

Dekompoziční techniky

B4B36PDV – Paralelní a distribuované výpočty

- Opakování z minulého cvičení
- Datová dekompozice se zařezáváním
- Explorativní dekompozice
- Rekurzivní dekompozice

- Zadání čtvrté domácí úlohy

Opakování z minulého cvičení

<http://goo.gl/a6BEMb>

Jakým způsobem bude následující kód proveden?

```
int a = getCurrentValue();
int b = compute(); // casove narocna funkce
int c = getCurrentValue();
if (a < b){
    a.compare_exchange_strong(c, b);
}
```

- Kód nelze zkompilovat
- Do A se vždy uloží B
- Do A se uloží C, pokud B není rovno C
- Do C se uloží A, pokud A není rovno B
- Do A se uloží B, pokud A není rovno C
- Kód vrátí chybu, pokud A není rovno C

V čem může být rozdíl po ukončení výpočtu následujících dvou funkcí?

```
void compute_1(){
    node * null_ptr = nullptr;
    node * a = getMainNode();
    if(a == null_ptr){
        if(getMainNode().compare_exchange_strong
           (null_ptr, getNewNode())){
            return;
        }
    }
    std::cout << a->value << std::endl;}
```

```
void compute_2(){
    node * null_ptr = nullptr;
    node * a = getMainNode();
    if(a == null_ptr){
        if(getMainNode().compare_exchange_strong
           (a, getNewNode())){
            return;
        }
    }
    std::cout << a->value << std::endl;}
```

Cvičení: Problém $3n + 1$

Problém $3n + 1$ (Collatzův problém)

Collatzova posloupnost je definovaná následovně (pro $n \in \mathbb{N}$):

$$f(n) = \begin{cases} n/2 & \text{pro } n \text{ sudé} \\ 3n + 1 & \text{pro } n \text{ liché} \end{cases}$$

Příklad pro počáteční $n = 5$:

5 16 8 4 2 1 4 2 1 4 2 1 ...

Problém $3n + 1$ (Collatzův problém)

Collatzova posloupnost je definovaná následovně (pro $n \in \mathbb{N}$):

$$f(n) = \begin{cases} n/2 & \text{pro } n \text{ sudé} \\ 3n + 1 & \text{pro } n \text{ liché} \end{cases}$$

Příklad pro počáteční $n = 5$:

5 16 8 4 2 1 4 2 1 4 2 1 ...

Má se za to, že tato sekvence vždy dosáhne **1** (*Collatz conjecture*).

Po kolika krocích se tak ale stane?

Collatzova funkce $C(n)$

např. $C(5) = 5$, $C(16) = 4$

Problém $3n + 1$ (Collatzův problém)

Úkol 1: Máme zadanou konečnou podmnožinu přirozených čísel $X \subset \mathbb{N}$. Jaké je minimum funkce $C(n)$ na množině X ?

$$\min_{n \in X} C(n)$$

Problém $3n + 1$ (Collatzův problém)

Úkol 1: Máme zadanou konečnou podmnožinu přirozených čísel $X \subset \mathbb{N}$. Jaké je minimum funkce $C(n)$ na množině X ?

$$\min_{n \in X} C(n)$$

Jak výpočet optima paralelizovat?

Problém $3n + 1$ (Collatzův problém)

Úkol 1: Máme zadanou konečnou podmnožinu přirozených čísel $X \subset \mathbb{N}$. Jaké je minimum funkce $C(n)$ na množině X ?

$$\min_{n \in X} C(n)$$

Jak výpočet optima paralelizovat?

Jak paralelní výpočet zrychlit? Musíme vždy generovat celé sekvence?

Problém $3n + 1$ (Collatzův problém)

Úkol 1: Máme zadanou konečnou podmnožinu přirozených čísel $X \subset \mathbb{N}$. Jaké je minimum funkce $C(n)$ na množině X ?

$$\min_{n \in X} C(n)$$

Jak výpočet optima paralelizovat?

Jak paralelní výpočet zrychlit? Musíme vždy generovat celé sekvence?

