Řadící algoritmy

B4B36PDV – Paralelní a distribuované výpočty

Osnova

- Opakování z minulého cvičení

- Dynamické vytváření úloh s #pragma omp task
- Paralelní merge sort
- Paralelní counting sort

- Zadání páté domácí úlohy

Opakování z minulého cvičení

http://goo.gl/a6BEMb

Co můžete říct o tomto kódu?

```
#pragma omp parallel
#pragma omp for
for(int i = 0 ; i < size ; i++) {
  if(is_solution(candidates[i])) {
    std::cout << candidates[i]
              << "is a solution" << std::endl;
    break;
  }
}
```

- Nepůjde pravděpodobně zkompilovat
- Paralelní blok skončí po nalezení prvního řešení
- Paralelní blok skončí, až všechna vlákna najdou řešení
- Aby blok skončil ihned po nalezení řešení, musíme (vhodně) doplnit `#pragma omp cancel for`
- Aby blok skončil ihned po nalezení řešení, musíme (vhodně) doplnit `#pragma omp cancelation point for`
- Měli bychom nastavit proměnnou prostředí `OMP_CANCELLATION=true`

Jak se bude chovat následující kód?

```
int parallel_worker(int d){
  if (d == 1) return 1;
  int t1 = 0, t2 = 0;
  #pragma omp task
  t1 = parallel_worker(d-1);
  #pragma omp task
  t2 = parallel_worker(d-1);
  #pragma omp taskwait
  return t1+t2;
}
(...)
#pragma omp parallel num_threads(4)
std::cout << parallel_worker(3) ;
```

#pragma omp task

#pragma omp task

Pokud nevíme, jaké úlohy budeme muset v průběhu výpočtu řešit, můžeme je vytvářet dynamicky...

```
void traverse(node * n) {
  for(node * successor : n->getSuccessors()) {
    #pragma omp task
    traverse(successor);
  }

  do_something();

  #pragma omp taskwait
}
```

#pragma omp task

Co kdybychom chtěli z tasků ale něco vracet?

```
unsigned long long traverse_and_sum(node * n) {
  std::atomic<unsigned long long> sum {0};
  for(node * successor : n->getSuccessors()) {
    #pragma omp task shared(sum)
    sum += traverse_and_sum(successor);
  }
  sum += n->getValue();
  #pragma omp taskwait

  return sum.load();
}
```

⚠ **Pozor!** Nutno použít `shared` (pro přístup k proměnné) a `taskwait`! Nepoužití těchto konstruktů povede k špatnému výsledku programu (data se nezapíší globálně) nebo i k pádu (proměnná `sum` zanikne po `return`)!

#pragma omp task

Něco nám tam ale chybí... Ještě potřebujeme „někoho“, kdo `tasky` bude řešit. Potřebujeme si připravit vlákna!

```
unsigned long long start_traversal() {
  #pragma omp parallel   // Vytvoříme si tým vláken
  traverse_and_sum(root);
}
```

#pragma omp task

Něco nám tam ale chybí... Ještě potřebujeme „někoho“, kdo `tasky` bude řešit. Potřebujeme si připravit vlákna!

```
unsigned long long start_traversal() {
  #pragma omp parallel   // Vytvoříme si tým vláken
  traverse_and_sum(root);
}
```

Rychlá otázka: Stane se skutečně to, co bychom chtěli?

#pragma omp task

My ale chceme, aby kořen zpracovávalo pouze jedno vlákno!

```
unsigned long long start_traversal() {
  #pragma omp parallel
  #pragma omp single     // pouze jednou!
  traverse_and_sum(root);
}
```

#pragma omp task

My ale chceme, aby kořen zpracovávalo pouze jedno vlákno!

```
unsigned long long start_traversal() {
  #pragma omp parallel
  #pragma omp single     // pouze jednou!
  traverse_and_sum(root);
}
```

Rychlá otázka: Stane se skutečně to, co bychom chtěli?

⚠ Režie s vytvářením a správou tasků může být drahá.

