Parallel and Distributed Computing (B4B36PDV)

Matěj Kafka, Michal Jakob

kafkamat@fel.cvut.cz

https://pdv.pages.fel.cvut.cz

Making parallelism easier

- Last week, we saw how to use multiple threads manually...
- ...but we want to focus on parallelism, not C++ tricks.
- We're focusing on parallel compute-bound programs, isn't there a more specialized tool?

```
size_t i = 0;
auto process = [&] {
    while (i < data.size()) {
        auto& entry = data[i++];
        entry = map_fn(entry);
    }
};

auto worker_threads = std::vector<std::jthread>();
for (size_t j = 0; j < THREAD_COUNT; j++) {
    worker_threads.push_back(std::jthread(process));
}
```

OpenMP

https://www.openmp.org/

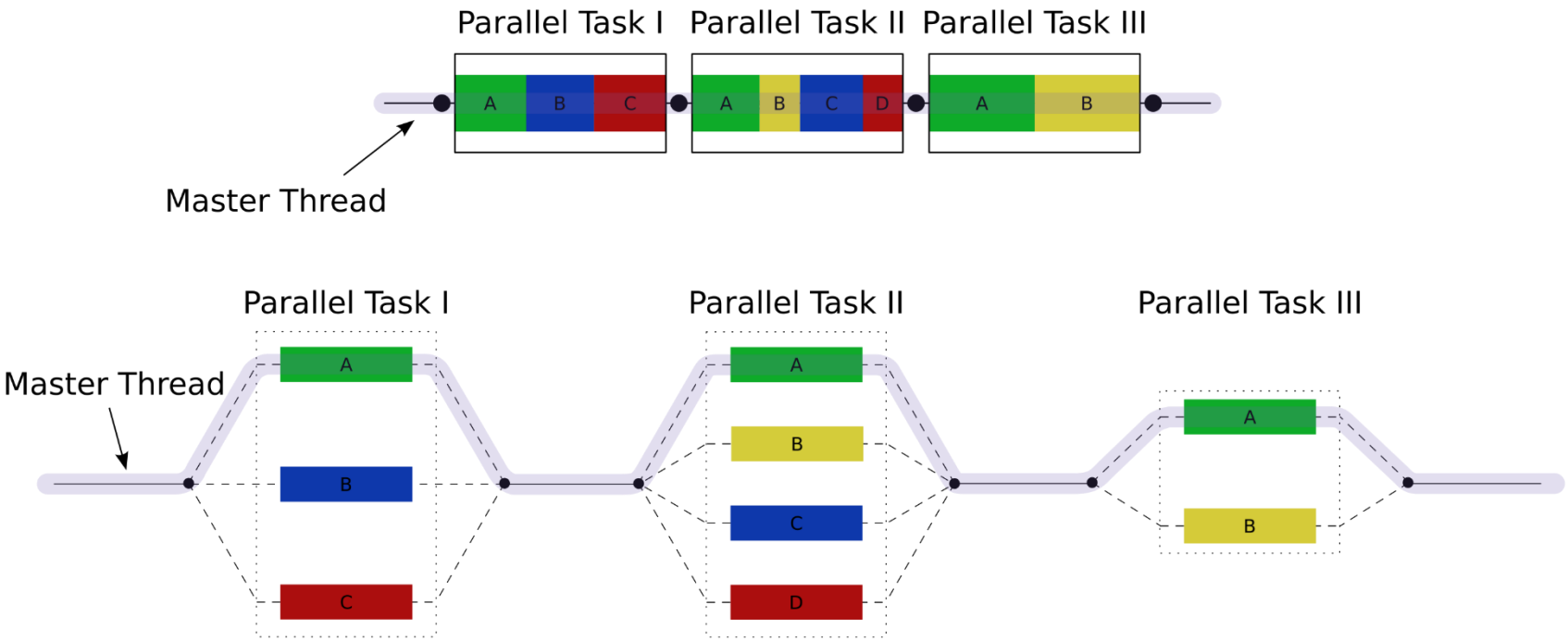
- A compiler extension and a runtime library for parallelization of compute-bound programs.
  - C / C++ / Fortran
- Each major compiler has its own implementation.
- Annotate code with high-level declarations, compiler reduces to low-level runtime library calls.

```
#pragma omp parallel for
for (auto& f : data) {
    f = map_fn(f);
}
```

Coding session 1

INTRODUCTION TO OPENMP

Fork-join model



Source: https://en.wikipedia.org/wiki/Fork%E2%80%93join_model#/media/File:Fork_join.svg

Authored by Branislav Bošanský (in Czech)

Roughly matches the content of the live coding session.

