# Midterm Parallel - Boids

Edoardo Fanciullacci

April 2025

## 1 Introduzione

Lo studio dei sistemi complessi e dei comportamenti collettivi ha un ruolo centrale in numerosi ambiti, dall'intelligenza artificiale alla simulazione di fenomeni naturali. Un esempio emblematico è rappresentato dal cosiddetto *flocking*, ovvero il comportamento collettivo osservabile in gruppi di agenti intelligenti — come stormi di uccelli o banchi di pesci — che si muovono in maniera coordinata e apparentemente organizzata, pur in assenza di un leader.

L'obiettivo di questo progetto è la simulazione computazionale di tale comportamento, attraverso un modello semplificato noto come **Boids**. In questo modello, ogni agente (o "boid") prende decisioni individuali basandosi unicamente sull'osservazione dei boid vicini, seguendo tre semplici regole di interazione locale:

- **Separazione** (*Separation*): evitare il sovraffollamento, mantenendo una certa distanza dagli altri boid;
- **Allineamento** (*Alignment*): dirigersi nella stessa direzione dei boid vicini;
- **Coesione** (*Cohesion*): avvicinarsi al centro del gruppo locale.

Queste tre forze agiscono simultaneamente su ciascun boid, determinandone l'accelerazione e quindi l'evoluzione dinamica. Nonostante la semplicità delle regole, il comportamento emergente è sorprendentemente realistico e riproduce fedelmente i movimenti fluidi e coordinati osservabili in natura.

Nel presente lavoro sono stati implementati e confrontati diversi approcci computazionali per la simulazione dei boid, con particolare attenzione all'ottimizzazione delle prestazioni attraverso tecniche di parallelizzazione e strutture dati efficienti.

## 2 Analisi della versione sequenziale (AoS)

La versione sequenziale del simulatore di boid adotta una rappresentazione **Array of Structures (AoS)**, in cui ogni agente è modellato come una struttura `Boid` che incapsula le sue proprietà fondamentali: `position` e `velocity`, entrambe di tipo `Vector2`.

## 2.1 Strutture dati fondamentali

Nel file `BoidsCommon.hpp` sono definite le costanti globali della simulazione (come le dimensioni del dominio, le forze massime ammissibili e i raggi di interazione) e le strutture dati principali. La classe `Vector2` fornisce un'interfaccia compatta per operazioni vettoriali bidimensionali (somma, sottrazione, prodotto scalare, normalizzazione, etc.), spesso utilizzate nei calcoli di forza tra agenti.

La struttura `Boid` è:

```
struct Boid {
    Vector2 position;
    Vector2 velocity;
};
```

Questa rappresentazione AoS è semplice e leggibile, ma presenta alcune limitazioni dal punto di vista della performance, specialmente in contesti di parallelizzazione e accesso alla memoria (vedi sezioni successive).

## 2.2 Regole comportamentali

Il file `BoidsUpdate.hpp` implementa le tre regole fondamentali del comportamento dei boid, proposte da Craig Reynolds:

- **Separazione**: evita che i boid si avvicinino troppo (`separation()`).
- **Allineamento**: orienta il boid nella direzione media dei vicini (`alignment()`).
- **Coesione**: tende a riunire il boid al centro del gruppo locale (`cohesion()`).

Ogni funzione riceve in input un boid `b` e lo stato corrente del sistema `current`, e restituisce un vettore di "steering" da applicare. Esempio di frammento in `separation`:

```
for (const auto& other :const Boid& : current) {
    if (const float dist = (b.position - other.position).magnitude();
        &b != &other && dist < SEPARATION_RADIUS) {
        Vector2 diff = (b.position - other.position).normalized();
        diff = diff / (dist * dist);
        steer = steer + diff;
        count++;
    }
}
```

Figure 1: Funzione di separazione

Tutte e tre le funzioni implementano un'iterazione sull'intero stato `current`, e filtrano i vicini validi sulla base del raggio di interazione. Questa struttura induce un costo computazionale $\mathcal{O}(N^2)$.

