Paralelní a distribuované výpočty (B4B36PDV)

Jakub Mareček, Michal Jakob

jakub.marecek@fel.cvut.cz

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

S využitím materiálů Brana Bošanského: https://cw.fel.cvut.cz/b192/courses/b4b36pdv/lectures/start

Dnešní přednáška

Další nástroje

Dnešní přednáška

OpenMP

2015: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

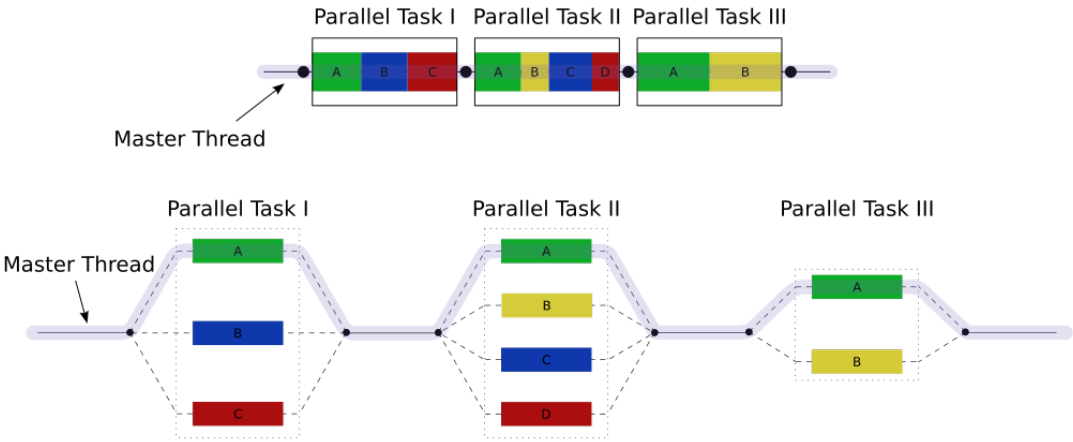
2020: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf

2022: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

OpenMP – úvod

Koncept a záruky

- OpenMP je běžný standard pro programování počítačů se sdílenou pamětí.
- OpenMP používá tzv. sériově-paralelní model uspořádání vláken (fork-join model of parallel execution).
- OpenMP: začneme z jednoho vláka (initial/master thread). Z jednoho vlákna můžeme spustit více vláken, které odpovídají paralelizovatelné úloze (task region). Pak můžeme čekat (join) na dokončení vláken pro tuto paralelizovatelnou úlohu vytvořených.