Doimplementujte metodu `findmin_parallel`

Doimplementujte tělo metody `findmin_parallel` v souboru `decompose.cpp` pro paralelní nalezení optima $C(n)$ na množině X (reprezentované vektorem `data`). Udržujte si dosud nalezené optimum pro zařezávání nepotřebných výpočtů $C(n)$ (tj., ve chvíli, kdy daný výpočet prokazatelně vede k suboptimálnímu řešení).

Úkol 2: Nalezněte číslo $n \in \mathbb{N}$ takové, že $C(n) \geq k$.

Úkol 2: Nalezněte číslo $n \in \mathbb{N}$ takové, že $C(n) \geq k$.

Sekvenčně je to jednoduché:

```
unsigned long i = 1;  
while(collatz(i) < k) i++;
```

Jak tento výpočet zparalelizujeme?

Úkol 2: Nalezněte číslo $n \in \mathbb{N}$ takové, že $C(n) \geq k$.

Sekvenčně je to jednoduché:

```
unsigned long i = 1;  
while(collatz(i) < k) i++;
```

Jak tento výpočet zparalelizujeme?

Doimplementujte metodu `findn_parallel`

Doimplementujte tělo metody `findn_parallel` pro paralelní nalezení čísla n , pro které platí $C(n) \geq k$.

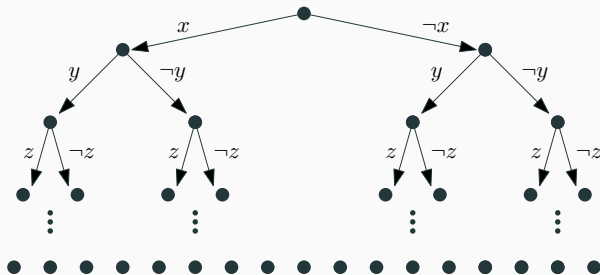
⚠ Pokud *explorujeme* prostor možných řešení a testujeme, zda existuje prvek, který splňuje nějakou podmínku, chceme výpočet ukončit okamžitě po nalezení prvního vhodného prvku.

⚠ Pokud *explorujeme* prostor možných řešení a testujeme, zda existuje prvek, který splňuje nějakou podmínku, chceme výpočet ukončit okamžitě po nalezení prvního vhodného prvku.

V případě paralelizace výpočtu chceme ukončit všechna vlákna!

Explorativní dekompozice: Řešení problému SAT

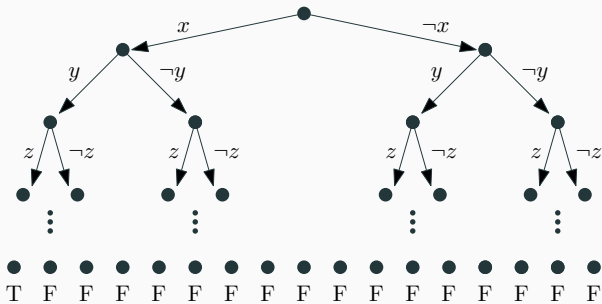
Chceme splnit booleovskou funkci ϕ nad booleovskými proměnnými x, y, z, \dots



Máme 4 vlákna – jak byste úlohu paralelizovali?

Explorativní dekompozice: Řešení problému SAT

Chceme splnit booleovskou funkci ϕ nad booleovskými proměnnými x, y, z, \dots



```
#pragma omp parallel  
#pragma omp for  
for (...){  
    if (condition){  
        #pragma omp cancel for  
    }  
}
```

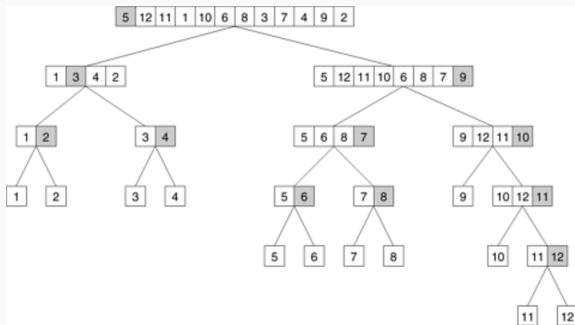
```
#pragma omp parallel
{
    while(true){
        #pragma omp cancellation point parallel
        if (condition){
            #pragma omp cancel parallel
        }
    }
}
```

