Inleveropgave 2b: Threaded Merge Sort

Leerling: Sietse Neve

Leerlingnummer: 1810364

Github link: https://github.com/AI-S4-2024/inleveropgave-3-array-sum-DrZisnotavailable.git

1. Ontwerp van het algoritme

We sorteren een array van 8 elementen: `[a0, a1, a2, a3, a4, a5, a6, a7]` met 1, 2, 4 en 8 threads.

1.1 Eén thread (sequentieel)

- Werkt als klassieke merge sort:
    1. Splits in twee helften van 4
    2. Elk deel gesorteerd op dezelfde wijze (recursief)
    3. Merge de twee helften

```
[a0 a1 a2 a3 a4 a5 a6 a7]
   |------------4------------|
 Gesorteerd in één thread → [..] → één merge
```

1.2 Twee threads

- Thread 1 sorteert `[a0..a3]`, Thread 2 sorteert `[a4..a7]`
- Daarna één thread doet `inplace_merge` op de twee gesorteerde helften

```
[a0 a1 a2 a3 | a4 a5 a6 a7]
    T1 (4)       T2 (4)
         \       /
          merge (8)
```

1.3 Vier threads

- Vier threads, elk sorteert blok van 2 elementen:
    - T1: `[a0,a1]`
    - T2: `[a2,a3]`
    - T3: `[a4,a5]`
    - T4: `[a6,a7]`
- Fase 1: vier locale merges per twee elementen → vier blokken van 2 gesorteerd
- Fase 2: twee merges in twee threads: blok0&1 en blok2&3
- Fase 3: één merge van de twee helften (8 elementen)

```
[ a0 a1 | a2 a3 | a4 a5 | a6 a7 ]
   T1=2    T2=2    T3=2    T4=2
      \    /       \    /
     merge2      merge2   → 2 threads
           \           /
            merge4 (1 thread)
```

1.4 Acht threads

- Elk element in eigen thread (blokken van 1)
- Fase 1: geen werk (singleton)
- Fase 2: vier merges van 2 in vier threads
- Fase 3: twee merges van 4 in twee threads
- Fase 4: één merge van 8 in één thread

```
[a0][a1][a2][a3][a4][a5][a6][a7]
 T x8 (blokken van 1)
  → 4 × merge2  → 2 × merge4  → 1 × merge8
```

2. Complexiteitsanalyse ontwerp

2.1 Sequentieel vs parallel

- **Sequentieel**: tijd $T_1(n) = O(n \log n)$
- **Met p threads** (ideaal zonder overhead): $T_p(n) \approx O((n \log n)/p)$
- **Werk (work)**: $W(n) = O(n \log n)$ voor beide varianten
- **Span (kritische pad)**: $S(n) = O((n/p) + \log p)$ door splits+merges in fasen

2.2 Bottlenecks en overhead

- Thread-spawn en join kosten: O(1) per spawn (kan oplopen tot O(log p))
- In-place merge: data-beweging van O(n) per fusiefase
- Samenvoegen door één thread in laatste stap: O(n)

3. Big-O notatie voor tijd en ruimte

Variant	Tijdcomplexiteit	Ruimtecomplexiteit
Sequentieel MergeSort	O(n log n)	O(1) extra (in-place)
Parallel p threads	O((n log n)/p + n·overhead)	O(p) (stack/threads)

Communication overhead: per splits en merge-fase wordt data in cache/threads verplaatst:

- Aantal fasen ≈ log p
- Per fase verplaatst O(n) elementen → overhead O(n log p)
- Verhouding overhead vs werk: $O((n \log p)/(n \log n)) = O(\log p / \log n)$

