Analisi in tempo reale di difetti nella produzione L-PBF con Apache Flink

Gianluca Ronzello
Università degli Studi di Roma Tor Vergata
gianlucaronzello@gmail.com

Marco Lorenzini
Università degli Studi di Roma Tor Vergata
marcolorenzini23@gmail.com

***Abstract*—Il processo di stampa 3D Laser Powder Bed Fusion (L-PBF) è soggetto alla formazione di difetti, in particolare porosità, che possono compromettere le proprietà meccaniche del prodotto. Per superare i limiti del controllo qualità ex-post, questo progetto propone una soluzione di monitoraggio in tempo reale basata su Apache Flink, capace di elaborare in streaming le immagini termiche acquisite tramite tomografia ottica (OT). L'obiettivo è stimare la probabilità di difetti durante la fabbricazione, sfruttando le capacità di Flink nella gestione di flussi dati ad alta frequenza e nella definizione di pipeline analitiche a bassa latenza. L'approccio consente interventi tempestivi sul processo, migliorando efficienza e affidabilità.**

I. Introduzione

La necessità di monitorare e intervenire in tempo reale nei processi di produzione avanzata è particolarmente rilevante nella stampa L-PBF, dove piccoli difetti possono influire significativamente sulla qualità del componente. Le tecniche di controllo a posteriori risultano spesso inefficienti, generando costi elevati e sprechi di materiale.

In questo contesto, l'analisi in tempo reale delle immagini termiche ottenute da tomografia ottica rappresenta una valida strategia per la rilevazione precoce di anomalie. Questo progetto implementa una pipeline di processamento streaming tramite Apache Flink, capace di ricevere e analizzare sequenze termiche strato per strato, stimando la probabilità di difetti nel momento stesso in cui il dato è generato. L'adozione di Flink consente l'elaborazione scalabile, distribuita e reattiva necessaria per applicazioni industriali critiche.

II. Architettura pipeline

Ogni componente è eseguito in ambiente containerizzato utilizzando *Docker Compose* per l'orchestrazione. La pipeline, figura 1, prevede i seguenti container:

- **micro-challenger**, container che espone le API REST sulla porta 8866 e fornisce la dashboard di valutazione delle prestazioni (latenza e throughput).
- **jobmanager**, nodo centrale del cluster Flink, esegue il coordinamento dei job e l'interfaccia web su porta 8081.
- **taskmanager**, nodo esecutore per l'elaborazione distribuita dei dati, il numero di slot viene configurato in base al livello di parallelismo.

I container utilizzano le immagini ufficiali di Apache Flink (versione `1.20.1-java17`), mentre `micro-challenger` è avviato tramite un'immagine customizzata, con accesso ai dati condivisi tramite volume locale. I risultati dell'elaborazione Flink sono persistiti nella directory `./Results`, montata sul container del *taskmanager*.

L'interfaccia web del *jobmanager* è accessibile tramite browser all'indirizzo `localhost:8081`, mentre la dashboard prestazionale del *micro-challenger* è disponibile su `localhost:8866`.

Il flusso si articola in:

1) Viene creata una sessione con il *LOCAL-CHALLENGER* tramite API REST.
2) Flink richiede iterativamente le immagini OT da processare attraverso chiamate REST.
3) Ogni immagine viene analizzata in tempo reale all'interno di un job Flink per stimare la probabilità di difetti.
4) I risultati vengono inviati nuovamente al *LOCAL-CHALLENGER* tramite appositi endpoint REST.
5) Al termine del benchmark, viene notificata la conclusione del processamento.

III. Sorgente

Per acquisire i dati generati dal *Local Challenger*, è stata implementata una sorgente custom compatibile con il nuovo `Source API` di Flink. La sorgente è responsabile dell'interazione via REST con il benchmark server, dell'acquisizione dei batch termici in formato binario, e dell'integrazione fluida nel dataflow streaming.

L'implementazione si articola nei seguenti componenti:

1) **MicroChallengerClient**: incapsula la logica di comunicazione HTTP con il server (avvio benchmark, richiesta batch, invio risultati, fine bench).
2) **MicroChallengerSource**: definisce la sorgente, specificando il tipo di dato (`byte[]`), il comportamento limitato (bounded), e i serializzatori di split e checkpoint.
3) **MicroChallengerReader**: implementa la lettura dei dati, interrogando ciclicamente il server per ottenere un nuovo batch tramite `nextBatch()` e marcando la fine dello stream quando non vi sono più dati disponibili.
4) **MicroChallengerEnumerator**: assegna un singolo split logico ai task Flink. Essendo il benchmark sequenziale, non è necessario partizionare lo stream.

L'interazione REST viene gestita tramite l'oggetto `MicroChallengerClient`, che viene instanziato all'interno del reader. I batch sono inviati dal server in formato binario e successivamente deserializzati e processati all'interno della pipeline.