OpenMP – úvod

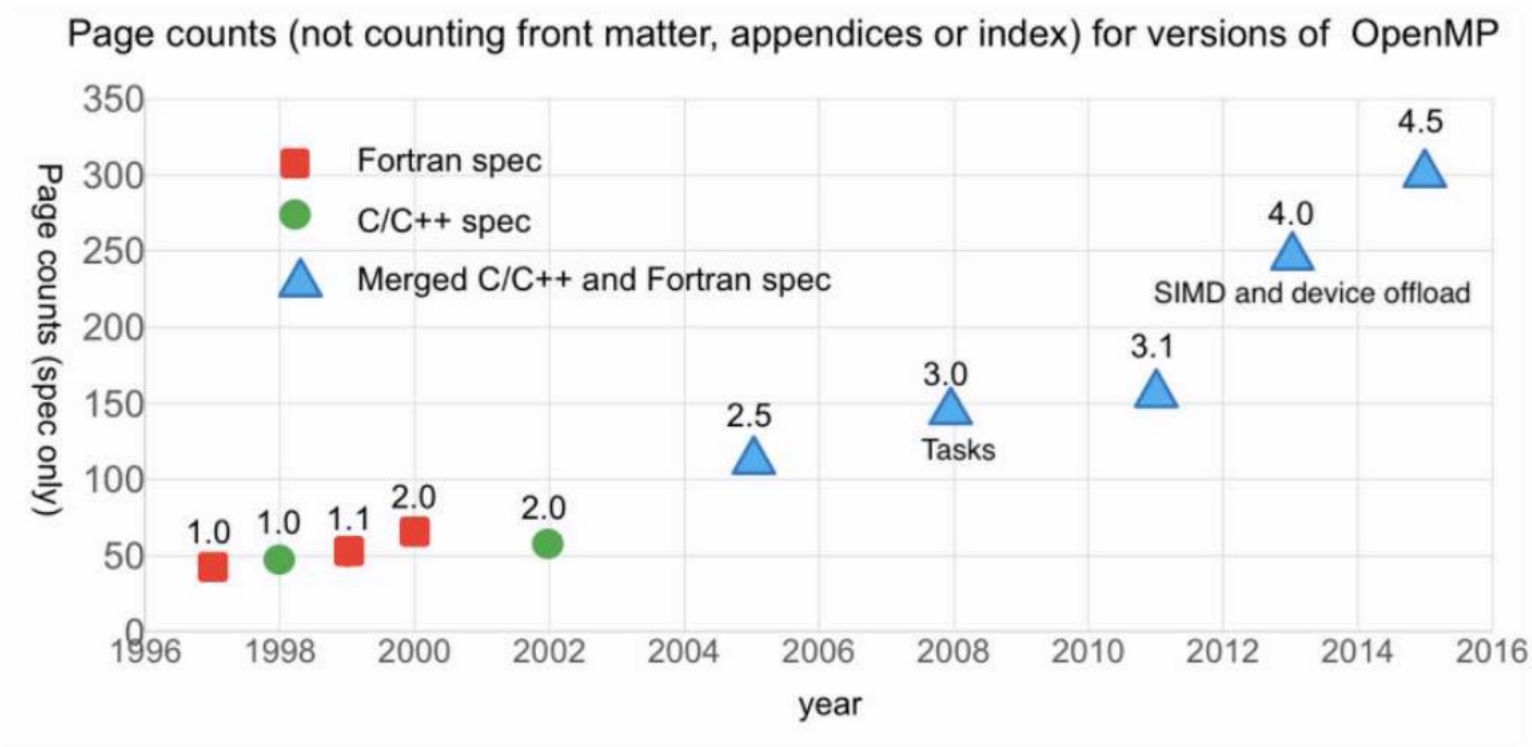
Koncept a záruky

- OpenMP používá direktivy preprocesoru a knihovnu funkcí.
- Program napsaný v OpenMP by mělo být možné spustit sériově i paralelně, s libovolným množstvím jader (vláken). Řadu programů jde napsat jen s direktivami preprocesoru, a pro jedno jádro (vlákno) tedy přeložit i bez knihovny.
- OpenMP nedává záruky na to, jak implementuje jednotlivé direktivy či funkce.
- OpenMP nedává záruky na shodný výstup při použití ruzného počtu jader (vláken), např. operace v plovoucí řádové čárce se mohou vykonávat jiném pořadí.
- OpenMP také nedává záruky na shodný výstup přes různé implementace knihovny.

Dnešní přednáška

OpenMP

- OpenMP je specifikace, která se vyvíjí (podobně jako C++):

