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

Dnešní přednáška

Motivace

Dnešní přednáška

Motivace

Dnešní přednáška

Motivace

Dnešní přednáška

Další nástroje

Dnešní přednáška

OpenMP

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

OpenMP – úvod, spuštění

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

- V rámci C/C++ exportovány hlavičkou „omp.h “
  - rozšíření kompilátoru (různé kompilátory podporují různé verze OpenMP)
  - 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_cykklů>)**

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 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;
      }
   }
```

- Která varianta bude nejrychlejší?

https://goo.gl/a6BEMb

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

```
#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 mehtod(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?

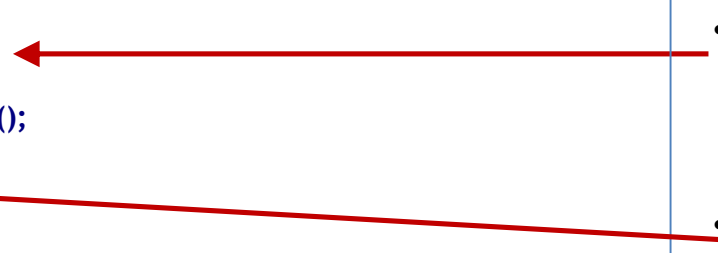
https://goo.gl/a6BEMb

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

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()+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
  - 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 OpenMP můžeme využít známých technik pro synchronizaci přístupu k sdíleným proměnným
  - kritické sekce - #pragma omp critical([<název_sekce>])
    - pouze 1 vlákno může být v kritické sekci
  - zámky
    - existuje omp_lock_t
    - lze použít standardní (C++11) mutexy
  - atomické operace
    - #pragma omp atomic
  - 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

- 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

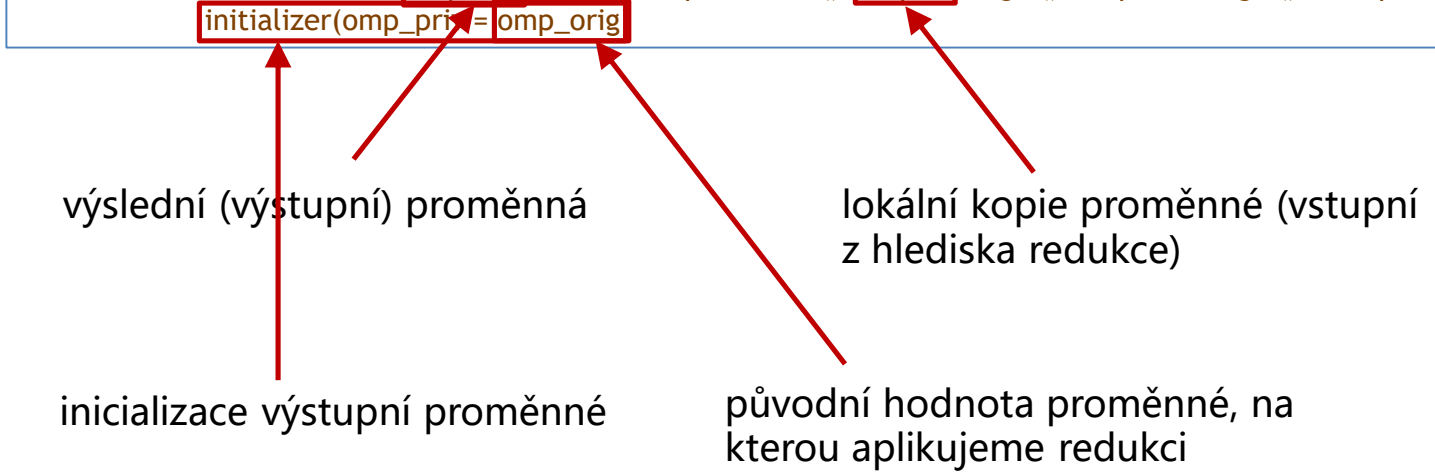
lokální kopie proměnné (vstupní z hlediska redukce)

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 nolocks(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;
        histogram[j]++;
    }
}
```

```
long local_locks(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;
        hist_part_mutex[j].lock();
        histogram[j]++;
        hist_part_mutex[j].unlock();
    }
}
```

OpenMP – histogram

```
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]++;
   }
}
```

```
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]++;
   }
}
```