);
      a->second += 1;
    }
  }
};
```