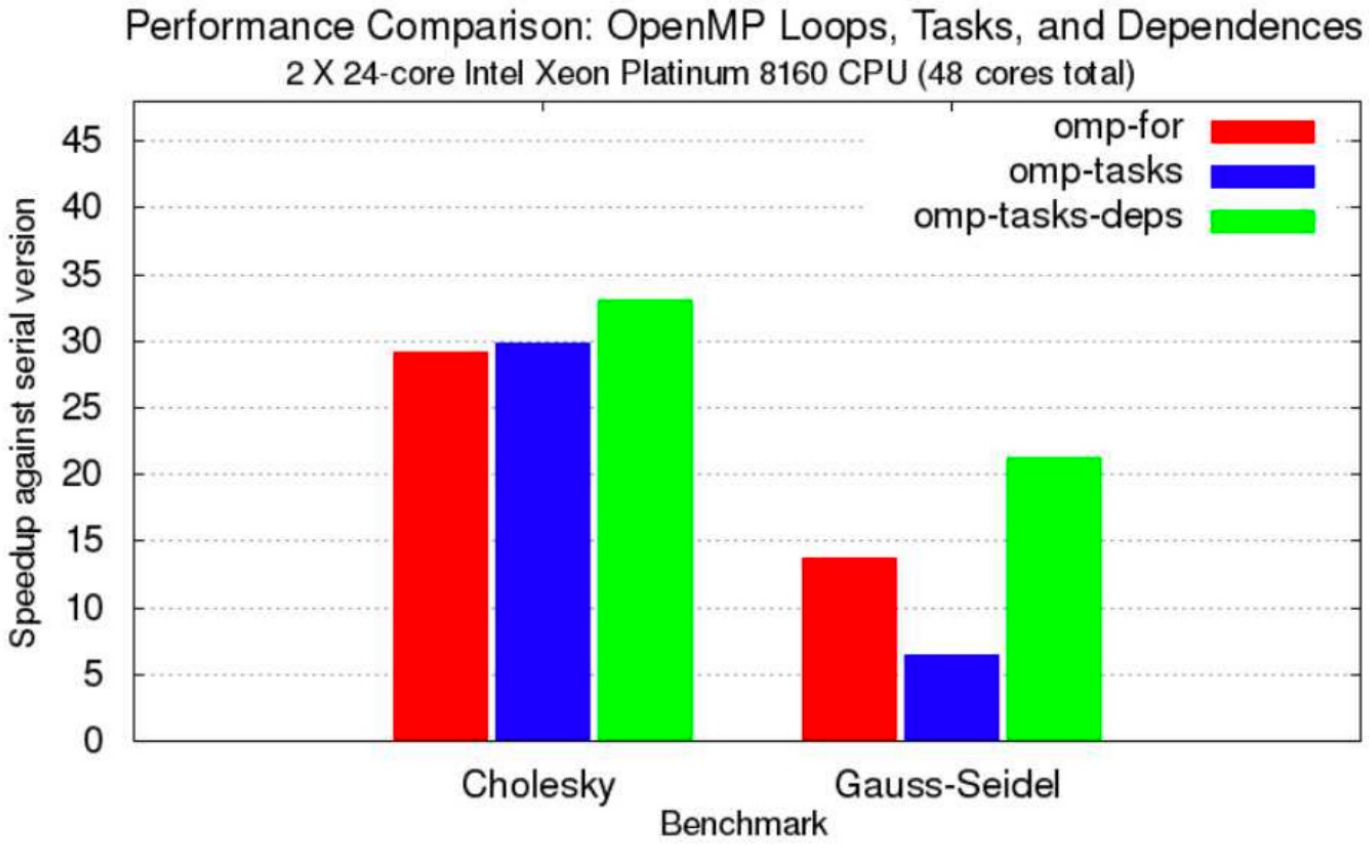


"The Ongoing Evolution of OpenMP"
https://doi.org/10.1109/JPROC.2018.2853600

Dnešní přednáška

OpenMP

- OpenMP je specifikace, která se vyvíjí (podobně jako C++):



"The Ongoing Evolution of OpenMP"
https://doi.org/10.1109/JPROC.2018.2853600

OpenMP – úvod

Koncept a záruky

- OpenMP je specifikace, která má mnoho implementací, je implementováno mnoha překladači a knihovnami.

  https://www.openmp.org/resources/openmp-compilers-tools/

- Na cvičeních uvidíme GNU libgomp. Např. uživatelé clang používají libomp nebo libomptarget, uživatelé překladačů Intel liomp5 atp.
- GNU knihovna umožňuje využití hardware od NVIDIA a AMD (Radeon).
- Knihovna od Intelu umožňuje využití hardware od Intel (vč. Intel® Graphics Xe.) a nabízí se s velkým balíkem dalších nástrojů. Dříve byla komerční, dnes je zdarma:

  https://software.intel.com/content/www/us/en/develop/tools/performance-libraries.html

- Superpočítače často mají vlastní "naladěnou" knihovnu.