⚠ Nutné nastavit v prostředí **OMP_CANCELLATION = true !**

Rekurzivní dekompozice

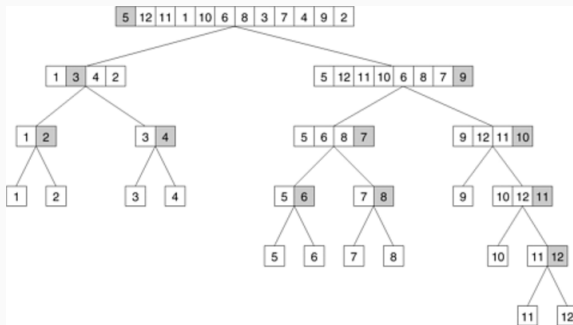
Z algoritmizace víte, že pro některé úlohy je vhodná rekurze.

Vzpomeňte si na řazení! (např. quick-sort z přednášky)



Z algoritmizace víte, že pro některé úlohy je vhodná rekurze.

Vzpomeňte si na řazení! (např. quick-sort z přednášky)



Jak takovýto rekurzivní výpočet zparalelizovat?

U rekurzivních úloh často ani nevíme, jaké podúkoly budeme muset řešit...

Řešení: Budeme paralelní úlohy spouštět dynamicky!

Ve chvíli, kdy potřebujeme zavolat jinou rekurzivní metodu spustíme nový `#pragma omp task`. Ten se zpracuje ve chvíli, kdy nějaké vlákno nemá, co dělat. Vzpomeňte si na thread-pool!

Řešení: Budeme paralelní úlohy spouštět dynamicky!

Ve chvíli, kdy potřebujeme zavolat jinou rekurzivní metodu spustíme nový `#pragma omp task`. Ten se zpracuje ve chvíli, kdy nějaké vlákno nemá, co dělat. Vzpomeňte si na thread-pool!

Otázka: Jak se tento přístup liší od `#pragma omp parallel for schedule(dynamic)`?

Příklad #pragma omp task: Paralelní suma

```
float sum(const float *a, size_t n){
    float r;
    #pragma omp parallel
    #pragma omp single
    r = parallel_sum(a, n);
    return r;
}
```

```
static float parallel_sum(const float *a, size_t n){
    if (n <= CUTOFF) { return serial_sum(a, n);}
    float x, y;          size_t half = n / 2;
    #pragma omp task shared(x)
    x = parallel_sum(a, half);
    #pragma omp task shared(y)
    y = parallel_sum(a + half, n - half);
    #pragma omp taskwait
    x += y;
    return x;
}
```

Fibonacciho posloupnost

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Fibonacciho posloupnost (pro $n \in \mathbb{N}$) je definovaná:

$$F(n) = \begin{cases} 1 & \text{pokud } n = 1 \text{ nebo } n = 2 \\ F(n-1) + F(n-2), & \text{jinak} \end{cases}$$

Fibonacciho posloupnost

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Fibonacciho posloupnost (pro $n \in \mathbb{N}$) je definovaná:

$$F(n) = \begin{cases} 1 & \text{pokud } n = 1 \text{ nebo } n = 2 \\ F(n-1) + F(n-2), & \text{jinak} \end{cases}$$

Doimplementujte metodu `fibonacci_parallel_worker`

Doimplementujte tělo metody `fibonacci_parallel_worker` v souboru `decompose.cpp`. Rekurzivní volání spouštějte pomocí direktivy `#pragma omp task`.

⚠ Proměnné v `task` jsou privátní (`lastprivate`) pro daný `task`, pokud neřeknete jinak (pomocí parametru OpenMP `shared(x)`).

Díky za pozornost!

Budeme rádi za Vaši
zpětnou vazbu! →



[https://forms.gle/
yi7FWBEw3mxgnJ9P7](https://forms.gle/yi7FWBEw3mxgnJ9P7)