Questa sorgente viene registrata nel job Flink tramite il metodo `env.fromSource(...)` e rappresenta il punto di ingresso dei dati nella pipeline.

A. Deserializzazione dei dati

Il componente `BatchDeserializer` si occupa della trasformazione dei dati binari ricevuti dallo stream in oggetti `Batch` strutturati e utilizzabili nella pipeline Flink. I dati in ingresso sono codificati in formato `MessagePack` e includono una mappa di valori contenente, tra gli altri, i campi `batch_id`, `print_id`, `tile_id`, `layer` e un'immagine TIFF compressa (campo `tif`) contenente i dati di temperatura.

Il deserializzatore esegue i seguenti passi:

- Estrae i campi principali dal messaggio `MessagePack` utilizzando le API di `msgpack-core`.
- Decodifica il campo TIFF per ottenere una matrice bidimensionale di interi (temperatura per ciascun pixel).

Per il parsing del file TIFF, viene utilizzata la libreria `TwelveMonkeys ImageIO`, che supporta la lettura di immagini TIFF contenenti dati a 16 bit. A tale scopo, è stato necessario aggiungere al progetto la relativa dipendenza. La libreria consente di leggere correttamente i dati grezzi dal raster dell'immagine e ricostruire una matrice `int[][]`, in cui ciascun valore rappresenta una misura di temperatura normalizzata.

Questa fase di deserializzazione è fondamentale poiché trasforma i byte in arrivo dalla sorgente in una struttura semantica adatta alle analisi successive.

IV. Query 1

La **Query 1** rappresenta il primo stadio della pipeline di processamento e si occupa dell'analisi dei valori di temperatura per ciascuna tile in ingresso. L'obiettivo è identificare i punti che si trovano al di fuori dell'intervallo di validità: valori inferiori a `5000` rappresentano zone vuote da ignorare, mentre valori superiori a `65000` sono considerati saturati e potenzialmente indicativi di difetti nel processo di stampa. I punti saturati vengono contati, ma esclusi dai successivi stadi di elaborazione.

La logica della query è implementata nella classe `Query1`, applicata direttamente allo stream in ingresso. La funzione opera in modalità *element-wise*, trasformando ogni oggetto `Batch` in un nuovo oggetto `BatchWithMask`, che include una maschera booleana (`validMask`) per filtrare i soli punti validi, e il conteggio dei pixel saturati.

L'output della query viene successivamente convertito in formato CSV e salvato su file tramite un `FileSink`, utile per analisi offline o validazione. Il sink scrive le righe nel formato: `seq id, print id, tile id, saturated`, evitando di inviare i risultati al `Local Challenger`.

V. Query 2

La **Query 2** implementa gli stadi 2 e 3 della pipeline. Dopo aver creato una sliding window che mantiene gli ultimi 3 layer con slide 1, per ogni punto p del layer più recente si calcola la *deviazione di temperatura locale*, definita come la differenza assoluta tra:

- la temperatura media dei *vicini prossimi* di p, ossia di tutti i punti a distanza di Manhattan $0 \leq d \leq 2$ da p, considerati su tutti e tre i layer;
- la temperatura media dei *vicini esterni* di p, ossia di tutti i punti a distanza di Manhattan $2 < d \leq 4$ da p, considerati su tutti e tre i layer.

Se la deviazione locale supera la soglia di 6000, il punto viene classificato come outlier. Inoltre, per ogni finestra la query restituisce i 5 punti con la deviazione più alta.

L'implementazione (classe `Query2`) si applica direttamente allo stream in uscita da `Query1`. Appena la finestra riceve 3 eventi con lo stesso `tileId`, viene creata una `PriorityQueue<PointWithDeviation>` con `initialCapacity=5` per la classifica. A questo punto sull'ultimo layer:

1) Si scartano i punti marcati come `false` nella maschera booleana `validMask` (generata in `Query1`).
2) Per ciascun punto valido del tile superiore, si compilano due liste:
   - `nearestNeighborTemps`, raccogliendo i valori di temperatura dei vicini a Manhattan $d \leq 2$;
   - `externalNeighborTemps`, raccogliendo i valori di temperatura dei vicini a Manhattan $2 < d \leq 4$.
3) Si calcolano la media dei due insiemi e la deviazione locale $\delta_{\text{loc}} = \left|\overline{T}_{\text{near}} - \overline{T}_{\text{ext}}\right|$.
4) Se $\delta_{\text{loc}} > 6000$, il punto diventa un `Outlier` e viene aggiunto alla lista degli outlier da restituire.
5) Successivamente si aggiorna la `PriorityQueue` dei top 5 utilizzando i metodi `poll()` (per rimuovere il minimo) e `add()` (per inserire il nuovo punto).