OpenMP – úvod, spuštění

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

- V rámci C/C++ exportovány hlavičkou omp.h
  - rozšíření kompilátoru
  - lze přistoupit pomocí #pragma omp
  - základní použití pro paralelizaci for cyklu

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

int main(int argc, char* argv[]) {
   std::cout << "Hello from the main thread\n";

#pragma omp parallel for
   for (int i=0; i<10; i++)
      std::cout << "Item " << i << " is processed by thread" << omp_get_thread_num() << std::endl;

   return 0;
}
```

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++)
    }

```
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 – 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

```
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 – 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

```
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 – 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;
}
```

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 for

```
#pragma omp parallel for schedule(static) num_threads(4)
      for (int i=0; i<20; i++) {
         std::this_thread::sleep_for(std::chrono::milliseconds((i%2)*i*300));
         std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;
      }
   }
```

```
#pragma omp parallel for schedule(dynamic) num_threads(4)
      for (int i=0; i<20; i++) {
         std::this_thread::sleep_for(std::chrono::milliseconds((i%2)*i*300));
         std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;
      }
   }
```

```
#pragma omp parallel for schedule(guided) num_threads(4)
      for (int i=0; i<20; i++) {
         std::this_thread::sleep_for(std::chrono::milliseconds((i%2)*i*300));
         std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;
      }
   }
```

OpenMP – základní způsob práce

Blok sections

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

OpenMP – základní způsob práce

Blok sections

- Můžeme paralelizovat pouze for cykly?
  - Můžeme paralelizovat libovolné {}/metody/úkoly.
  - Vlevo příklad ze specifikace, kde použijeme {}:

https://github.com/OpenMP/Examples/blob/main/sources/Example_parallel.1.c

```
#include <omp.h>

void subdomain(float *x, int istart, int ipoints) {
    int i;
    for (i = 0; i < ipoints; i++) x[istart+i] = 123;
}

void sub(float *x, int npoints) {
    int iam, nt, ipoints, istart;

#pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
    {
        iam = omp_get_thread_num();
        nt =  omp_get_num_threads();
        ipoints = npoints / nt;     /* size of partition */
        istart = iam * ipoints;  /* starting array index */
        if (iam == nt-1)      /* last thread may do more */
            ipoints = npoints - istart;
        subdomain(x, istart, ipoints);
    }
}

int main() {
    float array[10000];
    sub(array, 10000);
    return 0;
}
```

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**)

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();
      }
   }
}
```

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

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();
   }
```

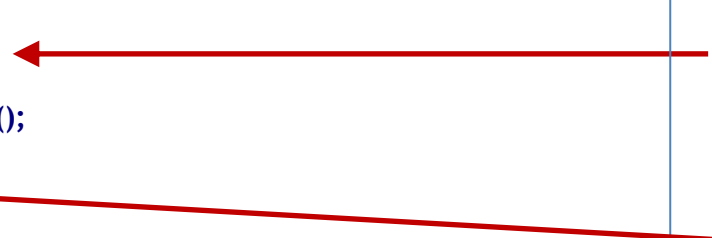
Co udělá tento kód?

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 zamknout **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

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

- V rámci bloku omp parallel můžeme určovat viditelnost proměnných
  - lastprivate(<proměnná1>, <proměnná2>,...)
    - hodnota proměnné z poslední loop/sekce se zapíše do původní proměnné