SLIDES FROM 2020

Paralelní a distribuované výpočty (B4B36PDV)

Branislav Bošanský, Michal Jakob

bosansky@fel.cvut.cz

Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

OpenMP – úvod, spuštění

V C++11 (podporuje C,C++)

- **#pragma omp parallel for** je pouze zkratkou za
  - #pragma omp parallel
    {
    #pragma omp for
      for (int i=0; i<MAX; i++)
    }
  - vytvoří tým vláken, které vykonávají blok
  - vezme následující for cyklus a rozdělí jej mezi vlákna v týmu
  - vlákna se připojí k hlavnímu vláknu (join)

```
int main(int argc, char* argv[]) {
#pragma omp parallel
   {
      std::cout << "this is a line printed by thread" << omp_get_thread_num() << std::endl;

#pragma omp for
      for (int i=0; i<10; i++)
         std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;

   }

   std::cout << "Hello from the main thread\n";

   return 0;
}
```

OpenMP – základní způsob práce

Blok parallel

- **#pragma omp parallel**
  - Spustí N vláken, kde N bývá (typicky) počet jader/paralelně zpracovatelných vláken (např. 4 na 2-jádrovém CPU s HT)
  - Pokud chceme počet vláken upravit, použijeme **num_threads(<číslo>)**

```
const int thread_count = 2;
int main(int argc, char* argv[]) {
#pragma omp parallel num_threads(thread_count)
   {
      std::cout << "this is a line printed by thread" << omp_get_thread_num() << std::endl;

#pragma omp for
      for (int i=0; i<10; i++)
         std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;

   }

   std::cout << "Hello from the main thread\n";

   return 0;
}
```

- vypíše se 2x
- pro jedno **i** se vypíše pouze jednou
- každé vlákno vypíše 5

OpenMP – základní způsob práce

Blok for

- Rozdělí iterace na bloky, které vlákna zpracují
  - Není garantované pořadí, ve kterém se jednotlivé iterace provedou
- Existuje několik možností pro úpravu způsobu rozvržení iterací
  - Statické (**static**) – iterace se zařadí do bloků, bloky se přiřadí vláknům (výchozí možnost; bloky jsou přibližně stejně velké)
  - Dynamické **(dynamic)–** iterace se zařadí do bloků (jejich velikost lze ovlivnit, výchozí hodnota je 1), vlákna si vždy vyžádají blok ke zpracování
  - Guided – podobně jak dynamické, ale velikost bloků se postupně zmenšuje
  - Auto – bude zvoleno automaticky
  - Runtime – lze ovlivnit za běhu nastavením proměnné v prostředí
- Pokud chceme, aby se nějaká část cyklu vykonala přesně v pořadí iterací, můžeme použít modifikátor **ordered**
- Pokud máme více vnořených cyklů, můžeme je paralelizovat použitím modifikátoru **collapse(<počet_for_cyklů>)**

OpenMP – základní způsob práce

Blok sections

```
#include <iostream>
#include <vector>
#include "omp.h"

const int thread_count = 2;

void method(const int& i) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    std::cout << "Hello from method " << i << " by thread " << my_rank << std::endl;
}

int main(int argc, char* argv[]) {

#pragma omp parallel  num_threads(thread_count)
    {
        method(1);
#pragma omp sections
        {
#pragma omp section
            {
                method(2);
                method(3);
            }
#pragma omp section
            { method(4); }
        }
    }

    return 0;
}
```

- Můžeme paralelizovat pouze for cykly?
  - Můžeme paralelizovat libovolné metody/úkoly.
  - Použijeme sekce (**sections**)

- vytvoří se tým 2 vláken
- každé vlákno vykoná method(1)
- sekce se rozdělí mezi vlákna v týmu
- každá z method(2)-(4) se vykoná pouze 1x
- method(2) a method(3) se musí vykonat sekvenčně
- 2 sekce mohou být vykonané paralelně

OpenMP – základní způsob práce

Blok tasks

- Můžeme paralelizovat pouze for cykly?
  - Můžeme paralelizovat libovolné metody/úkoly.
  - Použijeme úkoly (**tasks**)

```
const int thread_count = 4;

void Hello() {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   std::cout << "Hello from thread " << my_rank << " of " << thread_count << std::endl;
}

int main(int argc, char* argv[]) {
#pragma omp parallel num_threads(thread_count)
   {
#pragma omp single
      {
         Hello();
#pragma omp task
         Hello();
      }
   }
}
```

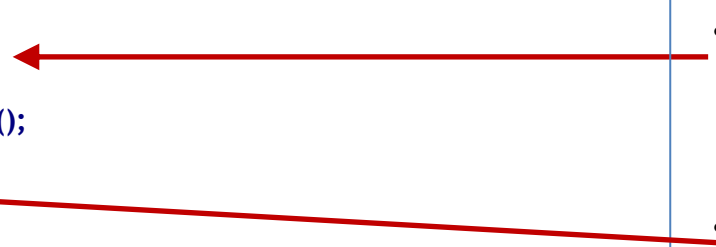
- vytvoří se tým 4 vláken
- pouze 1 vlákno bude vykonávat blok *single*
- (jiné) 1 vlákno bude vykonávat task Hello