L'output di **Query 2** viene infine convertito in CSV e salvato su file tramite `FileSink`,

`seq_id, print_id, tile_id, P1,` $\delta P1$`, P2,` $\delta P2$`, P3,` $\delta P3$`, P4,` $\delta P4$`, P5,` $\delta P5$

senza inviare nulla al `Local Challenger`.

VI. Query 3

La **Query 3** realizza lo stadio 4 della pipeline: prende in input i `TileWithOutliers` prodotti dalla `Query 2` e

raggruppa gli outlier tramite l'algoritmo DBSCAN (Density-Based Spatial Clustering of Applications with Noise), basato sulla distanza euclidea.

Definiamo innanzitutto l'algoritmo DBSCAN. Esso prende in input due parametri:

- $\epsilon = 20$: raggio di vicinanza;
- `minPts` $= 4$: numero minimo di punti richiesti per formare una regione densa.

Il concetto fondamentale alla base di DBSCAN è quello di ϵ-vicinato di un punto p, definito come:

$$N_\epsilon(p) = \{q \in D \mid d(p,q) \leq \epsilon\}$$

dove D è il dataset e d è la metrica di distanza (in questo caso, la distanza euclidea).

Un punto p è detto *punto core* se:

$$|N_\epsilon(p)| \geq \texttt{minPts}$$

Un punto $p \in D$ è invece classificato come **rumore** se non è densità-raggiungibile da nessun punto *core*, ovvero:

$$\forall q \in D, \quad p \notin \text{cluster}(q)$$

dove `cluster(q)` è l'insieme dei punti densità-connessi con q.
Nella figura 4 è possibile vedere un esempio di applicazione dell'algoritmo DBSCAN con minPts pari a 3.

Per l'implementazione di DBSCAN è stata utilizzata la libreria Java SMILE. Per ogni tile:

1) Si estrae la lista di `outliers` e la si trasforma in un array `double[][] data`, dove ogni riga è la coppia (x, y) di un outlier.
2) Si applica `DBSCAN.fit(data, minPts, ϵ)`, ottenendo:
   - un array `labels` con valore -1 per i *noise points* e un indice di cluster ≥ 0 per gli altri punti;
   - il numero di cluster trovati `k`.
3) Per ciascun cluster $c = 0, \ldots, k-1$:
   - si accumulano le coordinate (x, y) dei punti etichettati c;
   - si calcola il *centroide* dividendo per la dimensione del cluster;
   - si crea un oggetto `Centroid(cx, cy, size)`.
4) Si costruisce un `TileClusterResult` contenente:

   `batchId, printId, tileId,`
   `saturated, lista_dei_centroidi`

   e lo si emette nello stream di output.

In questo modo, a valle della Query 3 ogni tile è accompagnato dalla rappresentazione spaziale dei cluster di outlier, tramite i loro centroidi e le rispettive dimensioni. Il risultato finale di questa query viene poi inviato al `Local Challenger`.

VII. Risultati

In questa sezione presentiamo e discutiamo i risultati ottenuti variando il numero di slot in Flink tra i valori 1, 4, 6, 8 e 16. Per ciascuna configurazione abbiamo misurato il throughput e la latenza media del sistema, come illustrato nelle Figure 2 e 3.

I primi risultati evidenziano la differenza tra la versione Python non ottimizzata e l'implementazione in Flink: il throughput della soluzione Python è di circa 0.6 eventi/s, mentre Flink raggiunge già 19 eventi/s con un singolo slot. Analogamente, la latenza media è pari a 695 ms nella versione Python rispetto ai 103 ms di Flink con un solo slot.

Con l'aumentare del parallelismo, il throughput cresce in modo importante fino ai 4 slot, si stabilizza invece tra i 4 e 16 slot. Al contrario, la latenza media aumenta progressivamente al crescere del numero di slot.

Questo comportamento è dovuto principalmente al data skew: sebbene il carico di eventi risulti uniforme per chiave, il partizionamento keyBy di Flink assegna le chiavi ai Key Group tramite MurmurHash, causando, nel nostro caso, una distribuzione sbilanciata tra i subtask. Di conseguenza, alcuni subtask ricevono più eventi e diventano colli di bottiglia, rallentando l'elaborazione complessiva. L'overhead aggiuntivo di gestione di più task paralleli contribuisce ulteriormente all'aumento della latenza.

