Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Realizzazione di un algoritmo bio-ispirato per il clustering di stream di dati evolventi su architettura GPU

Giandomenico Spezzano,
Andrea Vinci

RT-ICAR-CS-15-03 **Marzo 2015**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.icar.cnr.it*

Sommario

In questo documento è dettagliata la realizzazione su architettura di tipo GPU di una algoritmo di swarm intelligence per il clustering di data stream evolventi. Nel documento, è descritta una variante dell'algoritmo Flockstream, adattato per l'esecuzione su architetture SIMD (Single Instruction Multiple Data). L'algoritmo è stato realizzato utilizzando la piattaforma NVIDIA CUDA. Per mostrare la validità dell'approccio, sono mostrati degli esperimenti eseguiti su dei dataset sintetici.

Indice

INTRODUZIONE 2

ALGORITMI DI CLUSTERING PER FLUSSI DI DATI EVOLVENTI 3
CONTENUTI 5

1. FLOCKSTREAM: UN ALGORITMO BIO-ISPIRATO PER IL CLUSTERING DI FLUSSI DI DATI EVOLVENTI 6

DENSTREAM 7
MULTIPLE SPECIES FLOCKING 10
FLOCKSTREAM 14

2. GPU COMPUTING 18

CUDA 21

3. FLOCKSTREAM SU GPU 26

IL MODELLO UTILIZZATO 26
L'IMPLEMENTAZIONE 31

4. RISULTATI SPERIMENTALI 42

CRITERI DI VALUTAZIONE 43
PRIMO GRUPPO DI TEST 44
SECONDO GRUPPO DI TEST 49

5. RIFERIMENTI 59

Introduzione

Il progetto ResNovae prevede che le principale vie della città di Cosenza siano coperte dalla piattaforma per Cyber-Physical System denominata Rainbow, sviluppata all'interno del progetto. Essa, ricordiamo, è composta da tre livelli (Figura 1):

- Il livello fisico, composto da dispositivi di sensing ed attuazione direttamente immersi nell'ambiente fisico da controllare.
- Un livello composto da una rete di nodi di computazione, anche loro posti nell'ambiente, ai quali sono direttamente connessi i dispositivi di sensing ed attuazione del livello precedente. A questo livello sono preposte le attività di controllo dell'ambiente, in un ciclo sensing-elaborazione-attuazione.

  Un singolo nodo computazionale ospita un Virtual Object Gateway, che gestisce un insieme di Virtual Object (VO). I VO sono delle astrazioni software dei dispositivi fisici, che espongono funzioni definite secondo una interfaccia ben specificata.

  Oltre al Gateway, i nodi di computazione ospitano un server per l'esecuzione di un sistema multi-agente, cuore dell'elaborazione in Rainbow. Applicazioni ad agenti possono essere dinamicamente caricate sulla rete di nodi, per eseguire in tempo reale elaborazioni e controllo utilizzando direttamente i dispositivi fisici, attraverso i VO.
- Un livello di cloud computing, al quale sono preposte attività che non possono essere ben svolte nel livello intermedio, quali elaborazioni che richiedono conoscenza completa del sistema, elevate risorse computazionali e di memoria, dati storici sul lungo periodo. Gli agenti possono sfruttare i risultati delle elaborazioni svolte a questo livello per svolgere al meglio i loro compiti. A questo livello sono anche delegate le attività di presentazione e visualizzazione dello stato e delle funzionalità del sistema.

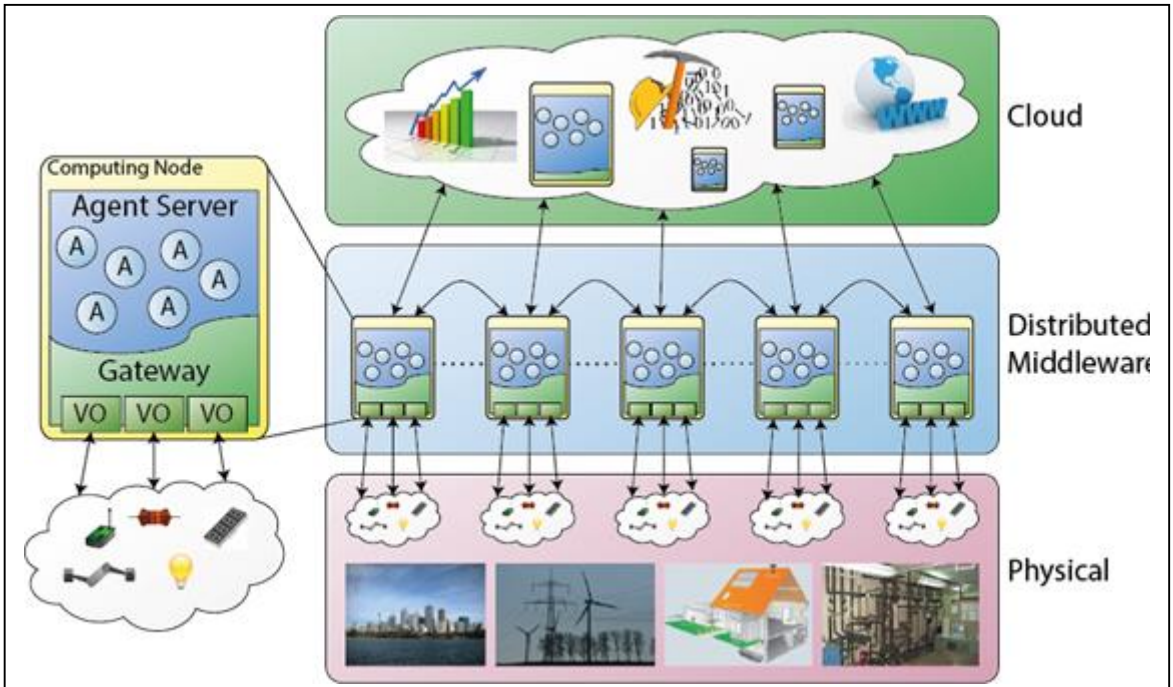


Figura 1. L'architettura Rainbow

Di seguito presenteremo una algoritmo di clustering di flussi di dati evolventi, quali sono quelli prodotti dai sensori e dagli agenti della piattaforma Rainbow. L'approccio è pensato per eseguire logicamente sul livello "Cloud" della piattaforma, poiché ha bisogno, durante la sua esecuzione, di mantenere una sintesi dei dati storici il cui aggiornamento avverrà man mano che nuovi dati vengono prodotti. L'algoritmo esegue allo scopo di poter identificare rapidamente ed in tempo reale cluster e pattern che emergano dai flussi di dati, e quindi intervenire tempestivamente all'occorrenza.

Individuare pattern e cluster di interesse è inoltre alla base della progettazione di agenti in grado di riconoscere situazioni di interesse direttamente "in network", nel livello intermedio di Rainbow, e da lì eseguire attuazioni se previsto.

Algoritmi di clustering per flussi di dati evolventi

Con l'avvenuta informatizzazione di gran parte delle attività umane, la quantità di dati digitalmente disponibili è andata aumentando vertiginosamente negli ultimi anni. Si sono resi necessari strumenti per l'analisi di grandi moli di dati per poter estrarre conoscenza, utile in settori che spaziano dalle indagini di mercato alle ricerche scientifiche.

La ricerca nell'ambito del data mining si è sviluppata proprio per rispondere a questa esigenza, producendo tecniche e metodologie in grado di assolvere a diverse tipologie di compiti. Il presente lavoro riguarda il task di clustering, cioè la scoperta di gruppi di oggetti il più possibile simili tra loro e dissimili dagli oggetti appartenenti ad altri gruppi. Considerando, ad esempio, una base di dati che descriva gli acquisti di ogni cliente presso un certo supermercato è possibile categorizzare questi ultimi per proporre ad ogni diverso gruppo di clienti una campagna di marketing personalizzata, e quindi più efficace.

Oltre a contesti abbastanza *statici*, come quello descritto nel precedente esempio, nei quali l'analisi dei cluster viene svolta su una base di dati con una bassa velocità di aggiornamento e rinnovo delle informazioni, vi sono altri contesti nei quali i dati vengono prodotti a ritmi molto elevati, come quelli generati da reti di sensori su Smart City, sensori atmosferici, osservazioni astronomiche, transazioni bancarie, flussi di click su siti web. In questi contesti si parla di *flussi di dati evolventi (evolving data streams)*. Per adempiere al task di clustering su flussi dati è necessario affrontare vincoli di *scalabilità* e *responsività* più restrittivi rispetto al contesto statico poiché spesso si analizzano dati che diventano obsoleti in breve tempo: se il flusso di dati riguarda la veloce evoluzione di un incendio, e l'analisi è necessaria per direzionare al meglio gli sforzi dei soccorritori, è desiderabile avere buoni risultati, anche parziali, nel minor tempo possibile.

Gli algoritmi *anytime* sono in grado di rispondere efficacemente alle problematiche presentate. Questi sono una tipologia di algoritmi in grado di fornire una buona soluzione (approssimata) anche se interrotti durante l'elaborazione. L'idea è che essi possano fornire un risultato in ogni momento, che è raffinato a mano a mano che l'elaborazione procede, convergendo eventualmente alla soluzione ottimale.

In questo contesto, sono state proposte soluzioni basate su *swarm intelligence*. Queste sono ispirate dal comportamento di alcune specie viventi, quali formiche, api, uccelli. In ognuno di questi casi, è possibile notare come, nonostante le capacità limitate di ogni individuo, il comportamento globale di un gruppo numeroso mostra capacità auto-organizzanti: le formiche sono in grado di ricercare efficacemente il percorso più breve per una fonte di cibo, gli uccelli sono in grado di formare stormi e muovere come fossero un'unica entità. Esempi di algoritmi ispirati dal comportamento delle formiche si possono trovare in [17][18], dove sono presentati, rispettivamente, un algoritmo per il clustering gerarchico ed uno per la costruzione di un sistema di informazione peer-to-peer su griglie.

Gli approcci basati su swarm intelligence presuppongono che ogni individuo disponga di *capacità limitate* e che *non conosca lo stato globale* di un sistema. È inoltre assente un ente unico coordinatore. Per queste peculiarità gli algoritmi basati su swarm intelligence sono *decentralizzati* e quindi altamente *paralleli*.

Potenti processori paralleli ormai presenti anche in ogni computer desktop sono quelli grafici. L'evoluzione tecnologica di queste architetture ha avuto negli ultimi anni un trend estremamente positivo. Per effetto di una concorrenza serrata, grandi aziende quali Nvidia e AMD/ATI, che da tempo producono soluzioni GPU per utenza domestica e professionale, stanno proponendo soluzioni innovative affinché le tecnologie parallele sviluppate per il calcolo grafico possano essere sfruttate anche per il calcolo non grafico. In particolare, Nvidia sta investendo molto nella sua piattaforma di GPU computing CUDA (Compute Unified Device Architecture), mentre AMD investe nell'integrazione di CPU e GPU, magari sullo stesso chip e con memoria condivisa.

Viste le caratteristiche degli algoritmi basati su swarm intelligence, è più che naturale ricercarne un'implementazione su GPU. Qui di seguito si propone un algoritmo *anytime* per il *clustering* di *flussi di dati evolventi* basato su *swarm intelligence* per architettura *GPU*.

Contenuti

Nelle sezioni seguenti, si presenterà per prima cosa l'algoritmo FlockStream, algoritmo anytime per il clustering di flussi di dati evolventi ispirato dal comportamento degli uccelli. Per arrivare a FLockstream, si introdurranno altri due algoritmi, dei quali FlockStream è l'evoluzione:

1. DenStream, un algoritmo per il clustering di flussi dati evolventi basato su densità, del quale principali limiti sono la centralizzazione dei dati e l'esecuzione di una fase offline per il clustering dei dati, che ne riduce la responsività.
2. Multiple Species Flocking Clustering, un algoritmo per il clustering di dati statici, basato su densità, *anytime* e ispirato dal comportamento degli uccelli.

Successivamente si effettuerà una breve analisi dell'architettura GPU e si mostrerà una implementazione di FLockstream su GPU. Infine si presenteranno i risultati sperimentali su dataset sintetici.

1. Flockstream: un algoritmo bio-ispirato per il clustering di flussi di dati evolventi.

In questa sezione è presentato un algoritmo parallelo per il clustering di flussi di dati evolventi basato su swarm intelligence.

Per prima cosa verrà discusso DenStream, un algoritmo per il clustering di flussi dati evolventi basato su densità, che analizza i nuovi dati resi disponibili dal flusso mantenendo un *modello* dei punti precedentemente analizzati. Nozione chiave alla base di questo algoritmo è quella di *micro-cluster*, un oggetto sintetico che rappresenta approssimativamente un gruppo di punti nello spazio della feature. Il modello costruito avrà quindi la forma di un insieme di *micro-cluster*.

DenStream esegue in due fasi: una prima fase online, nella quale si esegue l'aggiornamento del modello analizzando i nuovi punti in arrivo dal flusso dati, effettuando per ognuno di essi una ricerca esaustiva del più vicino micro-cluster presente nel modello, eventualmente aggregando il punto al micro-cluster o formandone uno nuovo; ed una seconda fase offline, nella quale si esegue il clustering dei micro-cluster del modello, trattandoli come fossero punti virtuali di un dataset statico, attraverso una variante della più nota tecnica DBScan [2].

Si mostrerà che l'algoritmo risulta essere centralizzato e poco scalabile, poiché la prima fase richiede una ricerca esaustiva su tutti i punti del modello. Inoltre, la necessità di eseguire spesso la fase offline, la più impegnativa computazionalmente, per poter seguire l'evoluzione ed il cambiamento dei cluster all'interno del flusso dati ne limita la responsività.

Nella sezione seguente verrà trattato, Multiple Species Flocking Clustering (MSFC) [3], un algoritmo per il clustering di dati statici, ispirato al comportamento degli uccelli.

In natura, si osserva come alcune specie di uccelli siano in grado di raggrupparsi e formare stormi, anche di grandi dimensioni. In [4] Reynolds formalizza matematicamente questo comportamento con il suo modello di Flocking, che consta di tre semplici regole, applicate autonomamente da ogni agente.

Se si considera uno spazio con più specie di uccelli, gli individui di specie diverse formeranno stormi differenti, comportamento che costituisce un esempio naturale di clustering. MSFC sfrutta una variante del modello di Reynolds, associa ognuna delle tuple di un dataset ad un singolo agente. Gli agenti muovono in uno spazio virtuale bidimensionale, agendo autonomamente per aggregarsi con altri agenti riconosciuti come appartenenti alla stessa specie, secondo una funzione di similitudine definita nello spazio delle feature del dataset.

MSFC gode di tre proprietà rilevanti: è decentralizzato, scala bene, poiché ogni agente autonomo agisce secondo informazioni locali e non esiste un'entità che conserva lo stato globale del sistema, ed è *anytime*, ovvero capace di fornire una soluzione approssimativa in qualunque momento, soluzione che viene raffinata continuando ad elaborare, convergendo all'ottimo.

Dopo aver introdotto DenStream ed MSFC, presenteremo finalmente un algoritmo innovativo per il clustering di flussi di dati evolventi che dai precedenti è ispirato. Come DenStream, il nuovo algoritmo mantiene un modello sintetico della porzione del flusso di dati già analizzata, e lo sfrutta per calcolare il clustering dei nuovi punti. Come MSFC è basato sul modello di Flocking di Reynolds, ereditandone le proprietà di scalabilità e responsività

Ogni punto del flusso dati da analizzare è associato ad un agente, come anche ogni punto (micro-cluster) del modello. Questi muoveranno insieme nello stesso spazio virtuale bidimensionale, aggregandosi per formare stormi (ovvero cluster). In qualunque momento è possibile visualizzare il risultato parziale approssimato di questa aggregazione. Quando il flusso di dati rende disponibile un nuovo gruppo di punti, il modello è aggiornato ed i punti del passo precedente aggregati al modello. Si prosegue quindi con l'analisi del nuovo gruppo. L'aggiornamento del modello sfrutta un insieme di dati parziali calcolati nel corso dell'elaborazione *online* del flocking.

Il nuovo algoritmo, chiamato FlockStream, gode quindi dei benefici di MSFC, è tollerante al rumore (come si mostrerà sperimentalmente) ed è in grado di restituire un risultato parziale in qualunque momento. Basandosi su agenti autonomi, si candida naturalmente ad un'implementazione parallela.

DenStream

DenStream [1], è un algoritmo per il clustering basato su densità di flussi di dati evolventi che mantiene un *modello* dei punti del flusso precedentemente analizzati. Nozione chiave alla base di questo algoritmo è quella di *micro-cluster*, un oggetto sintetico che rappresenta approssimativamente un gruppo di punti nello spazio delle feature. Il modello costruito avrà quindi la forma di un insieme di *micro-cluster*.

DenStream esegue in due fasi: una prima fase online, nella quale si esegue l'aggiornamento del modello analizzando i nuovi punti in arrivo dal flusso dati, effettuando per ognuno di essi una ricerca esaustiva del più vicino micro-cluster presente nel modello, eventualmente aggregando il punto al micro-cluster o formandone uno nuovo; ed una seconda fase offline, nella quale si esegue il clustering dei micro-cluster del modello, trattandoli come fossero punti virtuali di un dataset statico, attraverso una variante della più nota tecnica DBScan [2].

Una dei limiti di questo schema a due fasi è la necessità di eseguire spesso la fase offline, la più impegnativa computazionalmente, per poter seguire l'evoluzione ed il cambiamento dei cluster all'interno del flusso dati.

L'algoritmo assume un modello a finestre smorzate (dumped windows), nel quale il peso di ogni tupla del flusso dati decresce esponenzialmente col tempo t secondo una funzione di fading esponenziale $f(t) = e^{-\lambda t}$ con $\lambda > 0$. Il peso totale del flusso è la costante $W = \frac{v}{1-2^{-\lambda}}$, dove v è la costante di velocità del flusso, ovvero il numero di punti che arrivano in elaborazione nell'unità di tempo. La funzione di fading fa sì che l'importanza dei dati storici decresca in funzione del tempo e all'aumentare del coefficiente dell'esponenziale λ.

In DenStream, il concetto di core point presente nel DBSCAN [2] viene esteso e viene introdotta la nozione di micro-cluster, che conserva una rappresentazione approssimata di un gruppo di data point. In DBScan, un core point è un oggetto che ha almeno ε vicini, la cui somma dei pesi sia almeno pari a μ. Un clustering è un insieme di core point che hanno la stessa etichetta di cluster.

