**Di Riccio Tommaso 615257 21/05/2024 (Update)**

## Compilazione ed esecuzione

Per compilare è possibile utilizzare il comando:

*g++ -std=c++17 -O3 -o Word-Count-par Word-Count-par.cpp -fopenmp*

Per l'esecuzione il comando è:

*./Word-Count-par /opt/SPMcode/A2/filelist.txt [extraworkXline] [topk] [showresults] [nthreads] [chunk-size]*

dove

- *extraworkXline* è il lavoro extra svolto per ogni riga (default = 0)
- *topK* è il numero di parole da mostrare in ordine di frequenza (default = 10)
- *showresults* indica se i risultati devono essere mostrati o no (default = 0)
- *nthreads* è il numero di thread da utilizzare per l'esecuzione (default = 1)
- *chunk_size* è il massimo numero di linee consecutive di un file elaborate da ogni task (default = 100)

## Problemi affrontati e soluzioni

Nell'implementazione della versione parallela ho scelto di creare un OpenMP task per ogni *chunk_size* linee dei file su cui eseguire il conteggio. Questa scelta ha il tentativo di trovare un buon bilanciamento tra il numero di task creati e la loro granularità.

A destra sono riportati alcune delle parti di codice che hanno subito la maggior parte delle modifiche rispetto alla versione sequenziale.

Nella prima parte viene creata una *unordered_map* per ognuno degli *nth* thread creati con la direttiva *parallel* di OpenMP.

```
// define a local UM map for each thread
static umap* local_UM;
#pragma omp threadprivate(local_UM)

#pragma omp parallel num_threads(nth)
{
    // initialize the local UM map
    local_UM = new umap;
```

Successivamente, utilizzando la direttiva *single*, un singolo thread si incarica di creare tutti i task da eseguire (il codice utilizzato per dividere il file in chunk è omesso).

Il task consiste nel contare la frequenza di ogni token presente nel file. Il thread che lo esegue utilizza la propria *umap* locale per memorizzare i conteggi parziali.

```
// A single thread creates the tasks
#pragma omp single
{
    for (auto f : filenames) {···
}
```

Quando tutti i tasks sono portati al termine, i threads possono continuare l'esecuzione, arrivando alla sezione *omp critical*. Questa direttiva è necessaria per fare in modo che ogni thread possa aggiornare la *umap* globale con i propri risultati locali in mutua esclusione.

```
// Use a critical section to update the global UM map
#pragma omp critical
{
    for (const auto& entry : *local_UM) {
        UM[entry.first] += entry.second;
    }
}
```

Una ulteriore modifica è stata eseguita sulla variabile *total_words* che è stata trasformata in *atomic_int*. Questo ha permesso ai thread di eseguire in parallelo il metodo *tokenize_line(...)*, su diverse linee di testo, mantenendo la correttezza del conteggio delle parole totali.

**Esperimenti e risultati** *(test eseguiti su spmcluster.unipi.it)*

Sia per la versione sequenziale che per quella parallela sono stati considerati i tempi di esecuzione escludendo il sorting, dato che quest'ultimo era equivalente per entrambe le versioni.

Nella seguente tabella sono riportati i tempi per la versione sequenziale utilizzando 0, 1000 e 10000 come lavoro extra

| Sequenziale | | | |
|---|---|---|---|
| Extra | 0 | 1000 | 10000 |
| Tempo (sec) | 5,6 | 19,91 | 148,62 |

Per la versione parallela sono riportati i tempi di esecuzione utilizzando 2, 5, 10 e 20 threads, con i relativi *extraworkXline* e *chunk*_size utilizzata. La *chunk*_size è stata scelta sperimentando diversi valori per le varie combinazioni.

| 2 Threads | | | |
|---|---|---|---|
| Extra | 0 | 1000 | 10000 |
| Chunk size | 5000 | 5000 | 5000 |
| Tempo (sec) | 3,84 | 11,47 | 77,25 |

| 5 Threads | | | |
|---|---|---|---|
| Extra | 0 | 1000 | 10000 |
| Chunk size | 5000 | 2500 | 2500 |
| Tempo (sec) | 4,21 | 9,46 | 35,62 |

| 10 Threads | | | |
|---|---|---|---|
| Extra | 0 | 1000 | 10000 |
| Chunk size | 500 | 250 | 250 |
| Tempo (sec) | 4,88 | 8,83 | 22,03 |

| 20 Threads | | | |
|---|---|---|---|
| Extra | 0 | 1000 | 10000 |
| Chunk size | 100 | 100 | 100 |
| Tempo (sec) | 5,31 | 8,72 | 15,74 |

Da questi dati è possibile ricavare speedup ed efficienza, riportata nelle seguenti tabelle

| | | **Extra** | | |
|---|---|---|---|---|
| | | 0 | 1000 | 10000 |
| **Threads** | 2 | 1,46 | 1,74 | 1,92 |
| | 5 | 1,33 | 2,10 | 4,17 |
| | 10 | 1,15 | 2,25 | 6,75 |
| | 20 | 1,05 | 2,28 | 9,44 |

| | | **Extra** | | |
|---|---|---|---|---|
| | | 0 | 1000 | 10000 |
| **Threads** | 2 | 73% | 87% | 96% |
| | 5 | 27% | 42% | 83% |
| | 10 | 11% | 23% | 67% |
| | 20 | 5% | 11% | 47% |

Come è possibile vedere i risultati sono pessimi quanto *extraworkXline* ha valore 0. I risultati migliorano al suo aumento ma senza superare il 47% di efficienza utilizzando 20 thread.

Sperimentando per trovare le cause della bassa efficienza ho ipotizzato due possibili problemi:

- **Creazione dei task**: quando *extraworkXline* è 0, il singolo thread incaricato di creare i task potrebbe non riuscire a stare al passo con la velocità di esecuzione dei task. Un ulteriore aggravante di questo problema risiede nel fatto che lo stesso thread utilizzato per generare i task è anche utilizzato parte delle volte per eseguirli. È stato eseguito un esperimento

parallelizzando il for che itera sui file nel tentativo di velocizzare la creazione dei task ma questo non ha portato benefici per quanto riguarda i tempi di esecuzione.

- **Reduce**: una volta terminati i file ogni thread deve aggiornare in mutua esclusione la *umap* globale con i propri risultati locali. Questo overhead cresce con il numero di thread utilizzati. Il peggioramento dello speedup con l'aumento dei thread, quando il lavoro extra è basso, potrebbe trovare in parte spiegazione con questo problema. Ciò sarebbe confermato dal miglior andamento della speedup al crescere del lavoro extra che porta ad un overhead in proporzione più piccolo.
  Per cercare un miglioramento delle performance ho provato a spostare il merge dei risultati all'interno del task stesso, utilizzando una sezione *critic*. I tempi di esecuzione non sono però cambiati perché, così facendo, è stata probabilmente diminuito l'overhead a fine computazione dei file ma al tempo stesso è stato ridotto il grado di parallelismo dei tasks.

È inoltre possibile notare come all'aumentare del numero di thread si ottengono migliori performance utilizzando *chunk_size* più piccoli. Difatti quando il numero di thread è basso è ragionevole creare task con una granularità più grande per evitare un alto overhead nella creazione di molti task che non verranno poi essere eseguiti in parallelo. D'altra parte, con un altro numero di thread creare molti task con una granularità fine permette di sfruttare in modo migliore le risorse a disposizione.