### 2.3 Aggiornamento dello stato

La funzione `computeNextBoid()` combina le tre forze calcolate pesandole con costanti definite:

```
Vector2 acceleration = sep + ali + coh;
acceleration.limit(MAX_FORCE);
velocity = b.velocity + acceleration;
velocity.limit(MAX_SPEED);
position = b.position + velocity;
```

Figure 2: frammento di ComputeNextBoid

Il risultato è uno schema di integrazione esplicita, in cui le nuove velocità e posizione sono ottenute direttamente dalle precedenti e dalla forza risultante. Infine, viene applicato un *wrap-around* spaziale per garantire la periodicità del dominio.

```
constexpr float margin = 5.0f;
if (position.x < -margin) position.x = WIDTH + margin;
else if (position.x > WIDTH + margin) position.x = -margin;
if (position.y < -margin) position.y = HEIGHT + margin;
else if (position.y > HEIGHT + margin) position.y = -margin;
```

### 2.4 Considerazioni

Questa implementazione costituisce la base di riferimento per lo sviluppo delle versioni ottimizzate. Pur essendo chiara e facilmente estendibile, soffre di:

- **Scarsa locality della cache**: l'accesso ai dati (strutture complesse) è frammentato;
- **Scarso parallelismo implicito**: difficile sfruttare SIMD in modo efficiente;
- **Costo computazionale elevato** per l'interazione boid-boid ($\mathcal{O}(N^2)$).

## 3 Ottimizzazione tramite Uniform Grid

Per ridurre il costo computazionale della ricerca dei vicini, il progetto implementa un **partizionamento spaziale della scena** usando una struttura detta `UniformGrid` (`BoidsGrid.hpp`). Ogni boid viene assegnato a una cella della griglia in base alla propria posizione, e le interazioni vengono calcolate solo con gli altri boid presenti nella cella corrente o in quelle adiacenti.

## 3.1 Struttura dati `UniformGrid`

La griglia è definita da due dimensioni (`cellCountX`, `cellCountY`) e un array di celle, dove ciascuna cella contiene gli indici dei boid che vi risiedono. Inoltre, si mantiene un vettore `usedCells` per tracciare solo le celle effettivamente utilizzate, migliorando l'efficienza dell'operazione di `clear()`:

Listing 1: Struttura dati UniformGrid

```
struct UniformGrid {
    int cellCountX, cellCountY;
    std::vector<std::vector<int>> cells;
    std::vector<int> usedCells;
};
```

Per quanto riguarda il vettore `usedCells`, questo permette un **clear selettivo**, evitando di azzerare l'intera struttura ad ogni passo:

```
void clear() {
    for (const int idx : usedCells) {
        cells[idx].clear();
    }
    usedCells.clear();
}
```

Figure 3: Clear selettivo della griglia

La funzione `buildGrid` itera su tutti i boid, calcola le coordinate della cella corrispondente usando `getCellCoords`, e inserisce l'indice del boid nella cella appropriata.

```
for (int i = 0; i < static_cast<int>(oldState.size()); ++i) {
    auto [cellX :int, cellY :int] = getCellCoords(oldState[i].position, cellSize,
        grid.cellCountX, grid.cellCountY);
    const int cellIndex = grid.getCellIndex(cellX, cellY);
    grid.cells[cellIndex].push_back(i);
    grid.usedCells.push_back(cellIndex);
}
```

Figure 4: Frammento buildGrid

## 3.2 Ricerca locale dei vicini

Per ciascun boid, la ricerca dei vicini è effettuata attraverso la funzione `forEachNeighborBoid`, che ispeziona solo le 9 celle del *Moore neighborhood* (quella centrale più le 8 adiacenti), riducendo drasticamente il numero di confronti rispetto alla versione sequenziale.

```
for (int dx = -1; dx <= 1; dx++) {
    for (int dy = -1; dy <= 1; dy++) {
        const int nx = cellX + dx;
        const int ny = cellY + dy;

        if (nx < 0 || nx >= grid.cellCountX) continue;
        if (ny < 0 || ny >= grid.cellCountY) continue;

        for (const int neighborCellIndex = grid.getCellIndex( cellX: nx,  cellY: ny);
            const int otherIdx : grid.cells[neighborCellIndex]) {
            callback(otherIdx);
        }
    }
}
```