Si introducono tre nozioni di micro-cluster: i core-micro-cluster (c-micro-cluster), i potential core-micro-cluster (p-micro-cluster) e gli outlier micro cluster (o-micro-cluster).

Un *core-micro-cluster* al tempo t per un gruppo di punti vicini $p_{i_1}, \dots, p_{i_n}$con timestamps $T_{i_1}, \dots, T_{i_n}$ è definito come $CMC(w, c, r)$ con w peso , c centro e r raggio del c-micro-cluster. Il peso $w = \sum_{j=1}^{n} f(t - T_{i_j})$ deve essere tale che $w \leq \mu$. Il centro è definito come:

$$c = \frac{\sum_{j=1}^{n} f\left(t - T_{i_j}\right) p_{i_j}}{w}$$

E il raggio come:

$$r = \frac{\sum_{j=1}^{n} f\left(t - T_{i_j}\right) dist(p_{i_j}, c)}{w}$$

Con $r \leq \varepsilon$. $dist(p_{i_j}, c)$ è la distanza euclidea tra il punto p_{i_j} e il centro c. Si noti che il peso di un micro-cluster deve essere maggiore di una soglia predefinita μ per poter essere considerato un core-micro-cluster. Un micro-cluster può comunque variare secondo i dati nel flusso, contentendone un calcolo incrementale.

Un c-micro-cluster potenziale (p-micro-cluster) al tempo t per un gruppo di punti tra loro vicini $p_{i_1}, \dots, p_{i_n}$con timestamps $T_{i_1}, \dots, T_{i_n}$ è definito come $\{\overline{CF^1}, \overline{CF^2}, w\}$, dove il peso w deve essere tale che $w \geq \beta\mu$ con $0 < \beta \leq 1$. β è un parametro che definisce una soglia che distingua se il micro-cluster possa essere considerato o meno come outlier. $\overline{CF^1} = \sum_{j=1}^{n} f\left(t - T_{i_j}\right) p_{i_j}$ è la somma lineare dei pesi dei punti, mentre $\overline{CF^2} = \sum_{j=1}^{n} f\left(t - T_{i_j}\right) p_{i_j}^2$ è la somma pesata dei quadrati dei punti, inteso come quadrato del modulo del vettore posizione. Il centro di un p-micro-cluster è $c = \frac{\overline{CF_1}}{w}$ ed il suo raggio $r < \epsilon$ è $r = \sqrt{\frac{\overline{CF^2}}{w} - \left(\frac{\overline{CF^1}}{w}\right)^2}$.

Un p-micro-cluster è quindi un insieme di punti che potrebbe diventare un micro-cluster.

Un outlier micro-cluster (o-micro-cluster) al tempo t per un gruppo di punti tra loro vicini $p_{i_1}, \dots, p_{i_n}$con timestamps $T_{i_1}, \dots, T_{i_n}$ è definito come $\{\overline{CF^1}, \overline{CF^2}, w, t_0\}$. Le definizioni di $\overline{CF^1}, \overline{CF^2}, w$, centro e raggio sono le stesse del p-micro-cluster, mentre $t_0 = T_{i_j}$ rappresenta l'istante di creazione dell'o-micro-cluster. In un o-micro-cluster il peso w deve essere inferiore a una soglia fissata, quindi $w < \beta\mu$. Può comunque crescere e diventare un p-micro-cluster se, aggiungendo nuovi punti, il pese supera la soglia.

L'algoritmo consta di due fasi: la fase online, nella quale i micro-cluster sono mantenuti e aggiornati a mano a mano che nuovi punti arrivano dal flusso; la fase offline, nella quale i cluster

sono finalmente generati, a richiesta, dall'utente. Durante la fase online, quando un nuovo punto p è pronto, DenStream prova ad unirlo al più vicino micro-cluster c_p, soltanto se il raggio r_p di c_p non aumenta oltre ε, ovvero deve valere $r_p < \varepsilon$. Se questo vincolo non è soddisfatto, l'algoritmo tenta di aggregare il punto al più vicino o-micro-cluster c_o, a patto che $r_o < \varepsilon$. Si controlla quindi che il peso w deve sia tale che $w \geq \beta\mu$ ed in tal caso c_o viene promosso in un p-micro-cluster. Se non si verifica nessuna delle precedenti, un nuovo o-micro-cluster è generato dal punto p. Si noti che il peso di ogni p-micro-cluster esistente andrà a decadere gradualmente, secondo la funzione di fading, e, se non alimentato da nuovi punti, potrà essere declassata a o-micro-cluster, qualora il peso sia inferiore a $\beta\mu$.

La fase offline usa una variante dell'algoritmo DBSCAN, nella quale i p-micro-cluster sono considerati come punti virtuali. I concetti di densità-raggiungibile e densità-connesso sono usati da DenStream per generare il risultato finale. DenStream non può essere usato per quantità di dati troppo grandi, come quelle prodotte da reti di larga scala di sorgenti autonome, poiché è necessario che si calcoli il più vicino micro-cluster per ogni punto in arrivo e si assume che tutti i dati siano disponibili nella stessa locazione dove l'algoritmo è eseguito.

Caratteristiche salienti di DenStream sono quindi:

- **Riconoscimento di cluster di qualunque forma.** L'algoritmo si comporta come un algoritmo di clustering basato su densità.
- **Nessuna assunzione sul numero dei cluster.** A differenza di altri algoritmi, quali il k-means, non è necessario definire in input il numero di cluster attesi.
- **Tollerante al rumore.** Con opportune piccole modifiche implementative, si mostra che l'algoritmo è resistente al rumore.

- **Poco scalabile e centralizzato.** Il calcolo prevede, all'arrivo di ogni nuovo punto, un confronto con ogni micro-cluster presente nel modello.
- **Due fasi, online ed offline.** L'algoritmo richiede, per rispondere, una fase offline on-demand in cui viene eseguito un DBSCAN sui dati.

La bassa scalabilità, la centralizzazione e la bassa responsività dovuta alla fase offline sono problematiche risolte negli approcci basati su swarm intelligence.

Di seguito mostreremo l'algoritmo Multiple Species Flocking Clustering, decentralizzato e con alta scalabilità, ma non applicabile a flussi di dati evolventi.

Multiple Species Flocking

Multiple Species Flocking Clustering (MSFC) [3] è un algoritmo per il clustering di dati statici, ispirato al comportamento degli uccelli.

In natura, si osserva come alcune specie di uccelli siano in grado di raggrupparsi e formare stormi, anche di grandi dimensioni. In [4] Reynolds formalizza matematicamente questo comportamento con il suo modello di Flocking, che consta di tre semplici regole, applicate autonomamente da ogni agente.

Se si considera uno spazio con più specie di uccelli, gli individui di specie diverse formeranno stormi differenti, comportamento che costituisce un esempio naturale di clustering. MSFC sfrutta una variante del modello di Reynolds, associa ognuna delle tuple di un dataset ad un singolo agente. Gli agenti muovono in uno spazio virtuale bidimensionale, agendo autonomamente per aggregarsi con altri agenti riconosciuti come appartenenti alla stessa specie, secondo una funzione di similitudine definita nello spazio delle feature del dataset.

MSFC gode di due proprietà interessanti: è decentralizzato, poiché ogni agente autonomo agisce secondo informazioni locali e non esiste un'entità che conserva lo stato globale del sistema, ed è *anytime*, ovvero capace di fornire una soluzione approssimata in qualunque momento, soluzione che viene raffinata continuando ad elaborare, convergendo all'ottimo.

Introduciamo ora il modello di Reynolds alla base di MSFC. Ogni individuo è caratterizzato da un vettore posizione p ed un vettore velocità v nello spazio. Sono presenti anche due distanze $D1 > D2$, la prima indicante l'orizzonte di visibilità dell'individuo, la seconda una "distanza di guardia" che l'individuo tenterà di mantenere con ogni altro.

Sia b l'individuo (boid) in considerazione, ed x un altro generico individuo, ad ogni iterazione l'algoritmo calcolerà la nuova velocità dell'agente aggregandone le componenti di seguito presentate:

- **Allineamento:**

$$v_{al_b} = \frac{1}{|\Omega_b|} \sum_{x \epsilon \Omega_b} v_x \,, \qquad con\ \Omega_b = \{x \mid D2 < dist(p_b, p_x) \leq D1\}$$

  Ogni individuo tenderà a muoversi nella stessa direzione dei suoi vicini, secondo una semplice media aritmetica delle velocità (Figura 1.1).

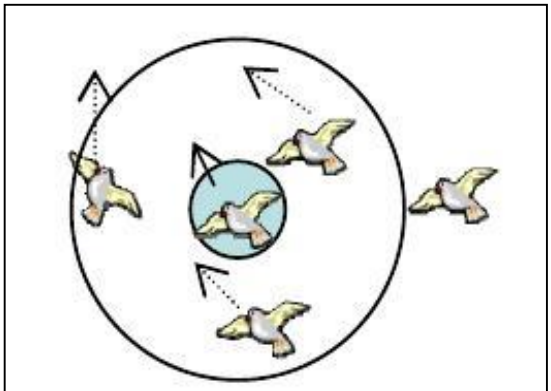


Figura1.1: Allineamento

- **Separazione:**

$$v_{se_b} = \sum_{x \epsilon \Omega_b} \frac{-(v_x + v_b)}{dist(p_b, p_x)}, \qquad con\ \Omega_b = \{x \mid dist(p_b, p_x) \leq D2\}$$

Individui troppo vicini tenderanno a prendere le distanze, fino a raggiungere almeno la "distanza di guardia" D2, con forza inversamente proporzionale alla distanza (Figura 1.2).

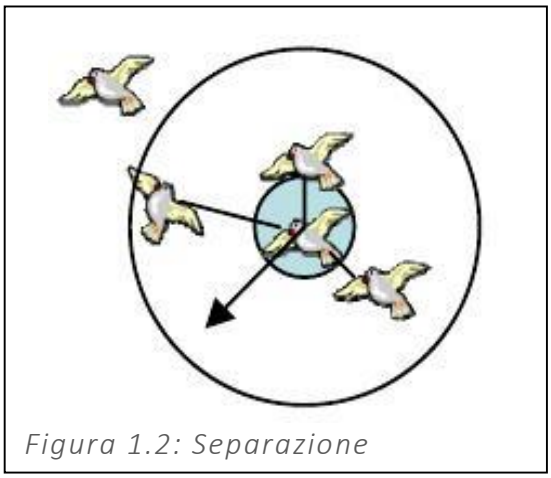


Figura 1.2: Separazione

- **Coesione:**

$$v_{co_b} = \sum_{x \epsilon \Omega_b} (p_x - p_b), \qquad con\ \Omega_b = \{x \mid D2 < dist(p_b, p_x) \leq D1\}$$

Gli individui tenderanno ad avvicinarsi tra loro fino al raggiungimento della distanza di guardia D2, con forza proporzionale alla distanza (Figura 1.3).

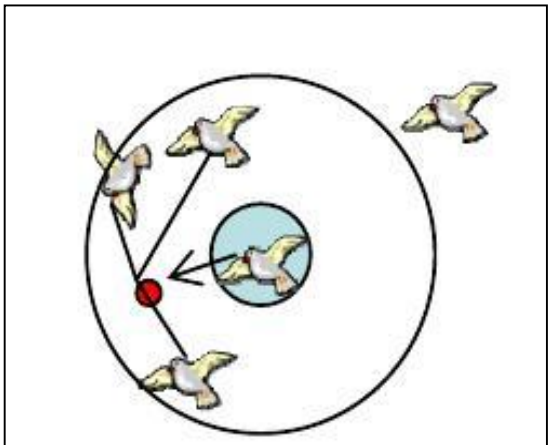


Figura 1.3: Coesione

Per ogni iterazione, la velocità di ogni agente è aggiornata secondo una somma pesata delle componenti calcolate.

$$\forall b, \qquad v_b = w_{al} \cdot v_{al_b} + w_{se} \cdot v_{se_b} + w_{co} \cdot v_{co_b}$$

La posizione di ogni agente è quindi modificata in funzione della velocità.

Attraverso l'applicazione delle regole mostrate, si simula il comportamento di formazione e movimento di stormi di uccelli, e più in generale di branchi di animali.

Estensione per clustering. Estendiamo questo modello per poter affrontare un task di clustering. Ad ogni tupla del dataset che si vuole elaborare associamo un diverso individuo. Esso sarà quindi caratterizzato, oltre che da velocità e posizione nello spazio di movimento degli individui, che chiameremo *spazio virtuale dei boid*, anche da un vettore f contenente informazioni nello *spazio delle feature* tratte dal dataset.

È necessario distinguere quindi due funzioni di distanza diverse: la prima, $dist_p(\cdot,\cdot)$, valida per calcolare la distanza di due punti sullo spazio virtuale dei boid; la seconda, $dist_f(\cdot,\cdot)$, che calcoli la distanza nello spazio delle feature. Assumiamo, come nel precedente caso, $dis_p(\cdot,\cdot)$ distanza euclidea, e non facciamo alcuna assunzione su $dist_f(\cdot,\cdot)$.

Introdotta una soglia di similarità ε, le regole precedentemente riportate sono modificate in modo da considerare anche la similarità delle feature:

- **Allineamento:**

$$v_{al_b} = \frac{1}{|\Omega_b|} \sum_{x \epsilon \Omega_b} v_x \,,$$

$$con\ \Omega_b = \{x \mid D2 < dist_p(p_b, p_x) \leq D1 \wedge dist_f(f_b, f_x) < \varepsilon\}$$

  la regola è modificata in modo da considerare i soli agenti vicini *simili*.

- **Separazione:**

$$v_{se_b} = \sum_{x \epsilon \Omega_b} \frac{-(v_x + v_b)}{dist(p_b, p_x)}, \qquad con\ \Omega_b = \{x \mid dist_p(p_b, p_x) \leq D2\}$$

  La regola è identica a come riportata in precedenza.

- **Coesione:**

$$v_{co_b} = \sum_{x \epsilon \Omega_b} (p_x - p_b) \cdot \left(1 - \frac{dist_f(f_b, f_x)}{\varepsilon}\right),$$

$$con\ \Omega_b = \{x \mid D2 < dist_p(p_b, p_x) \leq D1 \wedge dist_f(f_b, f_x) < \varepsilon\}$$

  La regola è modificata in modo da considerare i soli vicini simili; inoltre la componente apportata da ognuno dei vicini sarà minore quanto più distanti saranno dall'agente considerato.

È inoltre introdotta una nuova regola:

- **Dissimilarità** sulle feature:

$$v_{di_b} = -\sum_{x \epsilon \Omega_b} (p_x - p_b) \cdot \frac{dist_f(f_b, f_x)}{\varepsilon}, \qquad con\ \Omega_b = \{x \mid dist_p(p_b, p_x) \leq D1 \wedge dist_f(f_b, f_x) > \varepsilon\}$$

Individui dissimili tenderanno ad allontanarsi, con forza inversamente proporzionale alla distanza.

Come nel caso precedente, la velocità di ogni agente è aggiornata secondo una somma pesata delle componenti calcolate.

$$\forall b, \qquad v_b = w_{al} \cdot v_{al_b} + w_{se} \cdot v_{se_b} + w_{co} \cdot v_{co_b} + w_{di} \cdot v_{di_b}$$

L'estensione al modello di flocking appena descritta è in grado di affrontare il problema del clustering: individui simili tenderanno ad aggregarsi ed individui dissimili tenderanno ad allontanarsi: emergeranno quindi stormi differenti per ogni differente cluster.

L'esecuzione dell'algoritmo proposto presenta alcune caratteristiche rilevanti, che ritroveremo poi presentando l'algoritmo FlockStream:

- **Decentralizzazione.** Ogni individuo agisce in base ad informazioni soltanto locali, rendendolo adatto ad una implementazione distribuita.

- **Anytime e approssimato.** L'algoritmo è in grado di fornire dati parziali subito dopo un certo tempo di set-up. Proseguendo nel calcolo, l'algoritmo convergerà alla soluzione migliore.
- **Riconoscimento di cluster di qualunque forma.** L'algoritmo si comporta come un algoritmo di clustering basato su densità.
- **Nessuna assunzione sul numero dei cluster.** A differenza di altri algoritmi, quali il k-means, non è necessario definire in input il numero di cluster attesi.
- **Tollerante al rumore.** Con opportune piccole modifiche implementative, si mostra che l'algoritmo è resistente al rumore.

I lavori di Xiaohui Cui et altri [3] mostrano la validità dell'approccio nell'affrontare il problema specifico del clustering di documenti, per quanto riguarda efficacia, efficienza e decentralizzazione.

Abbiamo mostrato come MSFC risulti decentralizzato, scalabile ed anytime, proprietà assenti nel precedente algoritmo descritto, DenStream. MSFC però non può applicarsi a flussi di dati evolventi. Mostreremo ora un algoritmo innovativo per il clustering di flussi di dati evolventi che dai precedenti è ispirato.

FlockStream

Prendendo ispirazione dai due precedenti algoritmi descritti, presentiamo ora un algoritmo per il clustering di flussi di dati evolventi che come DenStream, mantiene un modello sintetico della porzione del flusso di dati già analizzata, e lo sfrutta per calcolare il clustering dei nuovi punti; come MSFC è basato sul modello di Flocking di Reynolds, ereditandono la proprietà desiderabili di scalabilità e responsività.

Il nuovo algoritmo, FlockStream[5], assume un modello a finestre smorzate (dumped windows), nel quale il peso di ogni tupla del flusso dati decresce esponenzialmente col tempo t secondo una funzione di fading esponenziale $f(t) = e^{-\lambda t}$ con $\lambda > 0$. Il peso totale del flusso è la costante $W = \frac{v}{1-2^{-\lambda}}$, dove v è la costante di velocità del flusso, ovvero il numero di punti che arrivano in elaborazione nell'unità di tempo. La funzione di fading fa sì che l'importanza dei dati storici decresca in funzione del tempo e all'aumentare del coefficiente dell'esponenziale λ.