- Tasky chceme vytvářet tehdy, pokud to povede k lepšímu vytížení procesoru.
- … ale ne nutně výhradně spravováním tasků ;-)

```
unsigned long long traverse_and_sum(node * n, int depth) {
  if (depth >= MAX_ALLOWED_DEPTH)
    return traverse_and_sum_sequential(n);

  std::atomic<unsigned long long> sum {0};
  for(node * successor : n->getSuccessors()) {
    #pragma omp task shared(sum)
    sum += traverse_and_sum(successor, depth++);
  }
  sum += n->getValue();
  #pragma omp taskwait

  return sum.load();
}
```

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
    return x + y;
}
```

Paralelní merge sort

Paralelní merge sort

Merge sort je řadící algoritmus, který pracuje následovně:

1. Rozdělí neseřazenou množinu dat na dvě podmnožiny o přibližně stejné velikosti.
2. Seřadí obě podmnožiny.
3. Spojí seřazené podmnožiny do jedné seřazené množiny.

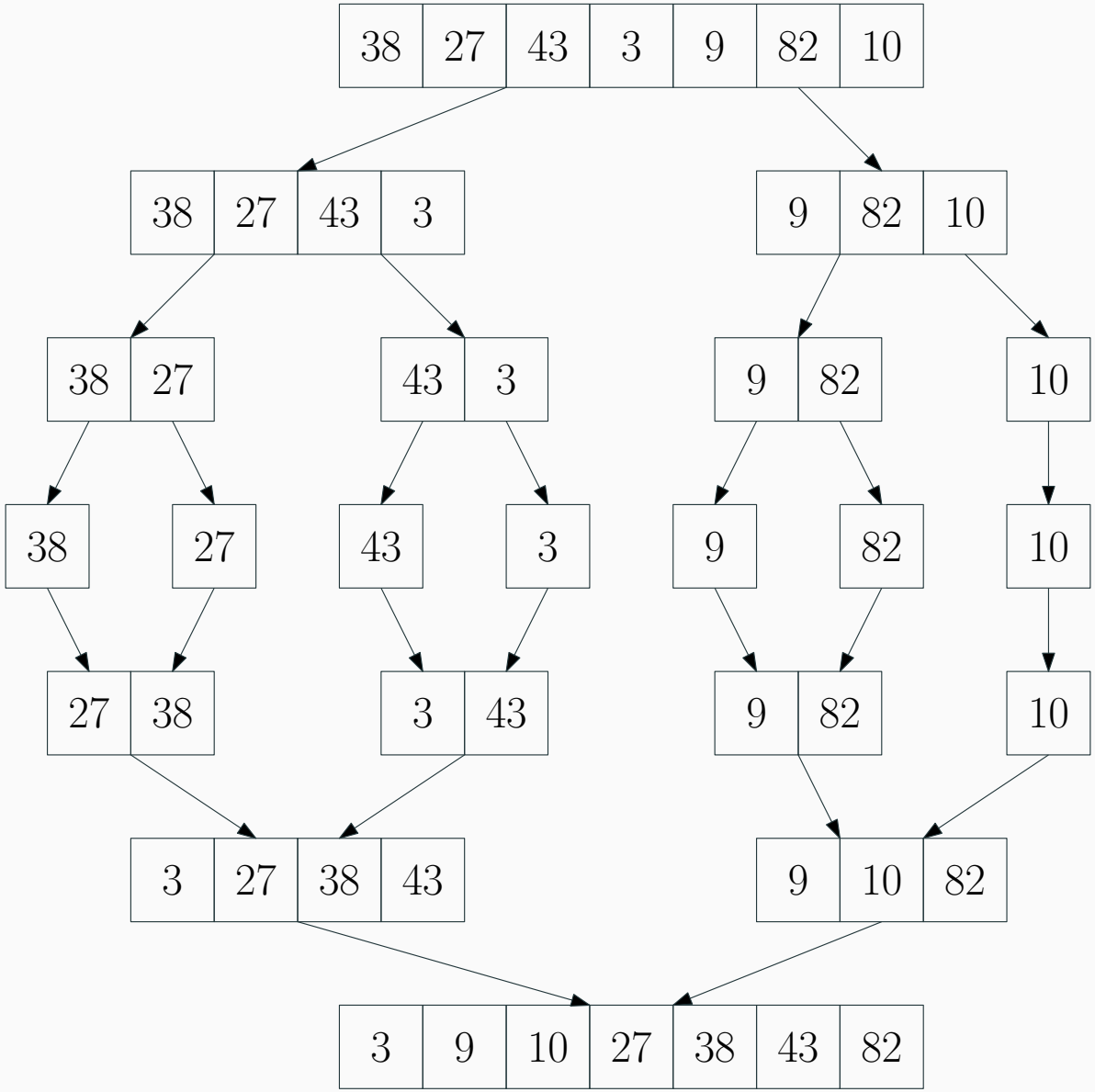
```
function mergesort(m)
    if (length(m) <= 1) return m