Figure 5: Scansione delle celle adiacenti

### 3.3 Esempio: `separationGrid` con Uniform Grid

La funzione `separationGrid` calcola la forza di separazione per un boid $b$ limitando il confronto ai boid nella sua cella e nelle 8 celle adiacenti, riducendo drasticamente il numero di confronti rispetto a una soluzione $\mathcal{O}(n^2)$. Questo approccio migliora l'efficienza mantenendo una buona qualità del comportamento simulato.

```cpp
inline Vector2 separationGrid(const Boid& b,
                              const int boidIndex,
                              const std::vector<Boid>& current,
                              const UniformGrid& grid,
                              const int cellX, const int cellY)
{
    Vector2 steer{ x_: 0, y_: 0};
    int count = 0;

    forEachNeighborBoid(cellX, cellY, grid, callback: [&](const int otherIdx) ->void {
        if (otherIdx == boidIndex) return;
        const auto&[position :const Vector2 , velocity :const Vector2 ] = current[otherIdx];
        if (const float dist = (b.position - position).magnitude();
            dist < SEPARATION_RADIUS) {
            Vector2 diff = (b.position - position).normalized();
            diff = diff / (dist * dist);
            steer = steer + diff;
            count++;
        }
    });

    if (count > 0) {
        steer = steer / static_cast<float>(count);
        steer = steer.normalized() * MAX_SPEED;
        steer = steer - b.velocity;
        steer.limit(MAX_FORCE);
    }
    return steer;
}
```

Figure 6: separationGrid

Questa funzione sfrutta la funzione ausiliaria `forEachNeighborBoid`, che applica la logica desiderata (`separation`, `alignment`, `cohesion`, ecc.) solo ai boid rilevanti, cioè quelli nella regione di interesse attorno al boid corrente.

## 3.4 Applicazione delle forze

Le funzioni `separationGrid`, `alignmentGrid` e `cohesionGrid` sono versioni ottimizzate che operano solo sui boid localizzati nelle celle rilevanti. Queste forze sono combinate nella funzione `computeNextBoidGrid`, che aggiorna posizione e velocità del boid target.

Listing 2: Calcolo aggiornamento tramite griglia

```
const Vector2 sep = separationGrid(...);
const Vector2 ali = alignmentGrid(...);
const Vector2 coh = cohesionGrid(...);
Vector2 acceleration = sep + ali + coh;
```

### 3.5 Vantaggi

Questa tecnica permette di ridurre la complessità da $\mathcal{O}(n^2)$ a $\mathcal{O}(n)$, assumendo una distribuzione omogenea degli agenti. Inoltre, si presta molto bene a una parallelizzazione successiva poiché l'accesso alle celle è localizzato e prevedibile.

## 4 Versione SoA (Structure of Arrays)

La versione **SoA** del simulatore Boids rappresenta un'evoluzione rispetto al modello tradizionale AoS (Array of Structures). Invece di rappresentare ciascun boid come un oggetto con posizione e velocità incapsulate in una singola struttura, la SoA separa i dati in array distinti: uno per ogni componente della posizione e della velocità.