Ogni punto dati multidimensionale è associato ad un agente, come anche ogni punto del modello. Gli agenti sono disposti su uno spazio virtuale bidimensionale di dimensioni prefissate. Nell'approccio seguito, si estendono le regole standard del modello di flocking aggiungendo la nozione di *tipo* per gli agenti. Gli agenti possono essere di tre tipi: di *base*, che rappresentano cioè un nuovo punto arrivato ad una certa unità di tempo; p-rappresentativi e o-rappresentativi, che corrispondono rispettivamente ai p-micro-cluster ed agli o-micro-cluster del DenStream, l'insieme dei quali costituisce il modello. Gli agenti muoveranno insieme nello spazio virtuale, aggregandosi per formare diversi stormi (ovvero cluster). In qualunque

momento è possibile visualizzare il risultato parziale approssimato dell'aggregazione. Quando il flusso di dati rende disponibile un nuovo gruppo di punti, il modello è aggiornato ed i punti del passo precedente aggregati al modello. Si prosegue quindi con l'analisi del nuovo gruppo. L'aggiornamento del modello è computazionalmente poco impegnativo, poiché sfrutta un insieme di dati parziali calcolati nel corso dell'elaborazione *online* del flocking.

Durante l'esecuzione, possiamo distinguere due momenti: l'inizializzazione, durante la quale è istanziato lo spazio virtuale popolandolo soltanto degli agenti di base, e, dopo la prima aggregazione ed il calcolo di un primo modello, la manutenzione dei micro-cluster e clustering vera e propria, nel quale sono presenti tutti e tre i tipi di agenti.

L'inizializzazione. Inizialmente, un insieme di agenti di base, ovvero un insieme di punti, è posta sul nello spazio virtuale. Gli agenti lavorano parallelamente per un numero di iterazioni predefinito e si muovono secondo l'euristica Multiple Species Flocking Clustering. Analogamente con gli uccelli del mondo reale, agenti che condividono vettori di feature similari si raggrupperanno insieme diventando uno stormo, mentre agenti tra loro dissimili si allontaneranno tra loro. Gli agenti usano una funzione di prossimità per identificare altri agenti a loro simili. Per come proposto, FlockStream utilizza la distanza Euclidea per misurare la dissimilarità delle feature dei due punti A e B: i due punti sono considerati simili se $d(A, B) \leq \varepsilon$, con ε rappresentante la soglia di similarità, oltreché il massimo raggio possibile per un micro-cluster. Si noti bene che anche se gli agenti si muovono su uno spazio virtuale, la similarità è calcolata nello spazio delle feature.

Proseguendo con le iterazioni, ogni agente A, varia la sua posizione P_a sullo spazio virtuale, influenzato dagli agenti X in posizione P_x all'interno del suo orizzonte di visibilità. La velocità dell'agente è calcolata secondo le leggi di Reynolds estese per considerare anche la similarità delle feature, come mostrato precedentemente. Il comportamento risultante consente anche ad un agente di muovere da un gruppo ad un altro, se il nuovo gruppo contiene agenti ad esso più simili. Emergeranno così gli stormi di oggetti simili, cioè i cluster.

Al termine del numero di iterazioni previsto per l'inizializzazione, sono sintetizzate delle statistiche e la finestra di dati è scartata. Il risultato di queste operazioni è l'introduzione sul piano virtuale dei due altri tipi di agenti, i p-rappresentativi e gli o-rappresentativi, costituenti il modello.

Manutenzione dei micro-cluster e clustering. Quando un nuovo volume di dati è pronto per essere elaborato, i rispettivi agenti sono inseriti nello spazio virtuale, ad una *velocità di stream fissa*. Si esegue la manutenzione dei p- ed o- rappresentativi ed il clustering dei nuovi punti, in un unica fase online, attraverso un numero di iterazioni predefinito. Il comportamento dei p- ed o-rappresentativi segue le regole dell'MSF, con in aggiunta le seguenti:

- Se un agente A di base incontra un secondo agente B di base, la similarità (nello spazio delle feature) viene calcolata e, se $d(A, B) \leq \varepsilon$, A e B si uniscono a formare un o-rappresentativo.

- Se un agente A di base incontra un secondo agente c_p o c_o (p- od o-rappresentativo), se i due sono simili (come nel caso precedente), A è si unisce a c_p o c_o soltanto se il raggio modificato del p- od o- rappresentativo rispetterà il vincolo $r \leq \varepsilon$. L'agente A si unisce nel senso che inizierà a muoversi nello spazio virtuale nello stesso gruppo dell'agente agente c_p o c_o, e potrà in seguito muoversi verso un altro gruppo se con esso l'agente ha più alta similarità.
- Se un agente p-rappresentativo c_p od o-rappresentativo c_o incontra un altro rappresentativo di qualunque tipo, si calcola la similarità, e se essa e minore della soglia ε i due si uniranno allo stesso gruppo, dando forma ad un cluster di rappresentativi simili.

A questo punto gli agenti aggiorneranno la loro posizione e la loro velocità applicando le regole di Reynolds, muovendosi nello spazio virtuale.

Quando il numero previsto d'iterazioni termina, sono calcolate e aggiornate le statistiche di ogni flock (cluster), il peso di ogni rappresentativo è quindi aggiornato secondo la funzione di fading, gli o-rappresentativi vengono eventualmente promossi a p-rappresentativi e i p-rappresentativi eventualmente declassati ad o-rappresentativi a seconda del loro nuovo peso, in modo identico al DenStream.

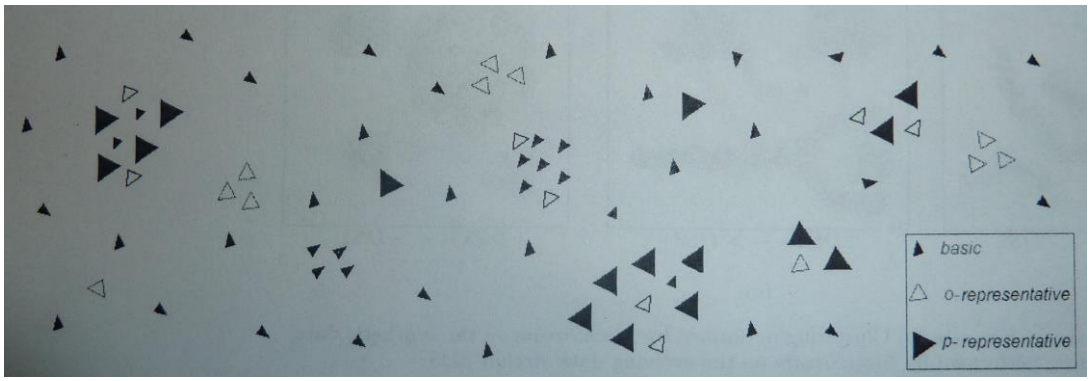


Figura 1.4: Visualizzazione di stormi di agenti e agenti isolati nello spazio virtuale in una generica iterazione.

Si noti che, a differenza di DenStream, l'algoritmo esegue in una singola fase online, poiché ogni stormo di agenti rappresenta un cluster, quindi la generazione degli stessi on demand può essere soddisfatta in qualunque momento semplicemente mostrando i gruppi calcolati fino a l'iterazione corrente.

FlockStream eredita le migliori caratteristiche dei due precedenti algoritmi:

- **Any-time e approssimato.** Come per il modello MSF, l'algoritmo è in grado di fornire dati parziali subito dopo un certo tempo di set-up. Proseguendo nel calcolo, l'algoritmo convergerà alla soluzione migliore.
- **Decentralizzato e scalabile.** Sfrutta un insieme di agenti autonomi che agiscono in base a delle sole informazioni locali.
- **Riconoscimento di cluster di qualunque forma.** L'algoritmo si comporta come un algoritmo di clustering basato su densità.
- **Nessuna assunzione sul numero dei cluster.** A differenza di altri algoritmi, quali il k-means, non è necessario definire in input il numero di cluster attesi.
- **Tollerante al rumore.** I lavori presi in considerazione mostrano che l'algoritmo è resistente al rumore.

Essendo basato su tecniche di swarm intelligence, FlockStream risulta essere implementabile su macchine parallele, quali i processori grafici. Nella prossima sezione presentiamo CUDA, una architettura per il calcolo su schede grafiche Nvidia.

2. GPU computing

Tradizionalmente i microprocessori hanno raddoppiato le performance circa ogni 18 mesi, seguendo la legge di Moore. A causa di problemi relativi alla dissipazione del calore e alla miniaturizzazione dei microchip, questo trend è venuto meno. Per continuare a migliorare le performance, si è quindi dovuto iniziare ad integrare più processori all'interno dello stesso chip. Si è entrati nell'era del multi-core: il futuro della computazione è nel parallelismo.

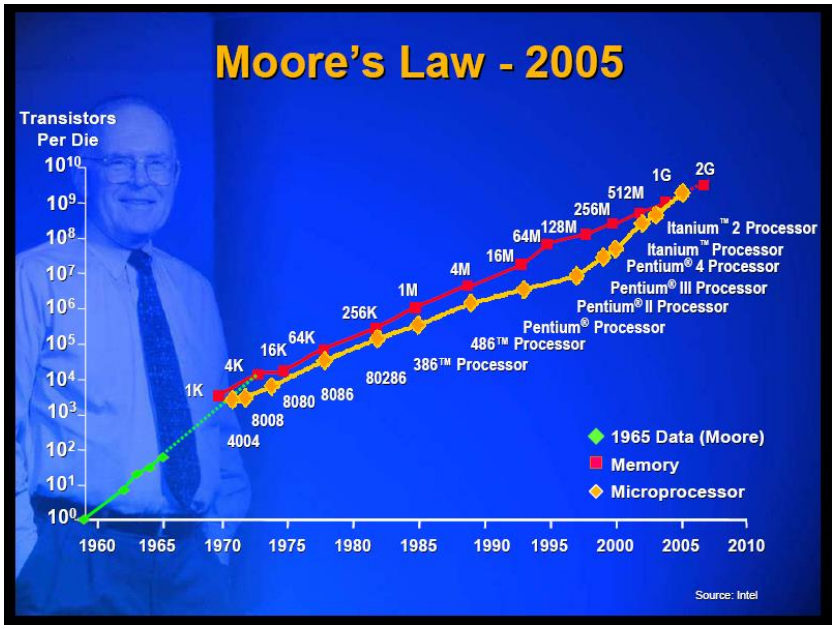


Figura 2.1: Legge di Moore

La crescita esponenziale del numero di core su un singolo chip, porterà in breve tempo a computer contenenti centinaia di core (CPU multi-core) [6].

Allo stesso modo, le GPU (Graphics Processing Unit) ad alto parallelismo stanno maturando rapidamente come potente motore per il calcolo. Esse contengono svariati elementi di calcolo (512 nella *GeForce GTX 580*) fornendo un livello di concorrenza che non si può riscontrare su nessun'altra piattaforma. Spinte da una domanda di mercato sempre crescente per applicazioni real-time di grafica 3D ad alta definizione, le GPU programmabili sono evolute in processori altamente paralleli, multi-thread e many-core, con una potenza computazionale elevata e una banda di memoria molto larga.

La ragione della discrepanza (figura 2.2) tra le capacità di calcolo floating-point delle CPU e delle GPU, è che quest'ultime sono specializzate nel calcolo intensivo ed altamente parallelo, qual è il rendering grafico, e sono state progettate in modo tale che i transistori siano principalmente dedicati al calcolo dei dati piuttosto che al chaching ed al controllo di flusso (figura 2.3).

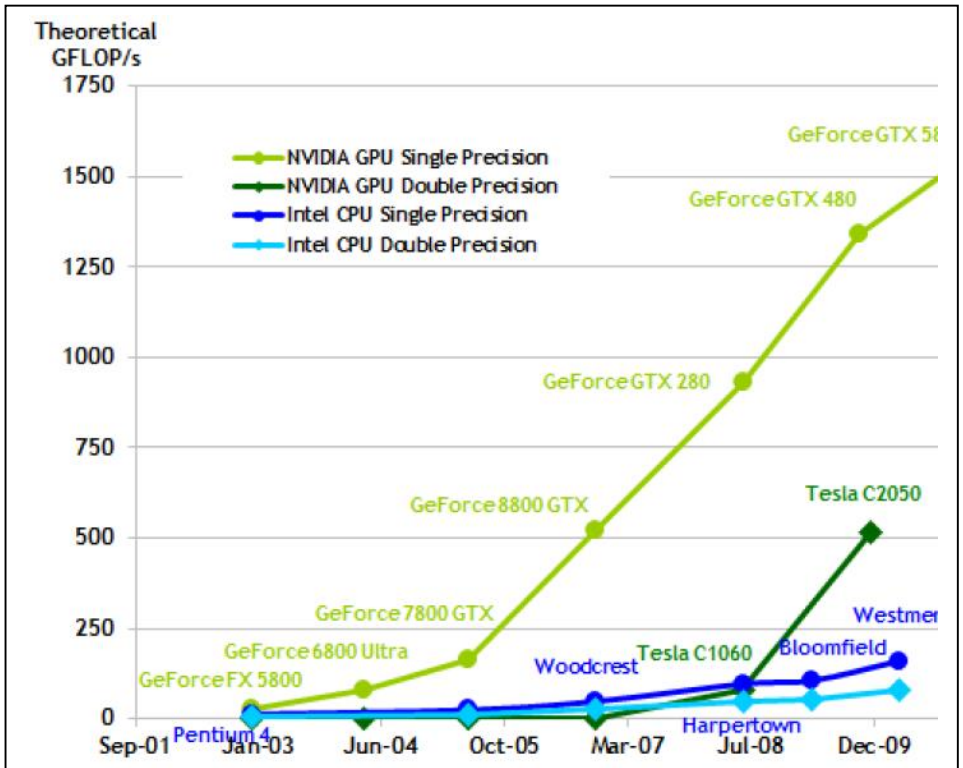


Figura 2.2: Operazioni Floating-Point al secondo nella GPU e nelle CPU

Più in dettaglio, le GPU sono orientate a risolvere problemi che possono essere espressi con il calcolo parallelo dei dati (lo stesso programma è eseguito su molti dati in parallelo) e con un'alta intensità aritmetica (ovvero il rapporto tra le operazioni aritmetiche e le operazioni in memoria). Poiché lo stesso programma è eseguito per ogni dato, non è necessario un sofisticato controllo di flusso, e le latenze di accesso in memoria sono mascherate dal calcolo anziché da grandi cache di dati.

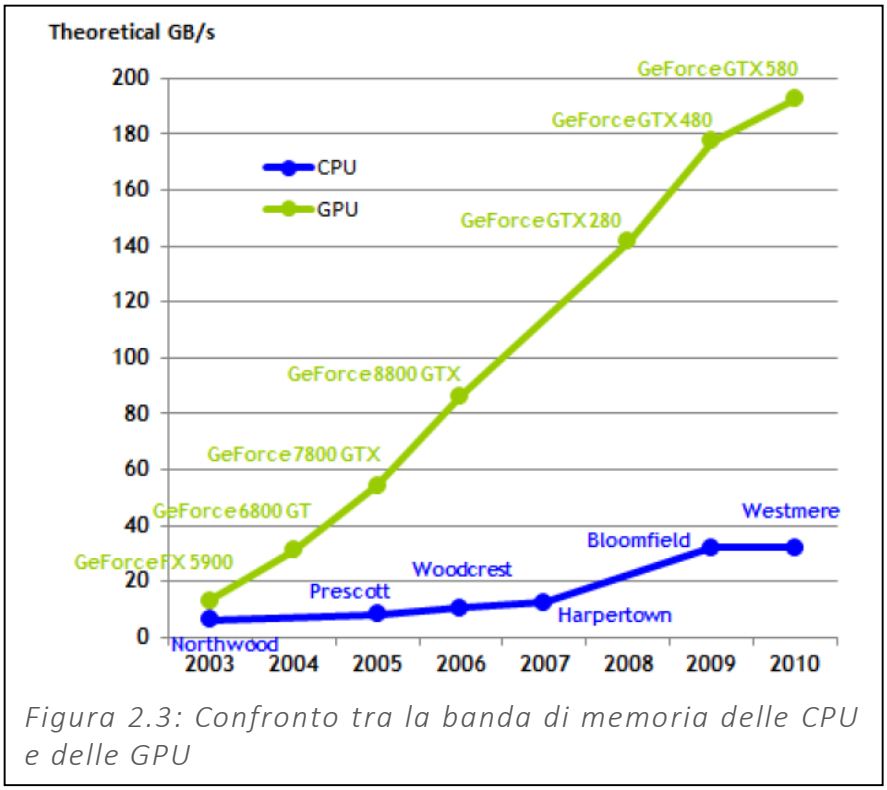


Figura 2.3: Confronto tra la banda di memoria delle CPU e delle GPU

Il modello *data parallel* mappa i singoli dati ai thread di calcolo. Molte applicazioni che processano grandi insiemi di dati possono usare il modello di programmazione data-parallelo per accelerare i calcoli. Nel rendering 3D, ad esempio, grandi insiemi di pixel e di vertici sono mappati in thread paralleli. In modo simile, le applicazioni per processare immagini e dati multimediali, come gli effetti di post-produzione o la codifica e la decodifica di video, etc., possono mappare parti di un'immagine o pixel in thread di calcolo paralleli.

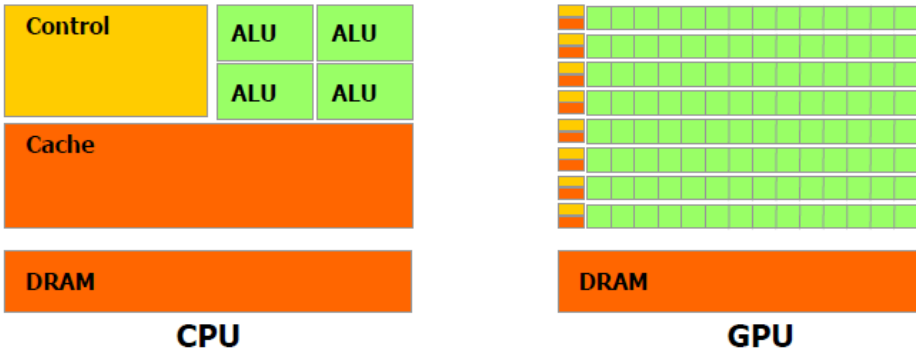


Figura 2.4: Confronto tra la struttura della CPU e della GPU

NVIDIA, uno dei maggiori produttori di GPU al mondo, ha rilasciato nel novembre del 2006 una piattaforma di programmazione, chiamata CUDA, per permettere ai programmatori di utilizzare facilmente le GPU per sviluppare applicazioni di calcolo.

CUDA

CUDA [7] ovvero *Compute Unified Device Architecture* è un'architettura di computazione parallela e general purpose. Essa è corredata da un'ambiente software che permette agli sviluppatori di usare C (o altri linguaggi molto diffusi) come linguaggio di programmazione di alto livello.