```
std::cout << "Variable 'b' = " << b << " before lastprivate " << std::endl;

   int i;
#pragma omp parallel for num_threads(10) lastprivate(i,b)
   for (i=0; i<10; i++) {
      std::cout << "Loop " << i << std::endl;
      if (i == 9)
         b = 15;
   }

   std::cout << "Variable 'b' = " << b << " after lastprivate " << std::endl;
   std::cout << "Variable 'i' = " << i << " after lastprivate " << std::endl;
```

OpenMP – přístup k proměnným

Privátní a sdílené proměnné

- V rámci OpenMP můžeme využít známých technik pro synchronizaci přístupu ke sdíleným proměnným
  - kritické sekce, kde pouze 1 vlákno může být v kritické sekci, pomocí #pragma omp critical([<název_sekce>])
  - atomické operace pomocí #pragma omp atomic
  - Zámky pomocí omp_nest_lock_t, omp_lock_t, které je potřeba inicializovat (omp_init_lock, omp_init_nest_lock)
    příp. standardní (C++11) mutexy
  - flush(<proměnná1>,…)
    - zabezpečí synchronizaci sdílených proměnných
    - před každým čtením, po každém zápise
    - pomalé z důvodu zabezpečení konzistence

OpenMP – přístup k proměnným

Privátní a sdílené proměnné

- V rámci OpenMP můžeme využít známých technik pro synchronizaci přístupu ke sdíleným proměnným
  - kritické sekce, kde pouze 1 vlákno může být v kritické sekci, pomocí #pragma omp critical([<název_sekce>])

```
#include <omp.h>
int soucet;
void Inkrement() {
#pragma omp critical
    soucet += 1;
}
```

```
long local_atomic(std::vector<int>& vector, std::vector<int>& histogram) {
   int j;
#pragma omp parallel for private(j) shared(vector,histogram) num_threads(thread_count)
   for (int i=0; i<SIZE; i++) {
      j = vector[i] % PARTS;
#pragma omp critical
      histogram[j]++;
   }
}
```

OpenMP – přístup k proměnným

Privátní a sdílené proměnné

- V rámci OpenMP můžeme využít známých technik pro synchronizaci přístupu ke sdíleným proměnným
  - atomické operace pomocí #pragma omp atomic

```
#include <omp.h>
int soucet;
void Inkrement() {
#pragma omp atomic
    soucet += 1;
}
```

```
long local_atomic(std::vector<int>& vector, std::vector<int>& histogram) {
   int j;
#pragma omp parallel for private(j) shared(vector,histogram) num_threads(thread_count)
   for (int i=0; i<SIZE; i++) {
      j = vector[i] % PARTS;
#pragma omp atomic
      histogram[j]++;
   }
}
```

OpenMP – přístup k proměnným

Privátní a sdílené proměnné

- V rámci OpenMP můžeme využít známých technik pro synchronizaci přístupu ke sdíleným proměnným
  - Zámky pomocí omp_nest_lock_t asi chceme obalit RAII

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

int count;
omp_nest_lock_t countMutex;

struct CountMutexInit {
    CountMutexInit() { omp_init_nest_lock   (&countMutex);   }
    ~CountMutexInit() { omp_destroy_nest_lock(&countMutex); }
} countMutexInit;

struct CountMutexHold {
    CountMutexHold() { omp_set_nest_lock    (&countMutex); }
    ~CountMutexHold() { omp_unset_nest_lock  (&countMutex); }
};

void Tick() {
    CountMutexHold destroyAtEndOfScope;

}

int main() {
    Tick(); return 0;
}
```

https://www.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/model-threading-designs/add-parallelism-to-your-program/replace-annotations-with-openmp-code/openmp-locks.html?wapkw=openmp%20locks

OpenMP – procvičovací příklad

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

void work(int n) { std::cout << n; }
void sub3(int n)
{
    work(n);
#pragma omp barrier
    work(n);
}
void sub2(int k)
{
#pragma omp parallel shared(k)
    sub3(k);
}
void sub1(int n)
{
    int i;
#pragma omp parallel private(i) shared(n)
    {
#pragma omp for
        for (i=0; i<n; i++)
            sub2(i);
    }
}
int main()
{
    sub1(2);
    sub2(2);
    sub3(2);
    return 0;
}
```

OpenMP – procvičovací příklad

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

void work(int n) { std::cout << n; }
void sub3(int n) {
    work(n);
#pragma omp barrier
    work(n);
}
void sub2(int k) {
#pragma omp parallel shared(k)
    sub3(k);
}
void sub1(int n) {
    int i;
#pragma omp parallel private(i) shared(n)
    {
#pragma omp for
        for (i=0; i<n; i++)
            sub2(i);
    }
}
int main() {
    sub1(2); sub2(2); sub3(2); return 0;
}
```