Listing 3: Definizione della struttura BoidSoA

```
struct BoidSoA {
    std::vector<float> posX, posY;
    std::vector<float> velX, velY;

    explicit BoidSoA(int n) {
        posX.resize(n); posY.resize(n);
        velX.resize(n); velY.resize(n);
    }
};
```

### 4.1 Vantaggi teorici dell'approccio SoA

L'approccio SoA è spesso preferito per le seguenti ragioni:

- **Cache Locality**: accedendo sequenzialmente ad array monodimensionali, si migliora il locality di cache rispetto a strutture complesse con salti di memoria.
- **Vettorizzazione**: i compilatori possono applicare ottimizzazioni SIMD (*Single Instruction, Multiple Data*) più facilmente, aumentando l'efficienza computazionale.
- **Parallellismo fine**: si facilita l'uso di istruzioni SIMD e di direttive OpenMP per riduzioni esplicite.

### 4.2 Applicazione delle forze con direttive SIMD

Tutte le funzioni di aggiornamento delle forze (separation, alignment, cohesion) utilizzano direttive `#pragma omp simd` e meccanismi di *riduzione esplicita*, ovvero una forma di accumulo parallelo.

```
#pragma omp simd reduction(+:sumX, sumY, count)
for (int j = 0; j < numBoids; ++j) {
    if (i == j) continue;

    const float dx = selfX - current.posX[j];
    const float dy = selfY - current.posY[j];

    if (const float dist = std::sqrt( x: dx * dx + dy * dy); dist < VIEW_RADIUS) {
        const float weight = (VIEW_RADIUS - dist) / VIEW_RADIUS;
        sumX += current.velX[j] * weight;
        sumY += current.velY[j] * weight;
        count++;
    }
}
```

Figure 7: Esempio di riduzione SIMD nella funzione alignment

Le riduzioni sono un costrutto chiave in ambienti SIMD: permettono di accumulare risultati parziali (es. somme o conteggi) in parallelo in modo sicuro e deterministico.

## 4.3 Aggiornamento dello stato

Il risultato delle tre forze viene sommato, limitato, e infine applicato per aggiornare la posizione e la velocità del boid:

```
const Vector2 sep = separation(i, oldState, numBoids) * SEPARATION_WEIGHT;
const Vector2 ali = alignment(i, oldState, numBoids) * ALIGNMENT_WEIGHT;
const Vector2 coh = cohesion(i, oldState, numBoids) * COHESION_WEIGHT;

Vector2 acceleration = sep + ali + coh;
acceleration.limit(MAX_FORCE);

Vector2 velocity{oldState.velX[i], oldState.velY[i]};
velocity = velocity + acceleration;
velocity.limit(MAX_SPEED);

Vector2 position{oldState.posX[i], oldState.posY[i]};
position = position + velocity;
...
newState.posX[i] = position.x;
newState.posY[i] = position.y;
newState.velX[i] = velocity.x;
newState.velY[i] = velocity.y;
```

Figure 8: Aggiornamento della posizione e velocità in SoA (computeNextBoidSOA)

### 4.4 Conclusioni sulla SoA

Il modello SoA rappresenta un compromesso ideale tra flessibilità e performance. È particolarmente efficace in scenari con un elevato numero di boid grazie alla possibilità di sfruttare al meglio la memoria e le unità SIMD del processore. La separazione delle componenti consente inoltre un parallelismo più fine e scalabile.

## 5 Analisi dei programmi principali

### 5.1 Versione sequenziale/AoS

La versione di base del simulatore utilizza la rappresentazione AoS e una semplice struttura iterativa per simulare il comportamento dei boid. Pur non essendo ottimizzata per la cache o la parallelizzazione profonda, questa versione rappresenta un punto di partenza utile per comprendere la logica della simulazione.

#### 5.1.1 Parsing e allocazione

Il programma riceve da linea di comando il numero di boid da simulare. Viene effettuata una validazione del valore e successivamente si allocano due vettori:

- `oldState` — contiene lo stato corrente dei boid.
- `newState` — contiene lo stato futuro che sarà calcolato ad ogni step.

`newState` viene inizializzato in parallelo con OpenMP, sfruttando il principio del *first-touch* per ottimizzare la disposizione dei dati in memoria NUMA.