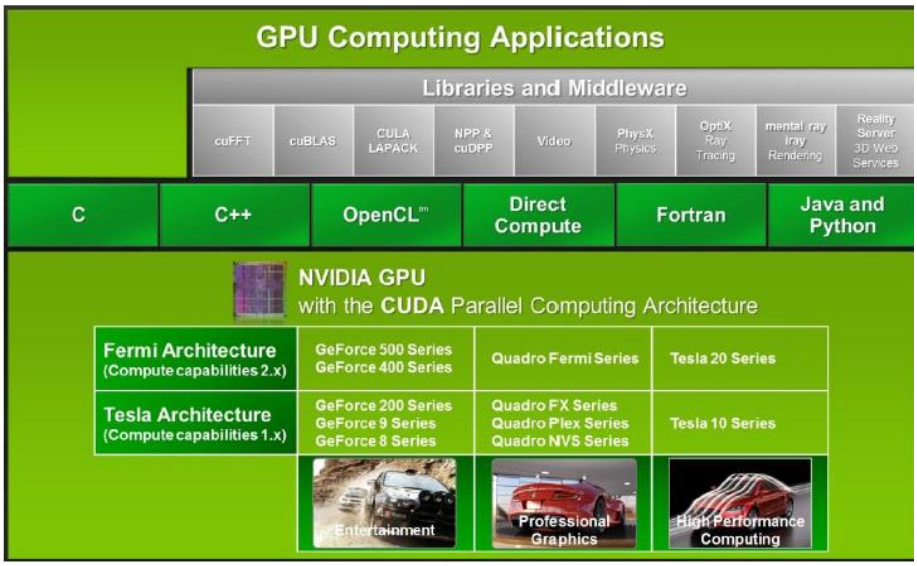


Figura 2.5: Panoramica dell'architettura di CUDA

L'avvento delle CPU multi-core e delle GPU many-core indica che ormai i computer attuali sono sistemi paralleli e continuano a scalare seguendo la legge di Moore. La sfida ora è quella di sviluppare applicativi software che scalino in modo trasparente il loro parallelismo, per bilanciare il crescente numero di core. Il modello di programmazione di CUDA è stato progettato per superare questa sfida mantenendo bassa la curva di apprendimento per i programmatori che hanno familiarità coi linguaggi di programmazione standard come il C.

I kernel e la gerarchia dei thread

CUDA C estende il linguaggio C permettendo al programmatore di definire funzioni C, chiamate kernel, che quando invocate vengono eseguite N volte in parallelo da N diversi thread CUDA, con N parametreo della funzione kernel.

Gli N thread di un kernel sono logicamente suddivisi in dei blocchi. Il numero di thread per blocco è fisicamente limitato dalla memoria del processore della GPU sul quale eseguono. Comunque, un kernel può essere eseguito da più blocchi di thread con la stessa forma, cosicché il numero totale di thread è pari al numero di thread in un blocco per il numero di blocchi.

I blocchi a loro volta sono organizzati in una griglia di blocchi di thread mono-, bi- o tridimensionale. Il numero di blocchi di thread in una griglia viene solitamente vincolato dalla taglia dei dati che devono essere processati o dal numero di processori nel sistema.

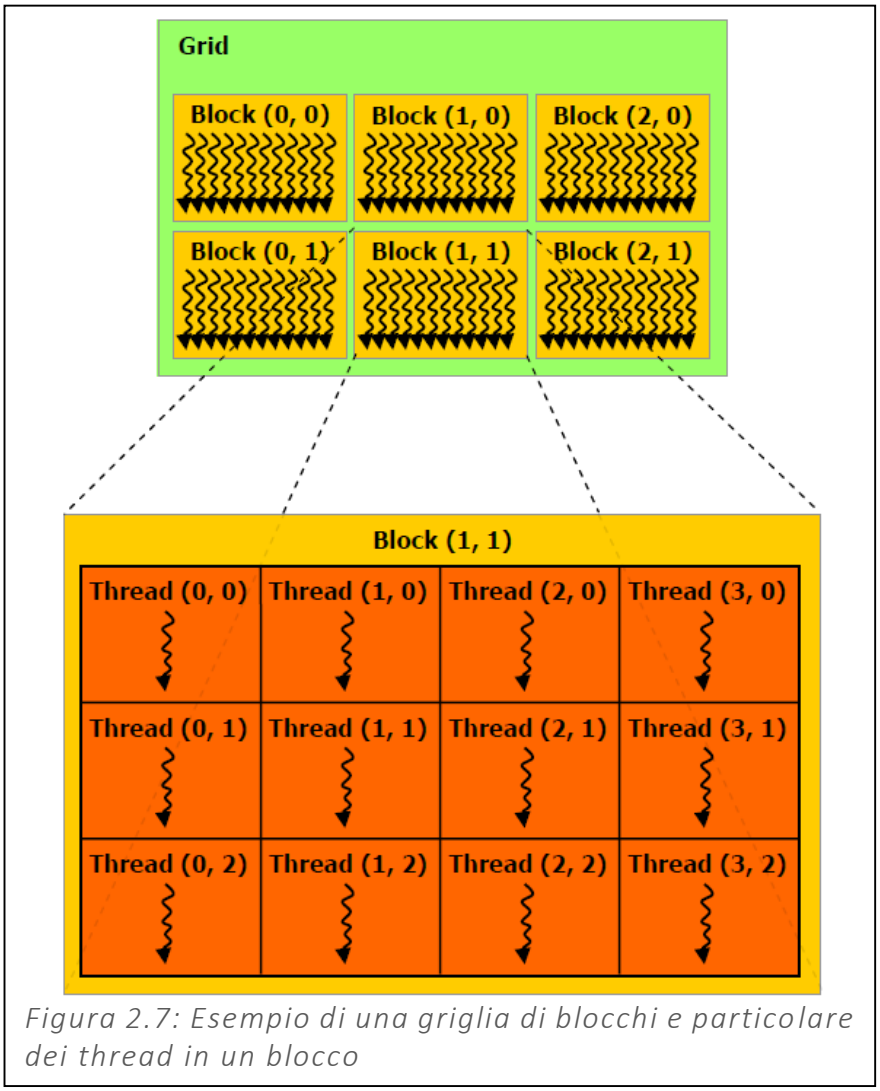


Figura 2.7: Esempio di una griglia di blocchi e particolare dei thread in un blocco

I blocchi di thread eseguono in modo indipendente tra loro. Questo requisito di indipendenza permette ai blocchi di thread di essere schedulati in qualsiasi ordine tra un qualsiasi numero di core, permettendo al programmatore di scrivere codice la cui esecuzione scala con il numero di core disponibili.

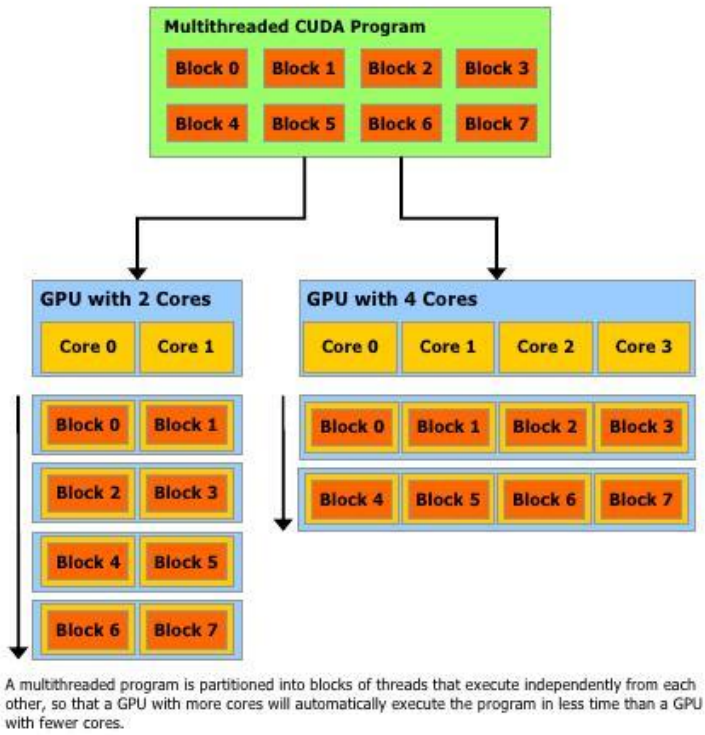


Figura 2.8: Esempio di flessibilità e scalabilità dell'esecuzione di un kernel su sistemi diversi

I thread all'interno di un blocco possono cooperare, condividendo dati attraverso alcune memorie condivise e sincronizzando la loro esecuzione per coordinare gli accessi in memoria.

La gerarchia di memoria.

I thread CUDA possono accedere ai dati di diversi spazi di memoria durante la loro esecuzione.

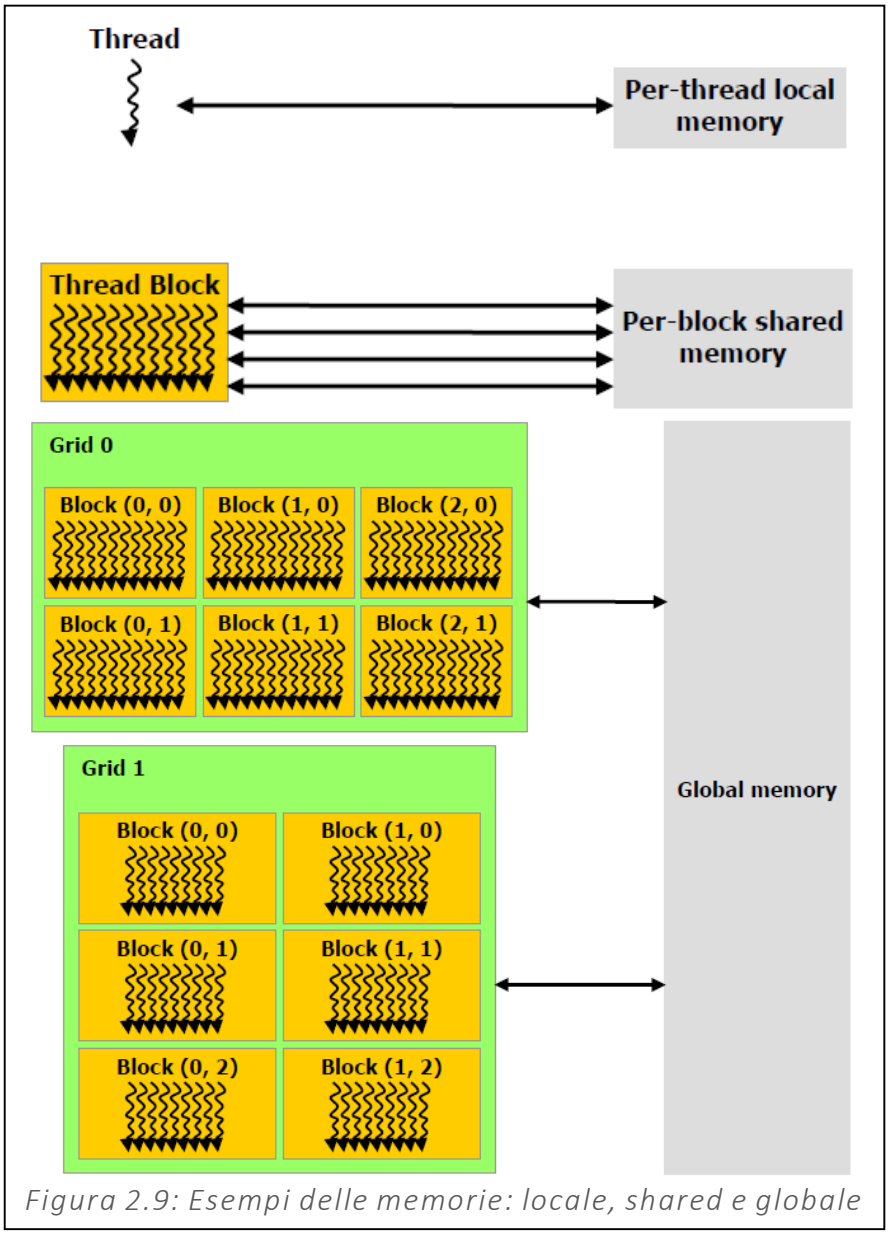


Figura 2.9: Esempi delle memorie: locale, shared e globale

Ogni thread ha la sua memoria locale privata. Ogni blocco di thread ha una memoria condivisa (shared) visibile a tutti i thread del suo blocco, la cui durata è pari alla vita del blocco. Tutti i thread hanno accesso alla stessa memoria globale.

Ci sono altri due spazi di memoria aggiuntivi di sola lettura accessibili da tutti i thread, lo spazio di memoria *costante* e lo spazio di memoria *texture*. Gli spazi di memoria, globale, costante e texture, sono ottimizzati per diversi usi. La memoria texture offre, inoltre, diverse modalità di indirizzamento, come ad esempio il filtraggio dei dati per determinati formati di dati. Gli spazi di memoria globale, costante e texture sono persistenti attraverso il lancio dei Kernel della stessa applicazione.

Programmazione eterogenea.

Il modello di programmazione CUDA assume che i thread CUDA vengano eseguiti su un dispositivo (*device*), fisicamente separato, che opera come un coprocessore per il programma

C che gira sul *host*. Ad esempio, questo accade quando i kernel vengono eseguiti sulla GPU ed il resto del programma C viene eseguito sulla CPU. Inoltre, il modello programmazione CUDA assume che sia l'host che il device mantengano i loro spazi di memoria separati nella DRAM, indicandoli rispettivamente come memoria host e memoria device.

3. Flockstream su GPU

Come evidenziato in precedenza, gli agenti utilizzati da FlockStream dispongono di *capacità limitate e non conoscono lo stato globale* del sistema. È inoltre assente un ente unico coordinatore. Per queste peculiarità risulta naturale pensare di avvantaggiarsi della piattaforma parallela CUDA per l'implementazione di questo algoritmo.

L'implementazione di FlockStream in CUDA ha affrontato diverse problematiche generali del calcolo parallelo oltre a problematiche specifiche per la piattaforma scelta. Questo ha portato anche ad alcune modifiche nel funzionamento dell'algoritmo rispetto a come presentato nel precedentemente.

In questa sezione, si discuterà in primo luogo del modello concretamente utilizzato, in seguito si esaminerà l'implementazione, mostrando un quadro generale, presentando le strutture dati utilizzate ed esaminando più a fondo l'implementazione di alcune fasi d'interesse.

Il modello utilizzato

Il modello utilizzato concretamente nell'implementazione dell'algoritmo, varia leggermente rispetto a quello del Multiple Species Flocking Clustering presentate precedentemente, in quanto:

- Sono state agiunte le regole per la propagazione delle etichette di clustering;
- Le regole di Reynolds sono ora anche pesate rispetto alla weight degli agenti, come definita dal FlockStream;
- Le regole di Reynlods sono pesate anche rispetto alle distanze D1 e D2, questo per rendere il comportamento più prevedibile rispetto alle modifiche di questi due parametri.

Definizioni

Parametri per il clustering:

- ε: distanza massima tra due tuple perché la coppia sia considerata simile, con $\varepsilon > 0$;
- λ: parametro della funzione di fading esponenziale $f(t) = e^{-\lambda t}$ con $\lambda > 0$;
- p_soglia: gli agenti con peso maggiore della p_soglia saranno considerati p-rappresentativi
- o_soglia: gli agenti con peso compreso tra la o_soglia e la p_soglia sono considerati o_rappresentativi;

Parametri per il flocking:

- $W_{al}, W_{co}, W_{se}, W_{di}$: massimi valori consentiti per le componenti della velocità di un boid, , rispettivamente, di allineamento, coesione, separazione, dissimilarità;
- D1: distanza nello spazio virtuale, orizzonte di visibilità dei boid nello spazio virtuale, $D1 > 0$;
- D2: distanza nello spazio virtuale, minima distanza di guardia che gli agenti tenteranno di mantenere con tutti gli altri, $0 < D2 < D1$;
- Densità: la densità dei boid nello spazio virtuale, serve a determinare l'area dello spazio in funzione del numero degli agenti presenti;
- maxVel: velocità massima degli agenti;
- minVel: velocità minima degli agenti.

Definizione di Agente (boid):

Ogni agente (boid) consta delle seguenti:

- id: identificativo univoco dell'agente;
- f: vettore mutidimensionale delle feature associate all'agente;
- p: vettore bidimensionale rappresentante la posizione nello spazio virtuale dei boid;
- v: vettore bidimensionale rappresentante la velocità nello spazio virtuale;
- w: peso, decrescente in funzione del tempo, determinato da una funzione di fading;
- type: tipo, definisce il tipo (base, o-representative, p-representative);

Per la propagazione delle etichette di clustering e l'individuazione degli oggetti che andranno a formare gli o-representative e i p-representativi, si introducono le seguenti variabili, per ogni agente:

- $weightVicini$: somma approssimata dei pesi dei vicini (sullo spazio delle feature) per ogni oggetto, è una misura della densità dei vicini all'interno del raggio ε (sullo spazio delle feature).

- r: il raggio (nello spazio delle feature) di un o-representative o p-representative. Per un agente semplice, il raggio è pari a 0. Il raggio sarà in ogni caso minore di ε;

- $colorMacro$: rappresenta l'etichetta di clustering, ovvero il più basso identificativo tra tutti gli oggetti presenti nel cluster;
- $colorMacroWeight$: è il valore di peso associato all'oggetto di identificativo pari a colorMacro;

- $color$: array contentente l'etichetta locale per l'aggregazione in o-repr o p-repr (di ogni oggetto), anche in questo caso sarà il minore tra gli id degli oggetti vicini;
- $colorDistance$: distanza approssimata sullo spazio delle feature dall'agente con id pari all'etichetta color;

Funzioni:

- $dist_p(\cdot,\cdot)$: funzione di distanza euclidea nel piano virtuale dei boid;
- $dist_f(\cdot,\cdot)$: funzione di distanza sullo spazio delle feature.

Regole per la propagazione delle etichette di clustering

Ad ogni iterazione della fase di flocking, per ogni agente (boid) b, si calcoleranno:

- Propagazione dell'etichetta di clustering (colorMacro):
  - $\text{colorMacro(b)} = \min_{x \epsilon \Omega_b} \text{colorMacro(x)},$

$$con\ \Omega_b = \{x \mid dist_p(p_b, p_x) \leq D1 \wedge \left(dist_f(f_b, f_x) - r_b - r_x\right) < \varepsilon \wedge weightVicini(x) > o\_soglia\}$$

  - $colorMacroWeight(b) = w_{colorMacro(b)}$

L'etichetta di colore sarà pari alla minore tra le etichette degli agenti prossimi nello spazio virtuale, tali che siano tra loro simili sul piano delle feature e che la densità (weightVicini) sia

maggiore della o_soglia. L'etichetta non si propaga quindi in presenza di elementi isolati, ma comunque simili, in modo identico al concetto di density-connected proprio del DBScan.