    middle = length(m) / 2
    left = m[0 ... middle-1]
    right = m[middle ... length(m)-1]

    left = mergesort(left)
    right = mergesort(right)

    return merge(left, right)
```

Paralelní merge sort



Paralelní merge sort

Doimplementujte metodu `mergesort_parallel`

Doimplementujte tělo metody `mergesort_parallel(...)` (a případných dalších metod, které budete potřebovat) v souboru `mergesort_parallel.h`. Pro implementaci můžete využít metodu `merge(...)` a můžete se inspirovat sekvenční implementací, kterou naleznete v souboru `mergesort_sequential.h`.

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

Otázka: Jakou složitost má sekvenční mergesort? A jak je na tom jeho paralelní verze?

Paralelní counting sort

Counting sort

Uvažujme, že máme za úkol seřadit pole prvků, které obsahuje hodnoty z malého omezeného rozsahu $a \leq x \leq b$.

Pak může být použití standartních algoritmů se složitostí $O(n \log n)$ nevhodné.

Counting sort:

1. Napočítáme si počty jednotlivých prvků $c(x)$ z rozsahu $x \in [a, b]$ („histogram“)
2. Počty prvků projdeme ve vzestupném pořadí. Prvek x zapíšeme do výstupního pole $c(x)$-krát.

$\rightarrow$ Složitost $O(n + k)$, kde $k = b - a + 1$

Counting sort

1. Napočítáme si počty jednotlivých prvků $c(x)$ z rozsahu $x \in [a, b]$ („histogram“)
2. Počty prvků projdeme ve vzestupném pořadí. Prvek x zapíšeme do výstupního pole $c(x)$-krát.

Jak bychom kroky 1 a 2 mohli paralelizovat?

Counting sort

1. Napočítáme si počty jednotlivých prvků $c(x)$ z rozsahu $x \in [a, b]$ („histogram“)
2. Počty prvků projdeme ve vzestupném pořadí. Prvek x zapíšeme do výstupního pole $c(x)$-krát.

Jak bychom kroky 1 a 2 mohli paralelizovat?

Doimplementujte metodu `counting_parallel`

Doimplementujte tělo metody `counting_parallel(...)` v souboru `countingsort.h`. Inspirovat se můžete sekvenční implementací tohoto řadícího algoritmu v metodě `counting_sequential(...)`

SPOILER ALERT!

Prefixní suma

1. Napočítáme si počty jednotlivých prvků $c(x)$ z rozsahu $x \in [a, b]$ („histogram“)
2. Počty prvků projdeme ve vzestupném pořadí. Prvek x zapíšeme do výstupního pole $c(x)$-krát.

Bod (2) algoritmu nešel snadno paralelizovat, protože nevíme, kam máme dané číslo umístit bez toho, abychom vyřešili předešlá čísla!

$$c(x) = [?,\ ?,\ 5,\ ?,\ ?]$$

→ Pojďme to vyřešit...

Prefixní suma

Pro posloupnost čísel $x_0, x_1, x_2, \ldots$ je prefixní suma posloupnost $y_0, y_1, y_2, \ldots$ taková, že

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= y_0 + x_1 \\ y_2 &= y_1 + x_2 \\ &\vdots \end{aligned}$$

Příklad:

Vstupní sekvence:	1	2	3	4	5	6	...
Prefixní suma:	1	3	6	10	15	21	...

Doimplementujte metodu `counting_parallel`

Použijte prefixní sumu pro paralelizaci bodu (2) counting sortu.

Prefixní suma

Jak bychom mohli výpočet prefixní sumy paralelizovat?

Hodnota y_i stále závisí na hodnotě y_{i-1}

Doimplementujte metodu `prefix_sum_parallel`

Doimplementujte tělo metody `prefix_sum_parallel` v souboru `prefixsum.h`.

Zadání páté domácí úlohy

Paralelní radix sort

Algoritmus pro lexikografické seřazení řetězců stejné délky.

Naimplementujte metodu `radix_par` v souboru `sort.cpp`.

Za správné výsledky a rychlé zpracování dostanete až **2b**.

Soubory `sort.cpp` a `sort.h` nahrajte do systému BRUTE.

Díky za pozornost!

Budeme rádi za Vaši zpětnou vazbu! →

https://bit.ly/396PnTE