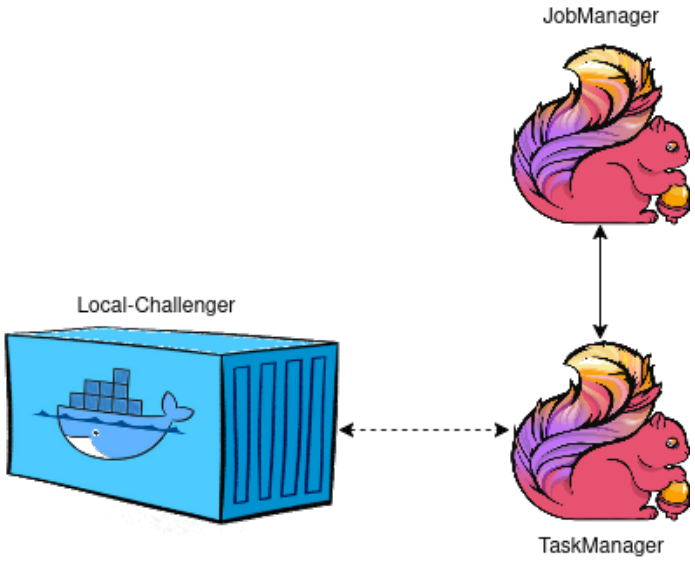

Fig. 1. Pipeline

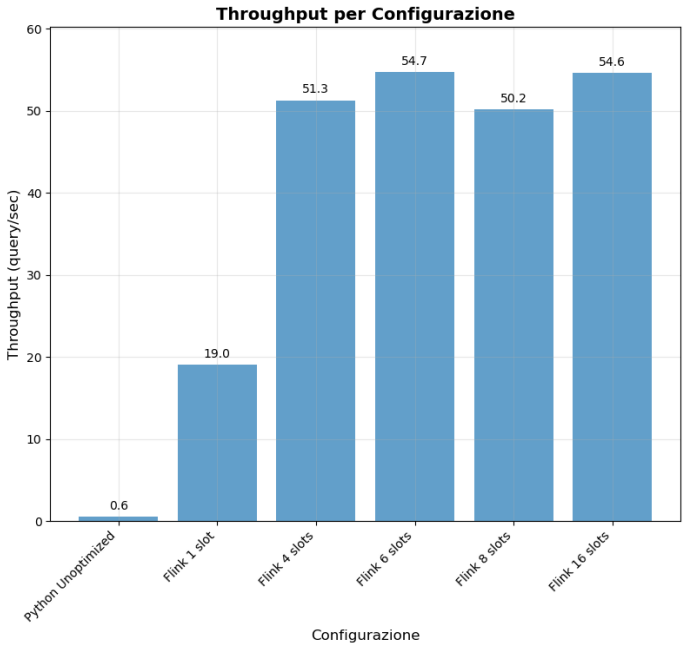

Fig. 2. Throughput in funzione del numero di slot

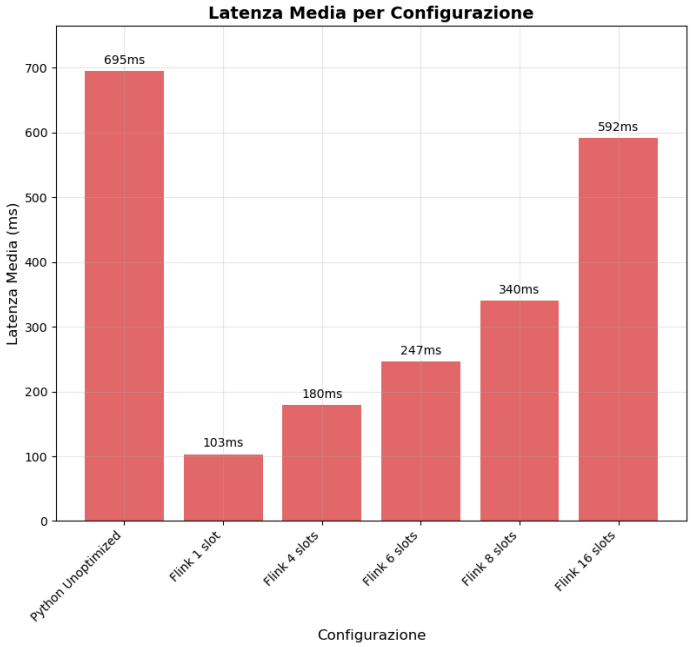

Fig. 3. Latenza media in funzione del numero di slot

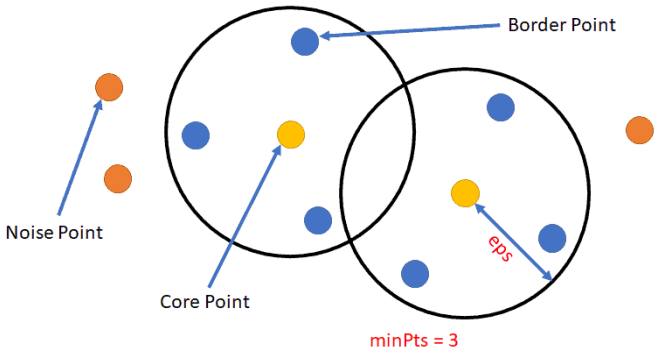

Fig. 4. DBSCAN