- Propagazione dell'etichetta per l'individuazione di gruppi di oggetti da aggregare in o- o p- rappresentativi (color):
  - $\text{color(b)} = \min_{x \epsilon \Omega_\text{b}} \text{color(x)}$,

$$con\ \Omega_\text{b} = \{x\ |dist_p(p_b, p_x) \leq D1$$
$$\wedge \left(dist_f(f_b, f_x) + \text{r}_\text{b} + \text{r}_\text{x} + \text{colorDistance(x)}\right) < \varepsilon\}$$

  - $colorDistance(b) = \left(dist_f(f_b, f_x) + \text{r}_\text{b} + \text{r}_\text{x} + \text{colorDistance(x)}\right)$,

$$\text{con x} = \text{color(b)}$$

  Attraverso le due regole sopra si garantisce che la distanza tra gli agenti con identica etichetta color non sia maggiore del raggio massimo, ovvero non sia maggiore di ε. Questo perché la condizione $\left(dist_f(f_b, f_x) + \text{r}_\text{b} + \text{r}_\text{x} + \text{colorDistance(x)}\right) < \varepsilon$ comporta la propagazione dell'etichetta soltanto se esiste un cammino (sullo spazio delle feature) dall'agente generatore dell'etichetta fino all'agente considerato minore del massimo raggio consentito per un o- o p- rappresentativo, ovvero ε. Considerare inoltre i raggi r_b ed r_x all'interno della condizione, assicura che la distanza considerata sia comprensiva dei raggi degli agenti considerati.
- Aggiornamento densità locale (weightVicini):

$$weightVicini_b = \max\left(weightVicini_x, \sum_{x \epsilon \Omega_\text{b}} w_x\right)$$
$$\text{con } \Omega_\text{b} = \{x\ |dist_p(p_b, p_x) \leq D1\}$$

  Questo valore è necessario per calcolare la densità locale di un agente nello spazio delle feature, e tenderà a convergere a $\sum_{y \epsilon \Omega_\text{b}} w_y$, con $\Omega_\text{b} = \{y\ |dist_f(f_b, f_y) \leq \varepsilon\}$ al proseguire delle diverse iterazioni, ovvero la sommatoria dei pesi di tutti i vicini nello spazio delle feature.

Regole per il flocking.

Di seguono si mostrano le regole utilizzate per il flocking in luogo di quelle proposte per l'MSFC, meglio adattate all'algoritmo in esame, per prendere in considerazione anche i pesi degli oggetti e le distanze D1 e D2.

Ad ogni iterazione della fase di flocking, per ogni agente (boid) b, si calcoleranno:

- **Allineamento:**

$$v_{al_b} = \frac{1}{|\Omega_b|} \sum_{x \epsilon \Omega_b} w_x \cdot v_x \,,$$

$$con\ \Omega_b = \{x \mid D2 < dist_p(p_b, p_x) \leq D1 \wedge dist_f(f_b, f_x) < \varepsilon\}$$

  La componente di allineamento dipende dal peso dei boid incontrati (w_x) come per tutte le rimanenti regole.

- **Separazione:**

$$v_{se_b} = \sum_{x \epsilon \Omega_b} w_x \cdot \frac{-(v_x + v_b)}{dist(p_b, p_x)} \cdot \left(1 - \frac{dist_p(p_b, p_x)}{D2}\right),$$

$$con\ \Omega_b = \{x \mid dist_p(p_b, p_x) \leq D2\}$$

  Si noti che la componente $\left(1 - \frac{dist_p(p_b, p_x)}{D2}\right)$ ha valore sempre compreso tra 0 ed 1, e pesa in funzione della distanza delle posizioni rapportata alla distanza di guardia D2. La separazione dipende, dal peso dei boid incontrati (w_x).

- **Coesione:**

$$v_{co_b} = \sum_{x \epsilon \Omega_b} w_x \cdot (p_x - p_b) \cdot \left(1 - \frac{dist_f(f_b, f_x)}{\varepsilon}\right) \cdot \left(\frac{dist_p(p_b, p_x)}{D1}\right),$$

$$con\ \Omega_b = \{x \mid D2 < dist_p(p_b, p_x) \leq D1 \wedge dist_f(f_b, f_x) < \varepsilon\}$$

  $\left(\frac{dist_p(p_b, p_x)}{D1}\right)$, di valore sempre compreso tra 0 ed 1, pesa in funzione della distanza delle posizioni rapportata alla distanza di guardia D1.

- **Dissimilarità** sulle Feature:

$$v_{di_b} = -\sum_{x \epsilon \Omega_b} w_x \cdot (p_x - p_b) \cdot \frac{dist_f(f_b, f_x)}{\varepsilon} \cdot \left(1 - \frac{dist_p(p_b, p_x)}{D1}\right),$$

$$con\ \Omega_b = \{x \mid dist_p(p_b, p_x) \leq D1 \wedge dist_f(f_b, f_x) > \varepsilon \wedge colorMacro_b \neq colorMacro_x\}$$

La componente $\left(1 - \frac{dist_p(p_b,p_x)}{\mathrm{D1}}\right)$ ha valore sempre compreso tra 0 ed 1., e pesa in funzione della distanza delle posizioni rapportata alla distanza di guardia D1. L'etichetta colorMacro, come mostrato, rappresenta l'etichetta di clustering dell'agente. La dissimilarità tra oggetti dello stesso cluster non viene presa in considerazione.

L'implementazione

Di seguito presentiamo un diagramma di flusso per mostrare il quadro generale dell'implementazione (Figura 3.1):

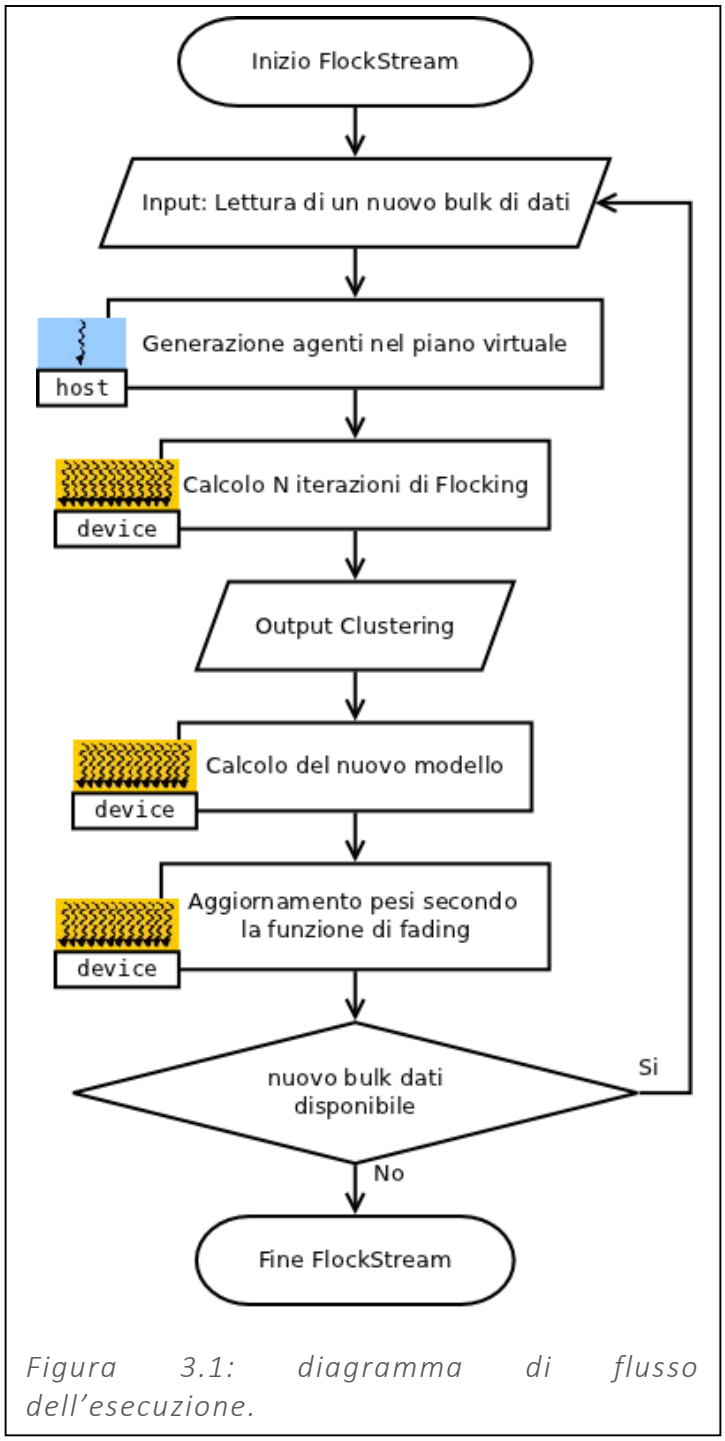


Figura 3.1: diagramma di flusso dell'esecuzione.

- **L'input**: come presentato nelle sezioni riguardanti DenStream e FlockStream, si assume un modello a finestre smorzate, nel quale il peso di ogni tupla del flusso dati decresce esponenzialmente col tempo t secondo una funzione di fading esponenziale $f(t) = e^{-\lambda t}$ con $\lambda > 0$. Il peso totale del flusso è la costante $W = \frac{v}{1-2^{-\lambda}}$, dove v è la costante di velocità del flusso, ovvero il numero di punti che arrivano in elaborazione nell'unità di tempo.
- **Generazione agenti nel piano virtuale**: per ogni tupla nell'input e per ogni o- o p-rappresentativo nel modello, si genera un agente (boid) nel piano virtuale.
- **Calcolo N iterazioni di Flocking**: questa fase, discussa più in dettaglio nella sezione successiva, rappresenta il nucleo esecutivo dell'algoritmo. Ad ogni finale di iterazione, è in grado di presentare un risultato di clustering parziale, migliore all'aumentare del numero di iterazioni, rendendo così l'algoritmo anytime: è possibile fermare l'esecuzione delle iterazioni in qualunque momento, anche senza completare tutte le N iterazioni prefissate
- **Output**: per ogni elemento della finestra dati esaminata, è possibile esibire l'etichetta di clustering.
- **Calcolo del nuovo modello**: gli oggetti verranno aggregati in o-rappresentativi o p-rappresentativi, formando o aggiornando il modello.
- Aggiornamento pesi secondo la funzione di fading: autoesplicativa.
- **Nuova finestra dati**: se è presente una nuova finestra dati da esaminare, riprendi l'esecuzione dalla lettura di un nuovo input.

Calcolo iterazioni di flocking

Il diagramma in Figura 3.2 approfondisce la fase del calcolo delle iterazioni di flocking, presentata in precedenza:

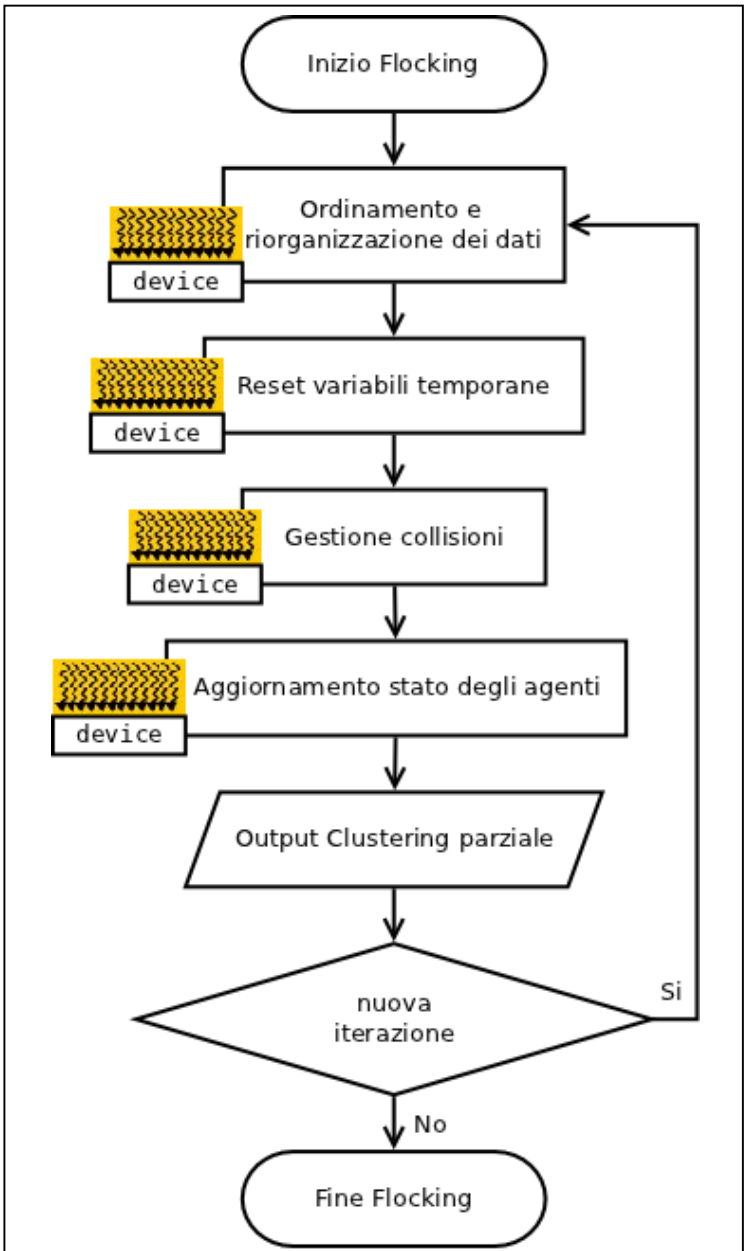


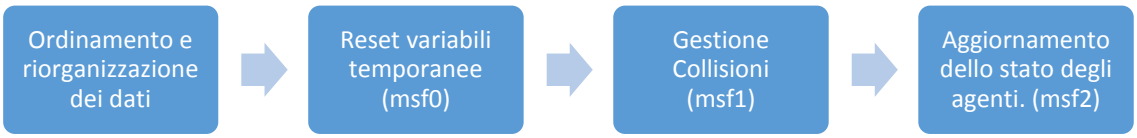
Figura 3.2: diagramma di flusso per il calcolo del flocking.

- **Ordinamento e riorganizzazione dati:** ordina gli agenti in base alla cella che essi occupano nello spazio virtuale dei boid. Riordina inoltre le strutture dati di conseguenza. Ciò è necessario per ottimizzare l'esecuzione dell'algoritmo.
- **Reset variabili temporanee:** prevede il reset delle variabili temporanee necessarie per il calcolo delle fasi successive
- **Gestione Collisioni:** questo è il cuore del calcolo, esegue l'individuazione delle collisioni tra gli agenti e ne modifica le variabili temporanee di conseguenza.

- **Aggiornamento delo stato degli agenti:** Modifica lo stato degli agenti in funzione dello stato precedente e dei valori contenuti nelle variabili temporanee
- **Output clustering parziale:** è possibile fermare l'esecuzione dell'algoritmo e visualizzare il risultato parziale del clustering. Questo risultato andrà raffinandosi con l'esecuzione di altre iterazioni.

Esecuzione di un'iterazione di flocking

Un'iterazione di flocking viene eseguita in quattro fasi, al termine della quale si avrà un risultato di clustering parziale, raffinato con il prosieguo delle iterazioni.



L'ordinamento: ordina gli agenti in base alla cella che essi occupano nello spazio virtuale dei boid. Riordina inoltre le strutture dati di conseguenza. Ciò è necessario per ottimizzare l'esecuzione dell'algoritmo.

L'inizializzazione: prevede il reset dei contenuti di tutte le variabili temporanee.

Gestione Collisioni: questo è il cuore del calcolo, esegue l'individuazione delle collisioni tra gli agenti e ne modifica le variabili temporanee di conseguenza.

Aggiornamento: Modifica lo stato degli agenti in funzione dello stato precedente e dei valori precedentemente calcolati.

Ordinamento e riorganizzazione

Per rendere più rapida l'esecuzione della fase di Gestione delle Collisioni, si è scelto di suddividere lo spazio virtuale bidimensionale degli agenti in una griglia, con celle di lato pari all'orizzonte di visibilità degli agenti ($D1$). Questo consente, durante la fase suddetta, di diminuire la divergenza inter-blocco tra i thread, come si mostrerà in seguito.

L'ordinamento prevede quattro fasi:

1. la generazione di un vettore di chiavi, calcolate con una funzione di hashing sulle posizioni degli agenti nello spazio dei boid;
2. la generazione di un vettore di valori, contenente per ogni agente la sua posizione nella struttura dati;
3. l'ordinamento, mediante implementazione parallela del radix sort, dei vettori generati;
4. il marshalling delle strutture dati: le strutture dati sono riorganizzate secondo l'ordinamento precedentemente calcolato.

Algoritmo per l'ordinamento in CUDA

Diversi algoritmi paralleli e concorrenti sono stati proposti per affrontare il problema dell'ordinamento. Tra questi, risultano interessanti gli approcci secondo modelli di reti di ordinamento [11][12], maturati in particolare per la progettazione di componenti elettroniche. Un altro buon risultato sull'ordinamento in CUDA è presente in [14]

In [8] è presentata un'implementazione efficiente per architettura CUDA dell'ordinamento. Questo è basato sul Parallel Prefix Sum (Scan)[13][15], utilizza il radix sort per l'ordinamento dei dati intra-blocco ed il merging bitonico dei segmenti ordinati, tutto calcolato su GPU.

Il radix sort è scelto nel caso trattato poiché il numero di bit della chiave dipende dal numero di celle della griglia, e questo, nei test svolti, è in generale molto inferiore ai 32 bit di in intero standard. L'efficienza del radix sort dipende linearmente dal numero di bit della chiave.

Prefix Sum. L'operazione di Prefix Sum (somma dei prefissi), prende in input un operatore binario associativo con identità I ed un array di n elementi,

$$[a_0, a_1, \dots, a_{n-1}]$$

E restituisce un array

$$[I, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})]$$

Per esempio, se consideriamo $\oplus$ addizione, allora l'operazione di somma dei prefissi sull'array:

$$[3, 1, 7, 0, 4, 1, 6, 3]$$

Restituirà:

$$[0, 3, 4, 11, 11, 15, 16, 22]$$

L'implementazione di base della Prefix Sum in CUDA è fornita in [8]: si costruisce un albero binario bilanciato sui dati in input e lo si scansiona prima dalle foglie alla radice e quindi dalla radice alle foglie.