4. Uitleg van de PoC-implementatie (stap 4)

De proof-of-concept implementatie in `parallel_merge_sort.cpp` gebruikt `std::thread` om tijdens de recursieve aanroep nieuwe threads te starten tot een maximum aantal is bereikt. Hieronder een samenvatting:

```
template <typename RandomIt>
void parallelMergeSort(RandomIt first, RandomIt last,
                       int max_threads, int active_threads) {
    auto length = std::distance(first, last);
    if (length <= 1)
        return;

    RandomIt middle = first + length / 2;

    // Bepaal of we een nieuwe thread mogen starten
    if (active_threads < max_threads) {
        // Sorteer rechterhelft in een nieuwe thread
        std::thread right_thread(
            parallelMergeSort<RandomIt>, middle, last,
            max_threads, active_threads * 2);

        // Sorteer linkerhelft in huidige thread
        parallelMergeSort<RandomIt>(first, middle,
                                    max_threads, active_threads * 2);

        // Wacht op rechter thread
        right_thread.join();
    } else {
        // Te veel threads, voer sequentieel beide helften uit
        parallelMergeSort<RandomIt>(first, middle, max_threads, active_threads);
        parallelMergeSort<RandomIt>(middle, last, max_threads, active_threads);
    }

    // Merge beide gesorteerde helften in-place
    std::inplace_merge(first, middle, last);
}
```

Toelichting van belangrijkste onderdelen:

1. **Threadlimieten beheren**
    - `max_threads`: het maximum aantal threads dat tegelijk actief mag zijn.

- `active_threads`: het huidige aantal actieve threads (begint op 1 in `main`).

2. **Thread spawning**
    - Als `active_threads < max_threads`, wordt de rechterhelft in een nieuwe `std::thread` gesorteerd.
    - De linkerhelft wordt direct in de huidige thread verwerkt.
    - Daarna roept de code `join()` op om te wachten tot beide helften klaar zijn.
3. **Sequentiële fallback**
    - Zodra het maximum is bereikt, worden beide helften sequentieel gesorteerd zonder extra overhead.
4. **Inplace merge**
    - Deze STL-functie voegt twee aangrenzende gesorteerde delen in-place samen tot één geheel zonder extra geheugenallocatie.

Benchmarkresultaten:

Gebaseerd op 10 runs per aantal threads met een lijst van 100.000 cijfers.

Aantal Threads	Tijd (microseconden)
1	97879
2	52410
4	36969
8	33029
16	36538

Analyse:

Bij gebruik van meer threads daalt het aantal microsecondes, totdat ik 16 threads gebruik, dan neemt het weer toe. Dit betekent dat de overheid bij het maken van 16 threads, groter is dan de tijd die wordt gewonnen.

5. Ontwerp met thread pool

In de voorgaande tekst stond per ongeluk twee keer dezelfde beschrijving en flow voor het thread-pool ontwerp; dit komt doordat het onderdeel tweemaal is ingevoegd tijdens het opstellen. Hieronder staat de eenduidige, enkele uitleg.

- **Thread pool**: een pool van *T* worker threads wordt éénmalig gestart vóór het sorteerproces.
- **Task queue**: elk deelprobleem (subarray) wordt als taak in de queue geplaatst.
- **Scheduler**: workers halen om de beurt een taak op, voeren `parallelMergeSort` uit en voegen nieuwe subtaken toe zonder extra `spawn`/`join` overhead.

Flow-thread-pool

1. **Master** voegt de initiële taak `[0,n)` toe aan de queue.
2. **Worker** haalt een taak (range) uit de queue:
    - Splits de range in twee helften.
    - Plaatst beide helften als nieuwe taken terug in de queue.
3. Stap 2 wordt herhaald totdat alle taken corresponderen met subarrays van grootte 1.
4. Nadat een worker een taak met twee reeds gesorteerde helften heeft, voert hij `inplace_merge` uit en markeert de taak als voltooid.

Voordelen:

- Minder spawn/join overhead omdat threads persistent zijn.
- Dynamische taakverdeling: idle threads blijven nieuwe taken pakken.

Nadelen:

- Synchronisatie op de queue vereist locks of lock-free structuren.
- Extra geheugen voor het beheren van de queue en taakmetadata.