### 5.1.2 Inizializzazione della scena

I boid vengono disposti su una griglia regolare e direzionati radialmente verso l'esterno:

```
const int gridSize = static_cast<int>(std::ceil(std::sqrt(numBoids)));
const float spacingX = WIDTH / static_cast<float>(gridSize);
const float spacingY = HEIGHT / static_cast<float>(gridSize);
constexpr float centerX = WIDTH / 2.0f;
constexpr float centerY = HEIGHT / 2.0f;

//  ogni boid viene posizionato sulla griglia e riceve velocità radiale
#pragma omp parallel for schedule(static)
for (int i = 0; i < numBoids; ++i) {
    const int row = i / gridSize;
    const int col = i % gridSize;
    const float posX = static_cast<float>(col) * spacingX + spacingX / 2.0f;
    const float posY = static_cast<float>(row) * spacingY + spacingY / 2.0f;

    oldState[i].position = { posX, posY };

    const float angle = std::atan2(posY - centerY, posX - centerX);
    oldState[i].velocity = { std::cos(angle) * MAX_SPEED,
        std::sin(angle) * MAX_SPEED };
}
```

Figure 9: Inizializzazione regolare dei boid

### 5.1.3 Simulazione

La simulazione si sviluppa in una regione parallela OpenMP. Ad ogni step temporale:

```
#pragma omp parallel default(none) shared(oldState, newState, numBoids)
{
    constexpr int STEPS = 600;
    for (int step = 0; step < STEPS; ++step) {
        // Calcolo parallelo del nuovo stato
        #pragma omp for schedule(static)
        for (int i = 0; i < numBoids; ++i) {
            computeNextBoid(i, oldState, newState);
        }

        // Swap sequenziale tra i due buffer
        #pragma omp single
        {
            oldState.swap(newState);
        }
    }
}
```

Figure 10: Regione parallela principale

1. `computeNextBoid` calcola posizione e velocità futura di ciascun boid (in parallelo).
2. Si effettua lo **swap** tra `oldState` e `newState` usando una sezione `single`, per evitare race conditions.

Lo *swap* consente di riutilizzare le stesse strutture dati, evitando riallocazioni e mantenendo le prestazioni costanti durante le iterazioni.

### 5.1.4 Misurazione del tempo

Al termine della simulazione, viene misurato il tempo totale impiegato con l'ausilio di `std::chrono`, permettendo un confronto diretto tra le versioni ottimizzate.

```
const auto start = std::chrono::high_resolution_clock::now();
// Simulazione ...
const auto end = std::chrono::high_resolution_clock::now();
```

**Conclusione**: Questa versione mostra un'implementazione chiara e leggibile della logica base del simulatore, pur rivelando limiti in termini di scalabilità e locality della memoria, che saranno affrontati nelle versioni successive.

## 5.2 Versione con griglia uniforme

Come già visto, questa versione introduce una struttura dati spaziale, la `UniformGrid`, per ottimizzare la località e ridurre la complessità delle interazioni tra boid. In-

vece di confrontare ogni boid con tutti gli altri (approccio $O(n^2)$), ciascun boid valuta solo i vicini presenti nella propria cella e in quelle adiacenti.

### 5.2.1 Costruzione parallela della griglia

La costruzione della griglia è effettuata in tre fasi:

1. Azzeramento delle celle effettivamente utilizzate al passo precedente.
2. Inserimento parallelo dei boid in celle locali per thread.
3. Merge finale in parallelo, con accesso protetto a `usedCells`.

```
#pragma omp for schedule(static)
for (int i = 0; i < N; ++i) {
    auto [cellX, cellY] = getCellCoords(oldState[i].position,
        cellSize, grid.cellCountX, grid.cellCountY);
    const int cellIndex = grid.getCellIndex(cellX, cellY);
    localCells[cellIndex].push_back(i);
}
```

Figure 11: Build parallelo ottimizzato

Questo schema evita la sincronizzazione nel loop interno e sfrutta la cache locality intra-thread.