In questo caso, l'albero non è una vera struttura, ma piuttosto un concetto per determinare quali elementi dell'array in input devono essere aggregati

Nella prima fase (Figura 3.4), detta di riduzione, poiché al termine dell'elaborazione otterremo la sommatoria di tutti gli elementi nell'ultima posizione, si effettua una aggregazione binaria dalle foglie alla radice, come mostrato in figura:

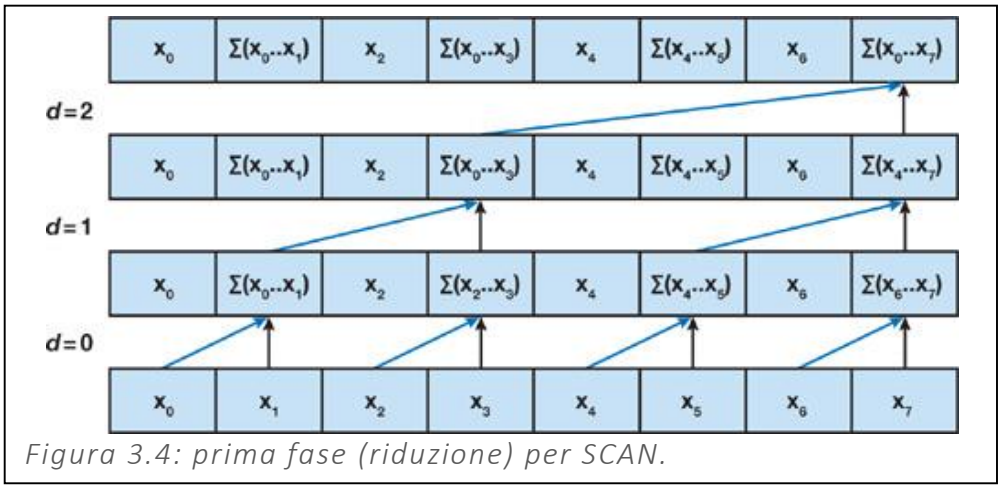


Figura 3.4: prima fase (riduzione) per SCAN.

Nella seconda fase (Figura 3.5) si procederà dall'alto verso il basso, utilizzanto le somme parziali ottenute durante la prima fase per calcolare la somma dei prefissi, come mostrato in figura.

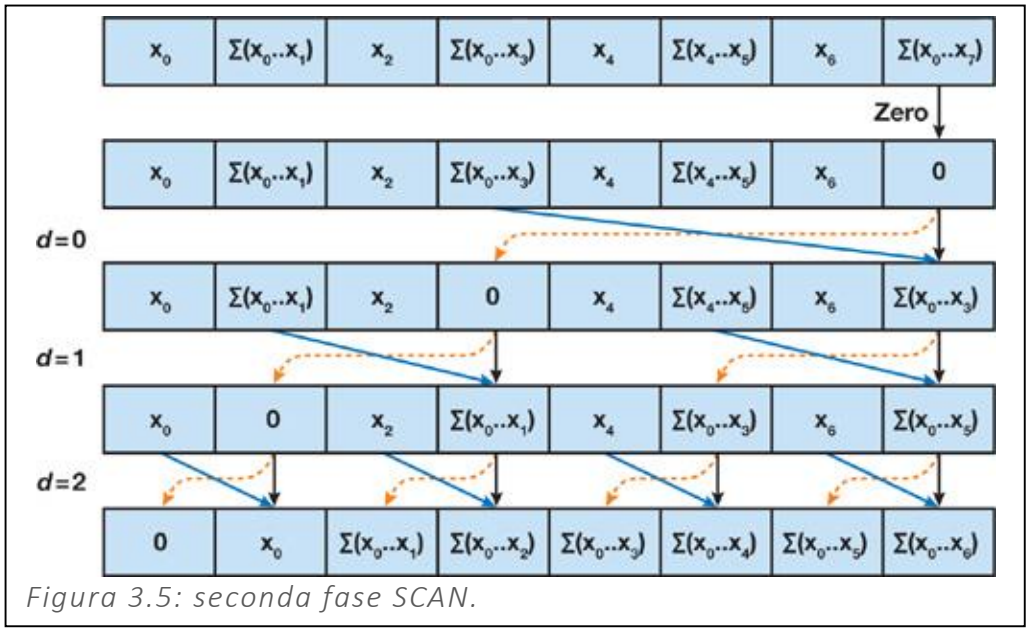


Figura 3.5: seconda fase SCAN.

Compattare un flusso dati. Il prefix scan è importante poichè consente, tra l'altro, di compattare un flusso di dati. Se consideriamo, ad esempio, un array binario dove 1 rappresenta un elemento che vogliamo mantenete e 0 un elemento che vogliamo scartare, ad esempio:

$$val = [a, b, c, d, e, f, g, h]$$

$$in = [0,1,1,0,0,1,0,1]$$

consideriamo $\oplus$ addizione, risulterà:

$$out = [0,0,1,2,2,2,3,3]$$

Basterà muovere ogni elemento con in[9]=1 nella posizione out[9] per compattare i dati, ottenendo:

$$[b, c, f, h]$$

Partizionamento. Estendendo, lo stesso si può applicare per partizionare i dati, oltre che per compattarli, come mostrato nella figura 3.6:

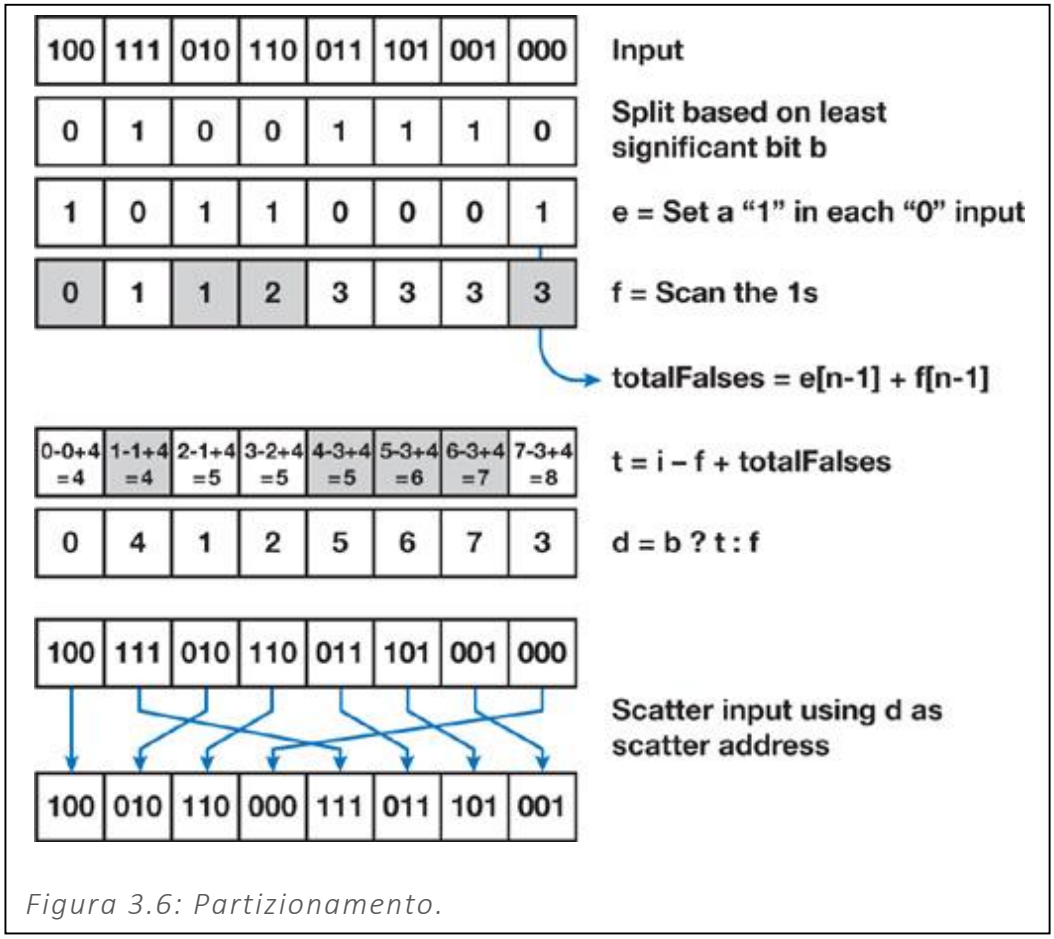


Figura 3.6: Partizionamento.

1. In un buffer temporaneo nella memoria condivisa, si setta 1 per tutti gli elementi con bit di chiave 0 e a 0 tutti gli elementi con bit di chiave 1.
2. Quindi si esegue una somma dei prefissi sul buffer. Tutti gli elementi con corrispondente bit di chiave pari a 0 avranno nel buffer la nuova posizione, come visto nel caso del compattamento di un flusso dati.
3. Si calcola totalFalse come il numero di elementi che ha bit di chiave pari a 0.

4. Si calcola la nuova posizione degli elementi con chiave 1, come mostrato in figura.
5. Si esegue lo spostamento dei valori.

Radix Sort. Risulta quindi semplice l'implementazione del radix sort, partizionando sui bit di chiave dal meno significativo al più significativo. L'ordinamento dei dati all'interno di un blocco si calcola in k passi, con k pari al numero di bit della chiave.

Effettuato l'ordinamento di una serie di blocchi, se ne esegue il merging come mostrato in figura 3.7:

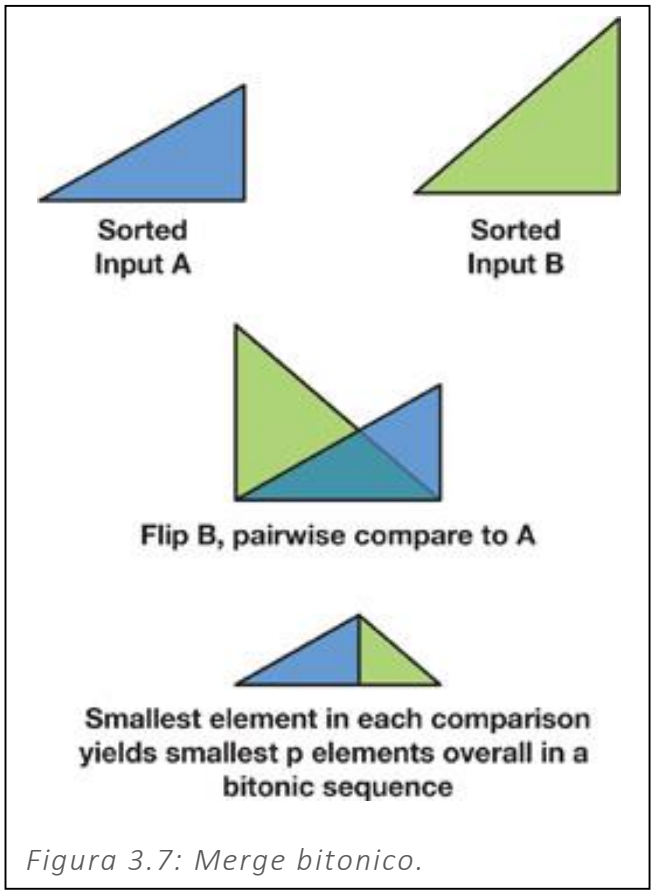


Figura 3.7: Merge bitonico.

Dati due input ordinati A e B, si compara bitonicamente dal più piccolo degli elementi di A e dal più grande degli elementi di B. Il minimo di ogni comparazione sarà inserito in A, il massimo in B. Al termine, avremo in A tutti gli elementi più piccoli della mediana degli elementi di A e B, ed in B tutti gli altri. L'operazione di merge costa una sola scansione dei dati.

Gestione Collisioni

In questa fase, si esegue l'individuazione delle collisioni tra gli agenti e si modificano le variabili temporanee `tempVars` di conseguenza, per consentire nella fase successiva l'aggiornamento dello stato.

Per eseguire l'algoritmo, è necessario che ogni agente confronti le sue feature con quelle di ogni altro agente incontrato, come mostrato in precedenza. Si è quindi dovuto scegliere tra due

diversi approcci: precalcolare la matrice delle distanze delle feature tra tutti gli agenti e tenerla in memoria oppure tenere in memoria le feature per ogni agente, e calcolare la distanza al volo ogni qual volta risultasse necessario.

Precalcolo Matrice delle Distanze: Il primo approccio, utilizzato in [10], è temporalmente il più efficiente: dopo un primo calcolo della matrice delle distanze, consta, in questa fase, di un minor numero di letture in memoria globale. Purtroppo però la dimensione della matrice risulta essere spazialmente quadratica rispetto al numero di agenti da elaborare. La GPU utilizzata per lo sviluppo, una GeForce 9600m, possiede una memoria di 512mb, e con questo approccio la memoria tende ad esaurirsi velocemente all'aumentare del numero di punti. Inoltre, il precedentemente descritto passo di riorganizzazione dei dati ha costi temporalmente maggiori, poiché prevede un numero di operazioni di lettura/scrittura in memoria globale maggiore, essendo necessario riorganizzare tutta la struttura dati.

Feature in memoria globale: L'approccio seguito è quindi quello del mantenimento in memoria globale delle feature relative agli agenti in esecuzione, spazialmente più efficiente, e che consente anche a GPU di segmento medio-basso l'elaborazione dell'algoritmo con una dimensione dell'input elevata. La tabella tenuta in memoria cresce linearmente col numero degli agenti e linearmente col numero delle feature, generalmente minore di quello degli agenti.

Con la suddivisione in celle dello spazio virtuale degli agenti ed il passo di riorganizzazione dei dati si è comunque ridotto il gap di efficienza temporale tra i due approcci.

In [16] si propone, per l'esecuzione del modello di flocking, un'implementazione basata su due kernel:

- Il primo kernel crea un thread per ogni coppia di agenti (n^2 thread in totale, con n numero di agenti) e compara le loro posizioni nello spazio virtuale bidimensionale per determinare se la distanza tra essi è all'interno della soglia $D1$. Se la distanza è sufficientemente piccola, si esegue il calcolo delle componenti parziali secondo il modello di Reynolds, che vengono scritte in memoria globale, producendo un output quadratico nel numero degli agenti, contenente il contributo che ogni agente ha su tutti gli altri.
- Il secondo kernel, di n thread, si occuperà di aggregare l'output del primo e di aggiornare lo stato di ogni agente.

Si noti che il primo kernel produce quindi un output di dimensioni quadratiche rispetto al numero degli agenti. Se si utilizzasse questo approccio, ritorneremmo con l'avere una complessità quadratica nello spazio di memoria (figura 3.8).

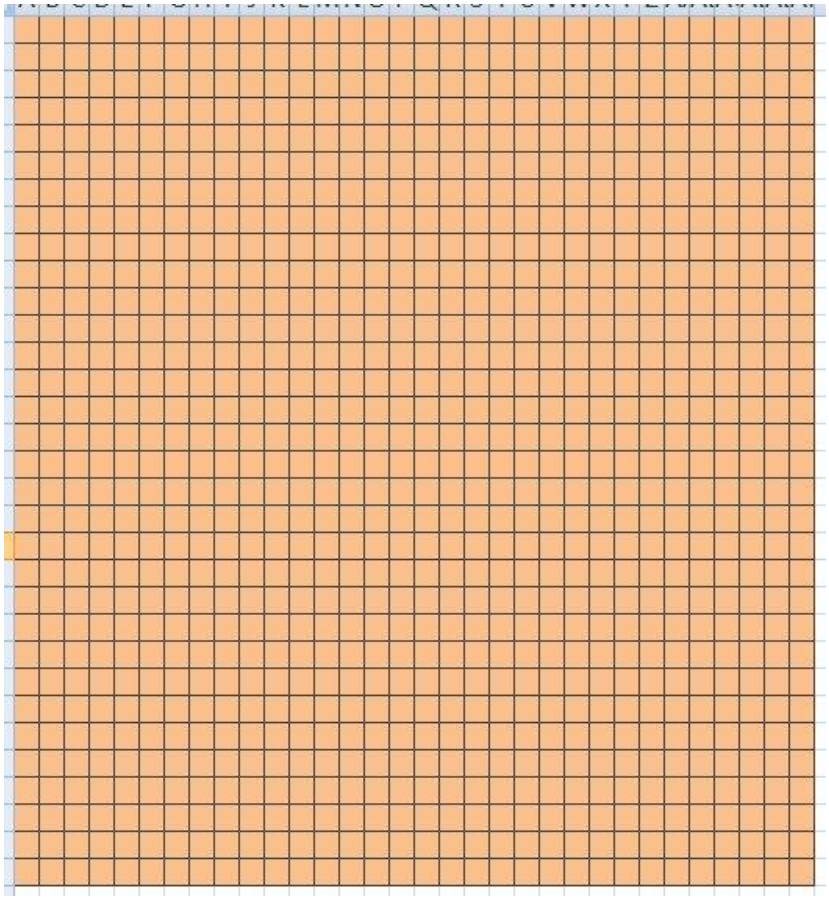


Figura 3.8: Una griglia 32x32. In Arancione i thread: ogni thread produce un valore.

L'approccio utilizzato prende spunto dal precedente, ma distribuisce l'esecuzione degli n^2 confronti in fette (slice) di dimensione predeterminata m (Figura 3.9).