OpenMP – procvičovací příklad

worksharing Constructs Inside a critical Construct

```
void critical_work() {
    int i = 1;
#pragma omp parallel sections
    {
#pragma omp section
        {
#pragma omp critical (name)
            {
#pragma omp parallel
                {
#pragma omp single
                    {
                        i++;
                    }
                }
            }
        }
    }
}
```

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

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];
        }
```

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)
```

výslední (výstupní) proměnná

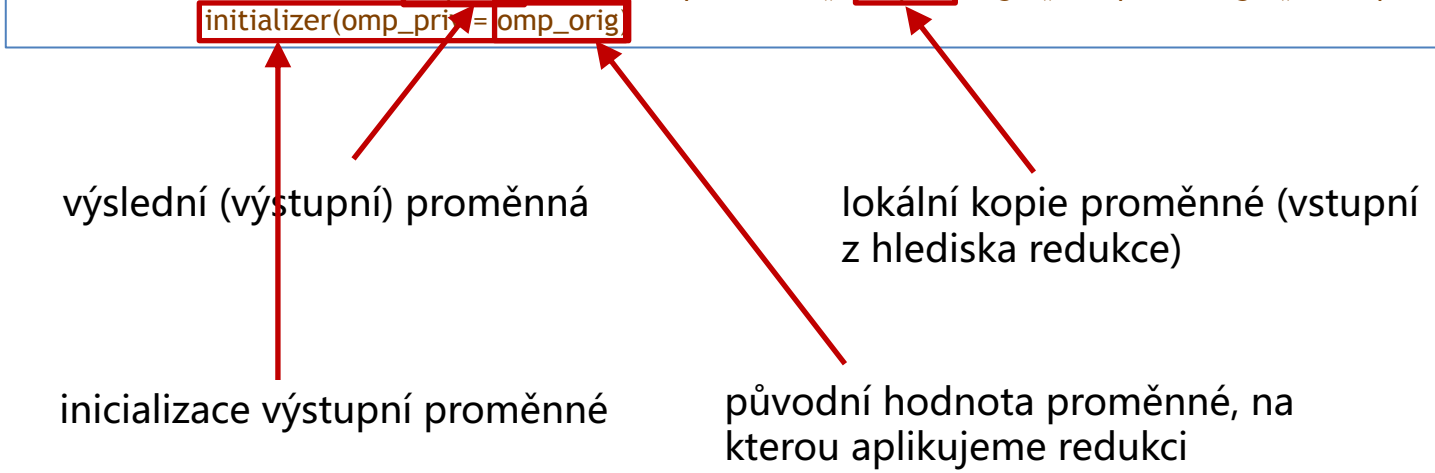
lokální kopie proměnné (vstupní z hlediska redukce)

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 – histogram

```
long local_reduction(std::vector<int>& vector, std::vector<int>& histogram) {
#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)
   int j;
#pragma omp parallel for private(j) shared(vector) num_threads(thread_count) reduction(vec_int_plus:histogram)
   for (int i=0; i<SIZE; i++) {
      j = vector[i] % PARTS;
      histogram[j]++;
   }
}
```

OpenMP

V C++11 (a dalších)

- Jednoduchý způsob paralelizace
- Úroveň „Hello world “: parallel for
- Co se přesně stane?
  - Parallel – vytvoří se několik vláken
  - For – vlákna si rozdělí iterace cyklu
- Platnost proměnných
  - private, shared, default
- Atomic, critical, locks
- Task, section, barrier, nowait
- Thread cancellation