### 5.2.2 Struttura del main

La simulazione è svolta in una regione parallela. Ad ogni iterazione temporale:

- `parallelBuildGrid` costruisce la griglia in una sezione `single`.
- Ogni thread aggiorna una porzione dello stato usando `computeNextBoidGrid`.
- I buffer `oldState` e `newState` vengono scambiati.

Listing 4: Loop della simulazione

```
#pragma omp single
parallelBuildGrid(...);
...
#pragma omp for schedule(static)
computeNextBoidGrid(...);
...
#pragma omp single
oldState.swap(newState);
```

### 5.2.3 Conclusione

La griglia uniforme permette di ridurre drasticamente il numero di confronti tra boid, rendendo la simulazione più scalabile. Questo approccio migliora sia le prestazioni che la cache locality, ed è particolarmente efficace in combinazione con parallelismo e accessi coalescenti alla memoria.

## 5.3 Versione SoA (Structure of Arrays)

La versione SoA rappresenta una strategia di ottimizzazione orientata alla memoria, progettata per migliorare l'efficienza di accesso ai dati in contesti fortemente numerici o paralleli. A differenza dell'approccio *Array of Structures (AoS)*, in cui ogni boid è rappresentato da un oggetto contenente posizione e velocità, la struttura SoA separa i dati in array distinti: `posX`, `posY`, `velX` e `velY`.

### 5.3.1 Vantaggi teorici

Questo approccio migliora la **località spaziale** e consente un utilizzo più efficiente delle *SIMD instructions*, facilitando operazioni di tipo vettoriale. Inoltre, l'accesso separato e contiguo a ciascun campo permette di applicare *riduzioni esplicite*, che ottimizzano le somme parallele con OpenMP.

### 5.3.2 Inizializzazione parallela dello stato

Prima della simulazione vera e propria, ogni boid viene posizionato su una griglia regolare e riceve una velocità iniziale radiale centrata rispetto all'origine. Anche questa fase è parallelizzata, sfruttando la politica di *first-touch* per allocare in modo ottimale la memoria sui core:

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < numBoids; ++i) {
    const int row = i / gridSize;
    const int col = i % gridSize;
    const float posX = static_cast<float>(col) * spacingX + spacingX / 2.0f;
    const float posY = static_cast<float>(row) * spacingY + spacingY / 2.0f;

    oldState.posX[i] = posX;
    oldState.posY[i] = posY;

    const float angle = std::atan2(posY - centerY, posX - centerX);
    oldState.velX[i] = std::cos(angle) * MAX_SPEED;
    oldState.velY[i] = std::sin(angle) * MAX_SPEED;
}
```

Figure 12: nizializzazione parallela deterministica

Questa inizializzazione produce una configurazione regolare che simula una condizione iniziale ordinata, utile per osservare l'evoluzione dinamica dell'interazione

tra agenti.

### 5.3.3 Loop principale

Come nelle versioni precedenti, la simulazione avviene in una regione `omp parallel`. Tuttavia, qui il calcolo è interamente vettorializzato e cache-friendly:

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < numBoids; ++i) {
    const int row = i / gridSize;
    const int col = i % gridSize;
    const float posX = static_cast<float>(col) * spacingX + spacingX / 2.0f;
    const float posY = static_cast<float>(row) * spacingY + spacingY / 2.0f;

    oldState.posX[i] = posX;
    oldState.posY[i] = posY;

    const float angle = std::atan2(posY - centerY, posX - centerX);
    oldState.velX[i] = std::cos(angle) * MAX_SPEED;
    oldState.velY[i] = std::sin(angle) * MAX_SPEED;
}
```

Figure 13: Main Loop SOA

### 5.3.4 Conclusione

L'adozione del modello SoA rappresenta un passo significativo verso l'ottimizzazione architetturale, riducendo i cache miss e massimizzando il throughput computazionale. Questa struttura è ideale per scenari ad alta intensità di calcolo, tipici di simulazioni biologiche, fisiche o GPU-like.