OpenMP – základní způsob práce

Synchronizace vláken

- Můžeme vynutit čekání vláken
  - barrier
  - některé bloky (na konci) obsahují implicitní bariéru
    - parallel, sections, for, single
    - pokud chceme implicitní bariéru zrušit, použijeme nowait

```
std::mutex m;
#pragma omp parallel num_threads(2)
   {
#pragma omp single
      {
         m.lock();
#pragma omp task
         {
            m.lock();
            Hello();
            m.unlock();
         }
      }
      Hello();
      m.unlock();
   }
```



- dojde k deadlocku, jelikož druhé vlákno, které chce zmknout **m** čeká na první vlákno, které zámek vlastní
- první vlákno čeká na ukončení druhého vlákna

OpenMP – základní způsob práce

Synchronizace vláken

- Můžeme vynutit čekání vláken
  - barrier
  - některé bloky (na konci) obsahují implicitní bariéru
    - parallel, sections, for, single
    - pokud chceme implicitní bariéru zrušit, použijeme nowait

```
std::mutex m;
#pragma omp parallel num_threads(2)
   {
#pragma omp single nowait
      {
         m.lock();
#pragma omp task
         {
            m.lock();
            Hello();
            m.unlock();
         }
      }
      Hello();
      m.unlock();
   }
```

OpenMP – přístup k proměnným

Privátní a sdílené proměnné

- V rámci bloku omp parallel můžeme určovat viditelnost proměnných
  - shared(<proměnná1>, <proměnná2>,...)
    - sdílené proměnné
  - private(<proměnná1>, <proměnná2>,...)
    - privátní proměnné
    - vytvoří se nenainicializovaná lokální kopie

```
const int thread_count = 2;
int main(int argc, char* argv[]) {
   int a = 42, b = 1;
#pragma omp parallel num_threads(thread_count) shared(a) private(b)
   {
      b = omp_get_thread_num()+2;
#pragma omp critical
      {
         a = a / b;
         std::cout << "Variable 'b' = " << b << " by thread " << (omp_get_thread_num()) << std::endl;
         std::cout << "Variable 'a' = " << a << " by thread " << (omp_get_thread_num()) << std::endl;
      }
   }
   std::cout << "Variable 'a' = " << a << " after omp " << std::endl;
   std::cout << "Variable 'b' = " << b << " after omp " << std::endl;
   return 0;
}
```

OpenMP – přístup k proměnným

Privátní a sdílené proměnné

- V rámci bloku omp parallel můžeme určovat viditelnost proměnných
  - firstprivate(<proměnná1>, <proměnná2>,...)
    - lokální kopie proměnné se nainicializuje dle původní hodnoty

```
const int thread_count = 2;
int main(int argc, char* argv[]) {
   int a = 42, b = 1;
#pragma omp parallel num_threads(thread_count) shared(a) firstprivate(b)
   {
      b = omp_get_thread_num()+b+2;
#pragma omp critical
      {
         a = a / b;
         std::cout << "Variable 'b' = " << b << " by thread " << (omp_get_thread_num()) << std::endl;
         std::cout << "Variable 'a' = " << a << " by thread " << (omp_get_thread_num()) << std::endl;
      }
   }
   std::cout << "Variable 'a' = " << a << " after omp " << std::endl;
   std::cout << "Variable 'b' = " << b << " after omp " << std::endl;
   return 0;
}
```

OpenMP – přístup k proměnným

Privátní a sdílené proměnné

- pro agregaci výsledků lze použít **reduction variable**
  - **#pragma omp parallel reduction(<operace>:<proměnná>)**
  - přístup ke sdílené proměnné
  - zabezpečí atomickou exekuci, nicméně pouze jedné předem definované operace
  - vhodné pro agregaci parciálních výsledků dílčích úkolů

```
long x=0;
#pragma omp parallel for num_threads(thread_count) reduction(+:x)
        for (int i=0; i<SIZE; i++) {
            x += vector_to_sum[i];
        }
```

  - vytvoří lokální kopii proměnné
  - na konci použije definovanou operaci pro sjednocení parciálních výsledků ze všech vláken

OpenMP – přístup k proměnným

Privátní a sdílené proměnné

- pro agregaci výsledků lze použít **reduction variable**
  - co když chceme agregovat výsledky v poli (případně v jiné datové struktuře)
  - můžeme si definovat vlastní operace redukce
    - #pragma omp declare reduction(name:type:expression) initializer(expression)
    - např. se může hodit agregace (suma) čísel ve vektoru

```
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
            std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \
            initializer(omp_priv = omp_orig)
```

OpenMP – přístup k proměnným

Privátní a sdílené proměnné

- pro agregaci výsledků lze použít **reduction variable**
  - co když chceme agregovat výsledky v poli (případně v jiné datové struktuře)
  - můžeme si definovat vlastní operace redukce
    - #pragma omp declare reduction(name:type:expression) initializer(expression)
    - např. se může hodit agregace (suma) čísel ve vektoru

```
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
        std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \
        initializer(omp_priv = omp_orig)
```