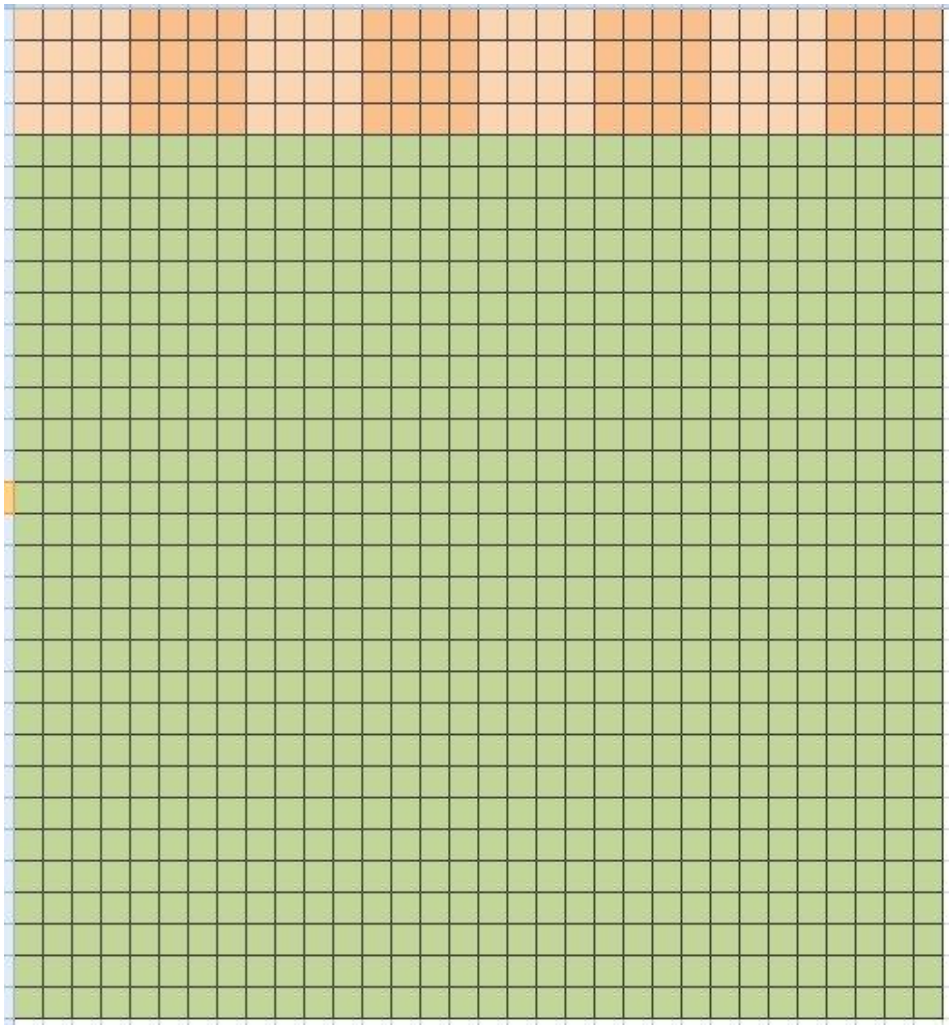


Figura 3.9: Una griglia 32x32, con slice di dimensione 4.

Si lancia un kernel di n thread, uno per agente, quindi, per ogni fetta di righe. Ogni thread calcola il contributo che gli m agenti appartenenti alla slice danno loro, effettuandone una aggregazione parziale. Si prosegue di seguito con le successive slice, fino alla compiuta esecuzione di tutti i confronti. In questo modo, l'occupazione della memoria globale risulterà lineare nel numero degli agenti.

Questo approccio, diminuendo il numero di thread lanciati contemporaneamente da n^2 a n, sembrerebbe diminuire la parallelità dell'implementazione, ma all'aumentare del numero di agenti, i thread saranno in grado di saturare i core della GPU.

Aggiornamento

Questa fase esegue l'aggiornamento dello stato degli agenti in funzione dello stato precedente e dei valori aggregati parziali, calcolati nella fase di Gestione delle Collisioni. Il kernel sarà eseguito su n thread, uno per ogni agente.

4. Risultati sperimentali

In questo sezione analizzeremo l'efficacia dellapproccio presentato su dei dataset generati artificialiamente, in modo randomico, DS1, DS2 e DS3, mostrati nelle figure 4.1, 4.2 e 4.3, considerando un rumore dei dati tra l'1% ed il 10%. È possibile generare, per ognuna delle forme proposte, un numero arbitrario di dati.

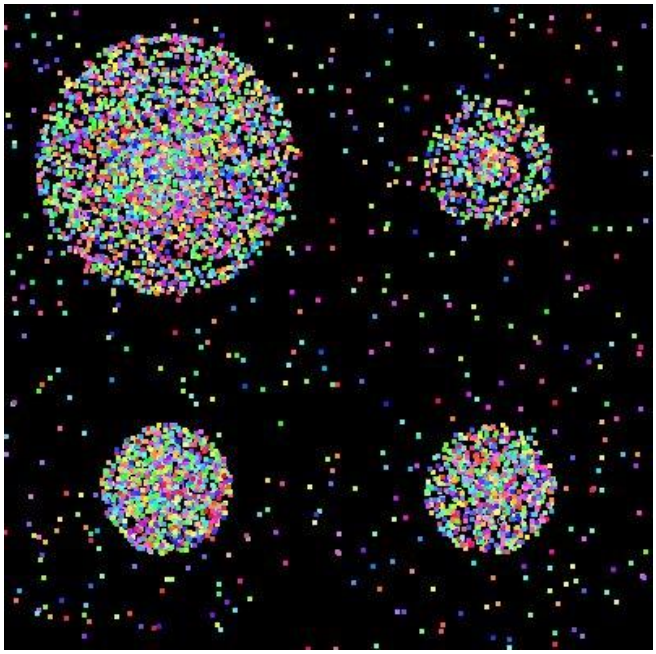


Figura 4.1: Dataset DS1, con 5% rumore.

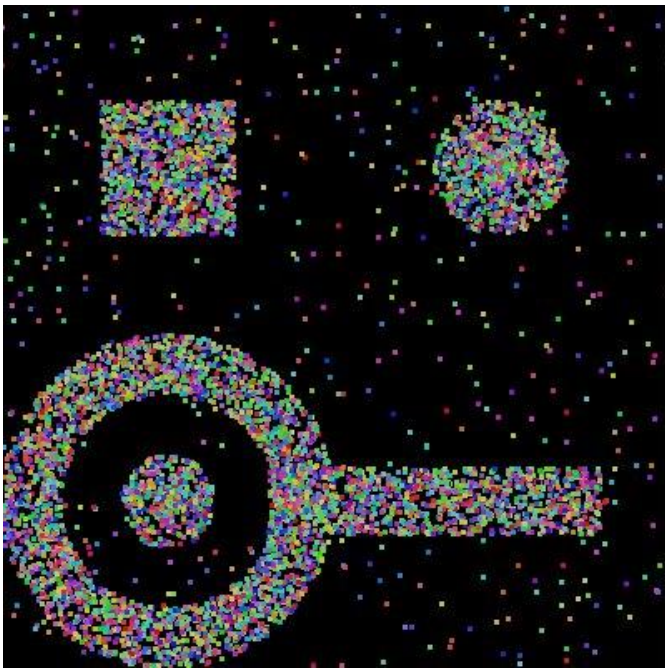


Figura 4.2: Dataset DS2, con 5% rumore.

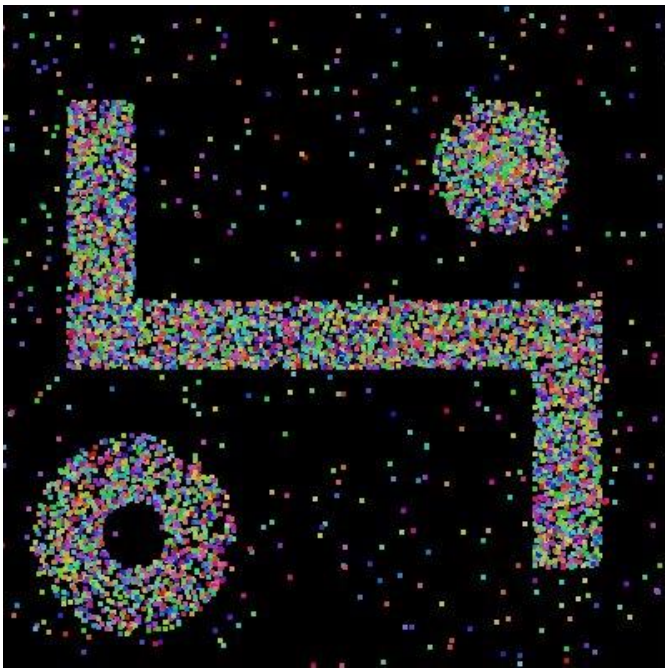


Figura 4.3: Dataset DS3, con 5% rumore.

Il set di dati evolvente EDS è stato generato dai tre precedenti, utilizzando la stessa strategia proposta in [1]: ognuno dei dataset è stato scelto a caso dieci volte, genarando ogni volta 10.000 punti, per un totale di 100.000 punti per il dataset evolvente.

Criteri di valutazione

Gli indici di valutazione scelti sono la *purezza media* e l'*informazione mutua normalizzata*. Il primo misura la purezza dei cluster ottenuti rispetto ai cluster attesi, assegnando ad ogni cluster atteso l'etichetta dominante tra quelle ottenute. Più formalmente:

$$purezza = \frac{\sum_{i=1}^{K} \frac{\left|C_i^d\right|}{|C_i|}}{K} * 100\%$$

Dove K indica il numero di cluster attesi, $\left|C_i^d\right|$ il numero di punti con etichetta di classe dominante rispetto al cluster atteso i, $|C_i|$ il numero dei punti nel cluster atteso i. L'indice di purezza produce un

L'*informazione mutua normalizzata* (NMI, normalized mutual information) misura quanto simili siano due clustering. Dato il cluster atteso $A = [A_1, \dots, A_k]$ ed un clustering $B = [B_1, \dots, B_h]$, sia C la matrice di confusione della quale ogni elemento C_{ij} è il numero di punti del cluster i di A che sono anche nel cluster j di B, l'$NMI(A, B)$ è definito come:

$$NMI(A,B) = \frac{-2 \sum_{i=1}^{c_A} \sum_{j=1}^{c_B} C_{ij} \log\left(\frac{C_{ij} N}{C_{i.} C_{.j}}\right)}{\sum_{i=1}^{c_A} C_{i.} \log\left(\frac{C_{i.}}{N}\right) + \sum_{j=1}^{c_B} C_{.j} \log\left(\frac{C_{.j}}{N}\right)}$$

Dove c_A (c_B) è il numero di gruppi della partizione A (B), $C_{i.}$ ($C_{.j}$) è la somma degli elementi di C sulla riga i (colonna j), e N il numero totale dei punti. Se $A = B$, $NMI(A, B) = 1$. Se A è completamente diverso da B, $NMI(A, B) = 0$.

I test sono stati eseguiti su una gpu NVIDIA Tesla con processore T10.

Primo gruppo di test

Obiettivo di questa prima serie di test è la valutazione della velocità di esecuzione dell'algoritmo al variare del numero di punti da analizzare. In particolare, si valuterà la velocità media di calcolo di una singola iterazione, calcolata su 10 finestre di dati con velocità del flusso (numero di punti per unità di tempo) crescente per ogni gruppo di test. Il dataset di riferimento DS3 ha la forma in figura 4.4:

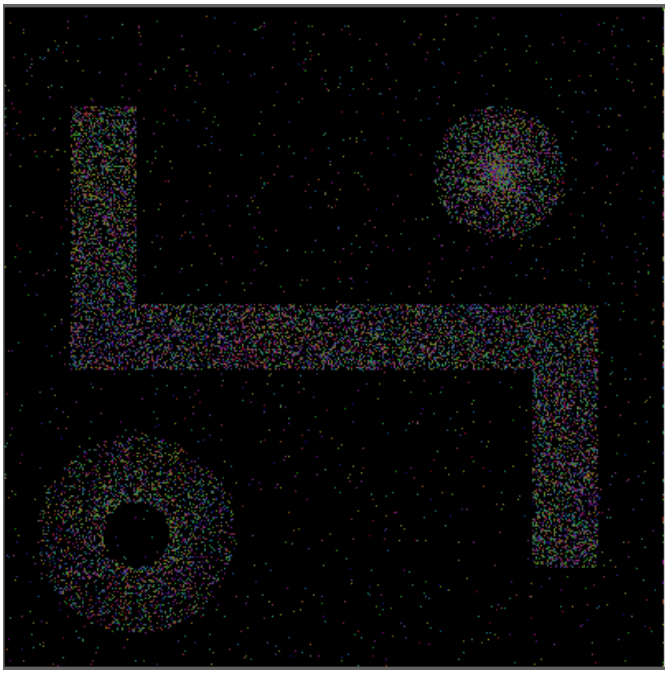


Figura 4.4: Dataset DS3, con 10% rumore.

Questo è un dataset "sporco", con un rumore generato pari al 10% del totale dei punti.

Il piano virtuale dei boid ha dimensione variabile rispetto al numero dei punti, ma con densità fissata e costante. Non vi sono altre variazioni nei parametri dell'algoritmo durante l'esecuzione di questa prima serie di test.

Di seguito alcuni risultati:

1024 punti per unità di tempo, 1024 punti per finestra					
ut	iterazioni per purezza al 90%	iterazioni per purezza al 95%	iterazioni per purezza al 98%	tempo per iterazione (ms)	numerosità modello
1	1100	1100	1100	4,623	0
2	700	900	900	6,07	133
3	400	500	500	6,515	237
4	600	600	700	6,674	292
5	400	500	600	6,836	316
6	500	600	700	6,79	310
7	500	600	600	6,747	297
8	700	700	700	6,692	292
9	600	600	700	6,653	286
10	700	700	900	6,68	292

Tabella 4.1:1024 punti per unità di tempo, 1024 punti per finestra.

La tabella 4.1 mostra in prima colonna le unità di tempo (nel caso in esame coincidenti con il numero delle finestre dati esaminate), nella seconda, terza e quarta colonna il numero di iterazioni necessarie per avere una purezza del 90%, 95% e 98%, la quinta colonna il tempo medio impiegato per eseguire una singola iterazione. La sesta colonna mostra il numero di punti del modello utilizzato nel calcolo del clustering per l'attuale finestra di dati.

Per ogni finestra si è scelto di eseguire fino a 2000 iterazioni dell'algoritmo.

È possibile notare come durante l'analisi della prima finestra, in mancanza di un modello precalcolato, siano necessarie più iterazioni per raggiungere un buon valore di purezza.

Come atteso, inoltre, la numerosità del modello tende a stabilizzarsi con l'aumentare delle finestre analizzate, in questo caso attorno ad un valore di circa 300 agenti.

Di seguito si riportano le tabelle ottenute considerando 2048, 4096, 8192 punti per finestra:

2048 punti per unità di tempo, 2048 punti per finestra					
ut	iterazioni per purezza al 90%	iterazioni per purezza al 95%	iterazioni per purezza al 98%	tempo per iterazione (ms)	numerosità modello
1	800	900	900	9,118	0
2	800	900	1300	11,725	182
3	600	700	900	12,448	321
4	800	800	1100	12,795	414
5	700	800	800	12,875	435
6	1600	1800	no	12,74	416
7	800	900	900	12,748	404
8	900	1100	1100	12,699	389
9	800	800	1000	12,67	383
10	1400	2000	2000	12,481	368

Tabella 4.2: 2048 punti per unità di tempo, 2048 punti per finestra.

4096 punti per unità di tempo, 4096 punti per finestra					
ut	iterazioni per purezza al 90%	iterazioni per purezza al 95%	iterazioni per purezza al 98%	tempo per iterazione (ms)	numerosità modello
1	1300	1400	1700	18,252	0
2	no	no	no	22,644	247
3	1000	1100	1200	23,929	427
4	1000	1200	1500	24,742	551
5	1100	1200	1500	24,38	564
6	1100	1300	1400	24,464	560
7	900	1000	1200	24,47	536
8	1500	no	no	24,318	520
9	1500	1500	2000	24,117	535
10	1000	1100	1200	24,216	517

Tabella 4.3: 4096 punti per unità di tempo, 4096 punti per finestra.

8192 punti per unità di tempo, 8192 punti per finestra					
ut	iterazioni per purezza al 90%	iterazioni per purezza al 95%	iterazioni per purezza al 98%	tempo per iterazione (ms)	numerosità modello
1	1500	1900	no	39,638	0
2	1700	1800	1800	49,295	403
3	no	no	no	51,735	716
4	1000	1200	1300	53,217	861
5	900	1000	1200	54,125	943
6	1100	1200	1500	52,998	887
7	1600	1700	no	53,301	898
8	1500	1500	1900	52,458	901
9	1300	1400	1700	52,734	922
10	1100	1100	1300	53,312	917

Tabella 4.4: 8192 punti per unità di tempo, 8192 punti per finestra.

Si noti come in alcuni casi (ad esempio all'unità di tempo 3 della Tabella 4.4) l'algoritmo non sia riuscito ad ottenere valori di purezza superiori al 90%. Al crescere del numero degli agenti all'interno dello spazio virtuale, è necessario un numero maggiore d'iterazioni per ottenere un risultato migliore, a parità di parametri.

Volendo, è possibile aumentare il parametro densità, in modo tale da diminuire la dimensione dello spazio virtuale e aumentare il numero di collisioni per iterazione del flocking. Questo comporta una maggiore velocità di convergenza in termini di numero d'iterazioni, ma minore decentralizzazione dell'algoritmo. Aumentando eccessivamente la densità, inoltre, lo spazio di manovra dei boid diminuisce, e nel peggiore dei casi si osserva un "effetto gabbia": gruppi di boid simili saranno bloccati dai troppi boid dissimili presenti nel loro intorno.

Come osservato per finestre di 1024 punti, la numerosità del modello si stabilizza attorno ad un certo valore per ogni dimensione di finestra. Tale valore cresce però all'aumentare della dimensione della finestra dati, effetto causato dalla presenza di rumore: i punti di rumore hanno densità molto bassa, e sono necessari molti più o-rappresentativi per tenerne traccia rispetto a quelli necessari per i punti ad alta densità.

Un'altra valutazione si ottiene mettendo in relazione il numero dei punti da analizzare e la velocità di esecuzione media delle iterazioni. Otteniamo il grafico in figura 4.5.

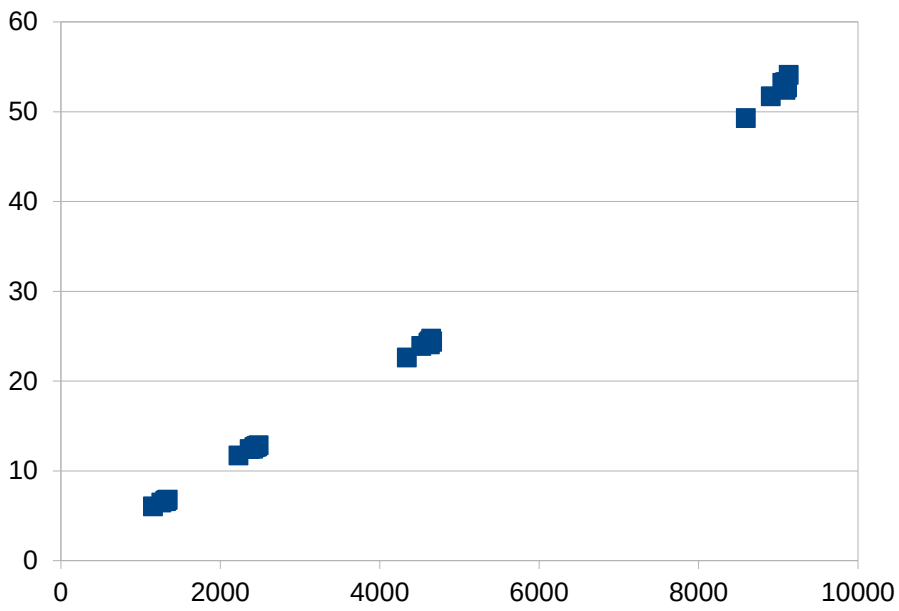


Figura 4.5: Velocità di esecuzione (ms) di una iterazione al crescere del numero di punti da analizzare.