## 6 Analisi delle Performance

Per valutare l'efficacia delle diverse strategie implementative (NoGrid, Grid, SoA), sono stati condotti esperimenti mirati al confronto dello *speedup* ottenuto in funzione del numero di boid simulati e del numero di thread utilizzati. I risultati sono riportati in Figura 14.

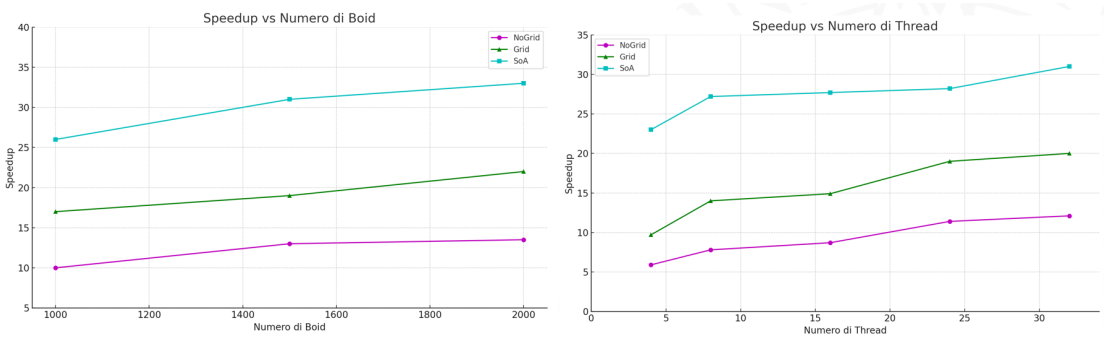


Figure 14: Confronto dello speedup rispetto alla versione sequenziale in funzione del numero di boid (sinistra) e del numero di thread (destra).

## 6.1 Scalabilità con il numero di agenti

Nel grafico di sinistra, si osserva come tutte le versioni beneficino di un incremento del numero di agenti. Tuttavia, l'entità dello speedup varia notevolmente:

- **NoGrid**: la versione base, priva di ottimizzazioni strutturali, mostra uno scaling molto limitato. Il suo algoritmo *naive* con complessità $O(n^2)$ e la scarsa località spaziale penalizzano sia l'efficienza cache che la scalabilità parallela.
- **Grid**: l'introduzione di una griglia uniforme permette un culling spaziale efficace, riducendo le interazioni da considerare. Il miglioramento prestazionale rispetto a NoGrid è evidente, e cresce con il numero di boid.
- **SoA**: si conferma la strategia più efficiente. La struttura dati ottimizzata facilita l'uso di istruzioni SIMD, riduce i cache miss e consente riduzioni parallele esplicite con `#pragma omp simd reduction`. Lo speedup raggiunge valori oltre 30×.

## 6.2 Scalabilità con il numero di thread

Nel grafico di destra si analizza la scalabilità forte rispetto al numero di thread:

- **NoGrid** scala in modo limitato e mostra saturazione già a 16 thread. Il collo di bottiglia principale è legato alla memoria e all'accesso inefficiente ai dati.
- **Grid** mostra una scalabilità migliore, grazie alla parallelizzazione del processo di `buildGrid` e alla località spaziale. Tuttavia, l'accesso irregolare alla griglia introduce contese e carichi sbilanciati tra thread.
- **SoA** scala in modo quasi ideale fino a 32 thread. La perfetta separazione tra dati e l'accesso contiguo abilitano un uso efficiente delle risorse SIMD e della banda memoria.

## 6.3 Conclusione

Le evidenze sperimentali mostrano che la ristrutturazione dei dati gioca un ruolo fondamentale nell'ottimizzazione di simulazioni computazionalmente intensive. In particolare, la struttura SoA rappresenta una scelta ottimale per architetture multicore con supporto SIMD, mentre la Grid è un compromesso efficace quando si vuole ridurre la complessità computazionale delle interazioni.