Sulle ascisse abbiamo il numero di punti, sulle ordinate il tempo medio di calcolo di un'iterazione, in millisecondi. Il grafico evidenzia una crescita dei tempi lineare rispetto al numero dei punti che l'algoritmo analizza. L'algoritmo sembra quindi scalare efficientemente.

Secondo gruppo di test

Questo gruppo di test vuole mostrare il comportamento dell'algoritmo su un dataset evolvente, EDS, costruito come descritto in precedenza, al variare della dimensione della finestra (1000 o 2000 punti), del numero di iterazioni di flocking (1000 o 2000 iterazioni) e del rumore del dataset (1%, 5% e 10%). In questo caso utilizziamo anche l'indice NMI come metrica di valutazione.

EDS con 1% rumore

Parametri: $o\_soglia = 1{,}5, p\_soglia = 3, \varepsilon = 4, \lambda = 0{,}85$.

finestra iterazioni	1000 1000	punti per finestra
ut	purezza	nmi
10	1	0,983
20	1	0,976
30	1	0,974
40	0,999	0,977
50	1	0,973
60	0,998	0,968
70	1	0,985
80	1	0,977
90	1	0,963
100	1	0,973

Tabella 4.5

finestra iterazioni	1000 2000	punti per finestra
ut	purezza	nmi
10	1	0,983
20	1	0,976
30	0,999	0,97
40	1	0,98
50	1	0,973
60	1	0,97
70	1	0,985
80	0,999	0,975
90	1	0,963
100	0,999	0,967

Tabella 4.6

finestra 2000 punti per iterazioni 1000 finestra		
ut	purezza	nmi
10	0,996	0,957
20	0,998	0,966
30	0,996	0,939
40	0,996	0,97
50	0,996	0,96
60	0,977	0,881
70	0,998	0,972
80	0,999	0,972
90	0,998	0,965
100	0,997	0,955

Tabella 4.8

finestra 2000 punti per iterazioni 2000 finestra		
ut	purezza	nmi
10	1	0,975
20	1	0,979
30	1	0,963
40	1	0,981
50	1	0,975
60	1	0,963
70	1	0,975
80	1	0,976
90	1	0,968
100	1	0,972

Tabella 4.9

L'algoritmo esegue su EDS con rumore all'1% con buoni risultati in tutti i casi testati. In Figura 4.6 un grafico comparativo dell'indice NMI:

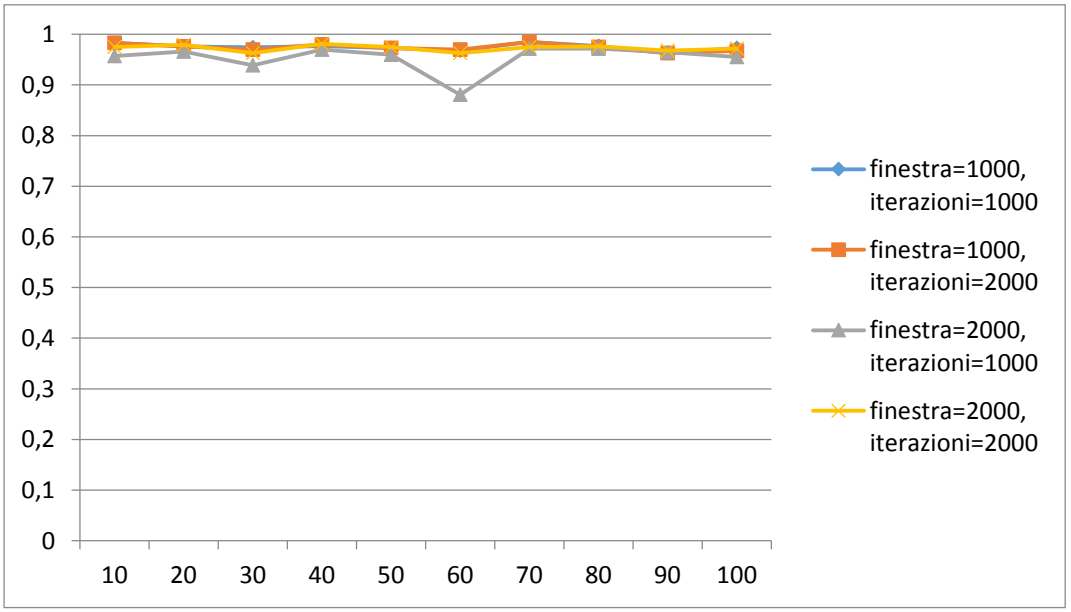


Figura 4.6: comparative indice NMI su EDS con 1% rumore. In ascissa le unità di tempo, in ordinata l'indice NMI

Il picco basso al tempo 60, considerando una finestra di 2000 punti e 1000 iterazioni (triangoli verdi in figura 4.6, valori nella Tabella 4.8) è dovuto alla mancata unione di due cluster predetti nell'unico cluster atteso. Aumentando le iterazioni (croci viola in figura 4.6, valori nella Tabella 4.9), a parità di finestra, questo errore viene corretto.

EDS con 5% rumore

Parametri: $o\_soglia = 1{,}5, p\_soglia = 3, \varepsilon = 4, \lambda = 0{,}85$.

finestra iterazioni	1000 1000	punti per finestra
ut	purezza	nmi
10	1	0,9
20	1	0,903
30	1	0,872
40	0,999	0,899
50	1	0,887
60	0,997	0,872
70	0,972	0,844
80	1	0,905
90	1	0,892
100	0,999	0,893

Tabella 4.10

finestra iterazioni	1000 2000	punti per finestra
ut	purezza	nmi
10	1	0,901
20	0,978	0,735
30	1	0,874
40	0,999	0,898
50	1	0,888
60	1	0,879
70	1	0,916
80	1	0,904
90	1	0,891
100	1	0,896

Tabella 4.11

finestra 2000 punti per finestra		
iterazioni 1000		
ut	purezza	nmi
10	0,991	0,874
20	0,989	0,866
30	0,989	0,836
40	0,997	0,899
50	0,996	0,883
60	0,997	0,858
70	0,952	0,792
80	0,995	0,89
90	0,989	0,853
100	0,998	0,878

Tabella 4.12

finestra 2000 punti per finestra		
iterazioni 2000		
ut	purezza	nmi
10	1	0,895
20	1	0,893
30	1	0,879
40	1	0,902
50	1	0,895
60	1	0,866
70	1	0,895
80	0,999	0,891
90	1	0,883
100	1	0,889

Tabella 4.13

Aumentando la rumorosità del dataset, si ottengono comunque buoni risultati di purezza. Il terzo caso (finestra 2000 punti, 1000 iterazioni per finestra, in tabella 4.12) mostra risultati di purezza minori rispetto agli altri. L'aumento del rumore in congiunzione dell'aumento della numerosità della finestra richiede un numero di iterazioni più elevato. Infatti, nel quarto caso (Tabella 4.13), che differisce per un numero di iterazioni doppio rispetto al precedente, otteniamo nuovamente risultati di purezza ottimali.

L'indice NMI, per costruzione, tende a diminuire aumentando il rumore del dataset. A questo proposito, i risultati sono allineati a quelli presentati in [5].

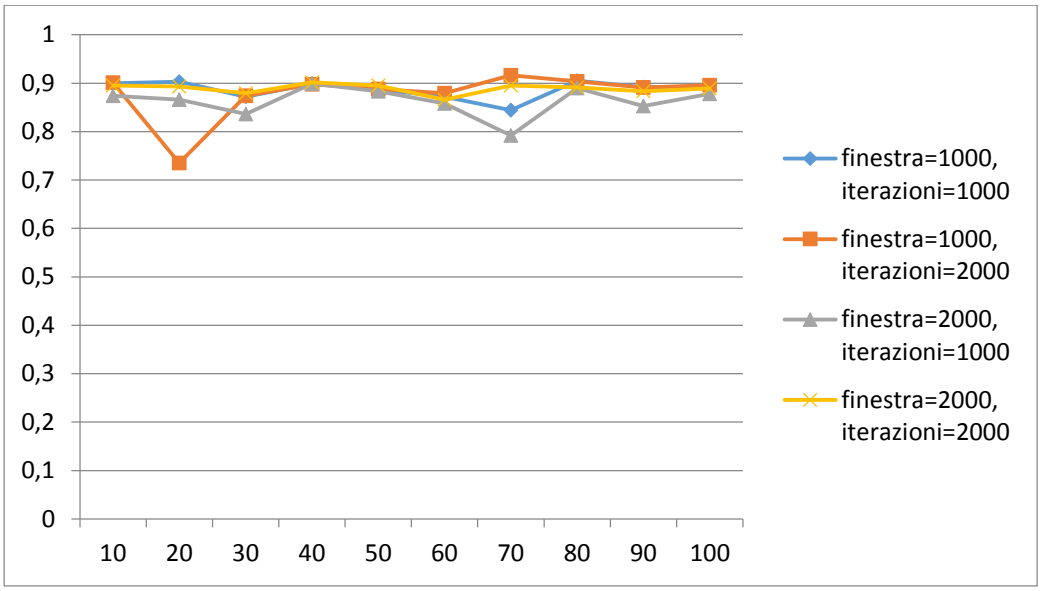


Figura 4.7: comparative indice NMI su EDS con 5% rumore. In ascissa le unità di tempo, in ordinata l'indice NMI

EDS con 10% rumore

Parametri: $o\_soglia = 1{,}5, p\_soglia = 3, \varepsilon = 3, \lambda = 0{,}85$.

Per poter ottenere buoni risultati a fronte di un aumento della rumorosità, è stato necessario diminuire la soglia ε rispetto ai casi precedenti.

finestra iterazioni	1000 1000	punti per finestra
ut	purezza	nmi
10	0,991	0,801
20	0,863	0,659
30	0,955	0,721
40	0,993	0,812
50	0,893	0,693
60	0,904	0,672
70	0,849	0,659
80	0,997	0,818
90	0,942	0,704
100	0,874	0,642

Tabella 4.14

finestra iterazioni	1000 2000	punti per finestra
ut	purezza	nmi
10	1	0,814
20	1	0,827
30	0,996	0,785
40	0,999	0,817
50	1	0,81
60	1	0,805
70	0,997	0,825
80	1	0,822
90	0,998	0,793
100	0,998	0,81

Tabella 4.15

finestra 2000 punti per iterazioni 1000 finestra		
ut	purezza	nmi
10	0,977	0,781
20	0,863	0,638
30	0,809	0,563
40	0,98	0,777
50	0,959	0,718
60	0,817	0,596
70	0,793	0,555
80	0,985	0,784
90	0,753	0,558
100	0,886	0,632

Tabella 4.16

finestra 2000 punti per iterazioni 2000 finestra		
ut	purezza	nmi
10	1	0,82
20	1	0,804
30	0,98	0,74
40	1	0,808
50	1	0,803
60	0,999	0,774
70	0,901	0,7
80	0,998	0,8
90	1	0,784
100	1	0,805

Tabella 4.17

Con una rumorosità del 10%, anche nel primo caso (1000 punti per finestra, 100 iterazioni) abbiamo ottenuto indici di purezza non ottimali. Valgono le considerazioni già fatte per quanto riguarda la serie precendente di esperimenti (rumore 5%): all'aumentare del rumore, sono necessarie più iterazioni per aggregare correttamente. In questo gruppo di esperimenti è stato inoltre usato un parametro ε inferiore, anche questo comporta un aumento delle iterazioni necessarie per la convergenza della soluzione: gli agenti riconosceranno un numero minore di altri agenti come loro simili, rendendo più lenta la propagazione delle etichette di clustering.

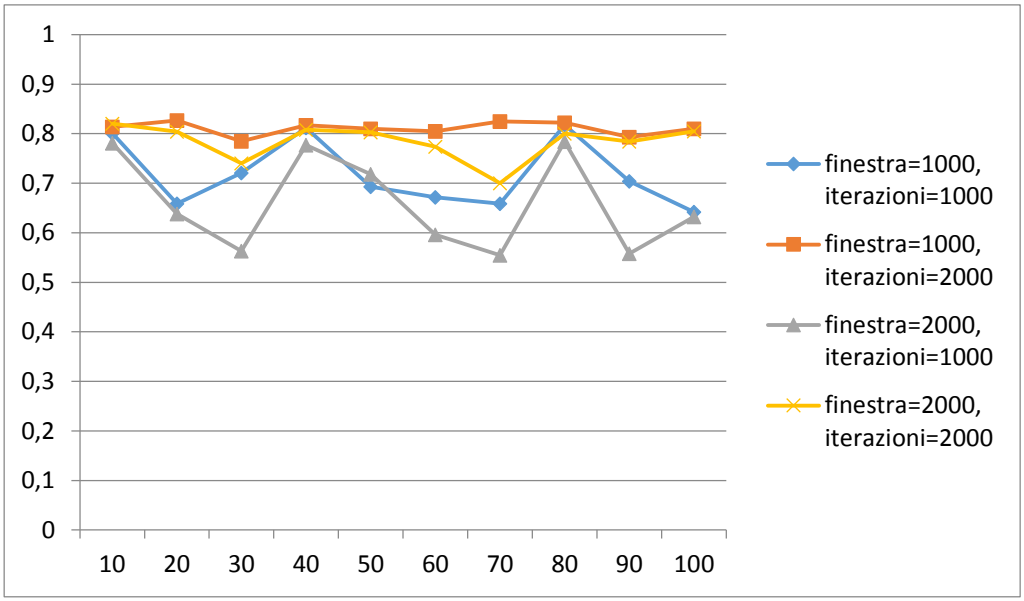


Figura 4.8: comparative indice NMI su EDS con 10% rumore. In ascissa le unità di tempo, in ordinata l'indice NMI

Quanto appena detto viene meglio evidenziato nel grafico in Figura 4.8. Solo gli esperimenti riferiti alle tabelle 4.15 e 4.17, (quadrati rossi e croci viola) ottengono risultati apprezzabili. Il numero di iterazioni utilizzato è maggiore rispetto agli altri casi.

5. Riferimenti

[1] Feng Cao, Martin Ester, Weining Quian, and Aoying Zhou. *Density-based clustering over evolving data stream with noise.* In proceedings of the Sixth SIAM International conference on Data Mining (SIAM'06), pages 326-337, 2006.

[2] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proc of KDD, 1996.

[3] X. Cui, J. Gao, and T. E. Potok. *A flocking based algorithm for document clustering analysis.* Journal of Systems Architecture, 52(8-9):505-515, 2006.

[4] C. W. Reynolds. Flocks, Herds and Schools: A Distributed Behavioral Model. Computer Graphics (ACM) 21, 1987.

[5] A. Forestiero, C. Pizzuti, G. Spezzano. *A Single Pass Algorithm for Clustering Evolving Data Streams based on Swarm Intelligence.* Data Mining and Knowledge Discovery, Springer, 2012, pp. 1-26. doi:10.1007/s10618-011-0242-x Key: citeulike:10003428.

http://www.springerlink.com/content/62256m81228j42t5/fulltext.pdf.

[6] Giandomenico Spezzano. Seminario su Modelli di programmazione parallela per sistemi ibridi Multi/Many Core. CNR-ICAR & DEIS-UNICAL. 2010.

[7] NVIDIA. NVIDIA CUDA C Programming Guide. Version 4.0. 2011.

[8] H. Nguyen. *GPU Gems 3.* Addison-Wesley Professional (August 12, 2007).

[9] A. Forestiero, C. Pizzuti, G. Spezzano, *FlockStream: a Bio-inspired Algorithm for Clustering Evolving Data Streams.* Proceedings of ICTAI'09, Proc. of the 21st International Conference on Tools with Artificial Intelligence (ICTAI'09), Newark, New Jersey, USA, 2009.

[10] X. Cui, J. S. Charles, T. E. Potok. *Graphics Processing Unit Enhanced Parallel Document Flocking Clustering*. Springer Lecture Notes in Computer Science (LNCS), Proceedings of the International Conference on Swarm Intelligence (ICSI'2010), Beijing, China, 2010.

[11] Ian Parberry. *On the Computational Complexity of Optimal Sorting Network Verification.* PARLE '91: Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms, Eindhoven, The Netherlands, June 10-13, 1991, Proceedings: 252–269.

[12] K.E. Batcher. *Sorting networks and their applications.* Proceedings of the AFIPS Spring Joint Computer Conference 32, 307–314, 1968.

[13] G. E. Blelloch. *Scans as Primitive Parallel Operations.* IEEE Transactions on Computers, C-38(11):1526–1538, November 1989.

[14] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. *GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management.* In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 325–336, 2006.

[15] Guy E. Blelloch. *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University. 1990.

[16] X. Cui, J. Gao and T. E. Potok. A Flocking Based Algorithm for Document Clustering Analysis, Journal of System Architecture. June, 2006, ISSN: 1318-7621.

[17] H. Van Dyke Parunak, Richard Rohwer, Theodore C. Belding, Sven Brueckner: *Dynamic Decentralized Any-Time Hierarchical Clustering*. ESOA 2006: 66-81.

[18] A. Forestiero , C. Mastroianni , G. Papuzzo, G. Spezzano. *Towards a Self-Structured Grid: An Ant-Inspired P2P Algorithm.* Transactions on Computational Systems Biology (TC SB) X, LNB I5410, pp. 1-19, Springer Verlag, Berlin, 2008.

[19] NVIDIA. *NVIDIA Kepler GK-110 Architecture Whitepaper.* http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[20] YongChul Kwon, Wing Yee Lee, Magdalena Balazinska, Guiping Xu. *Clustering Events on Streams Using Complex Context Information.* Proceeding of ICDMW '08 Proceedings of the 2008 IEEE International Conference on Data Mining Workshops, pages 238-247.