UNIVERSITÀ
DEGLI STUDI
FIRENZE

PARALLEL K-MEANS CLUSTERING ALGORITHM

Miruna Alexandrescu 7163599

UNIVERSITÀ DEGLI STUDI FIRENZE

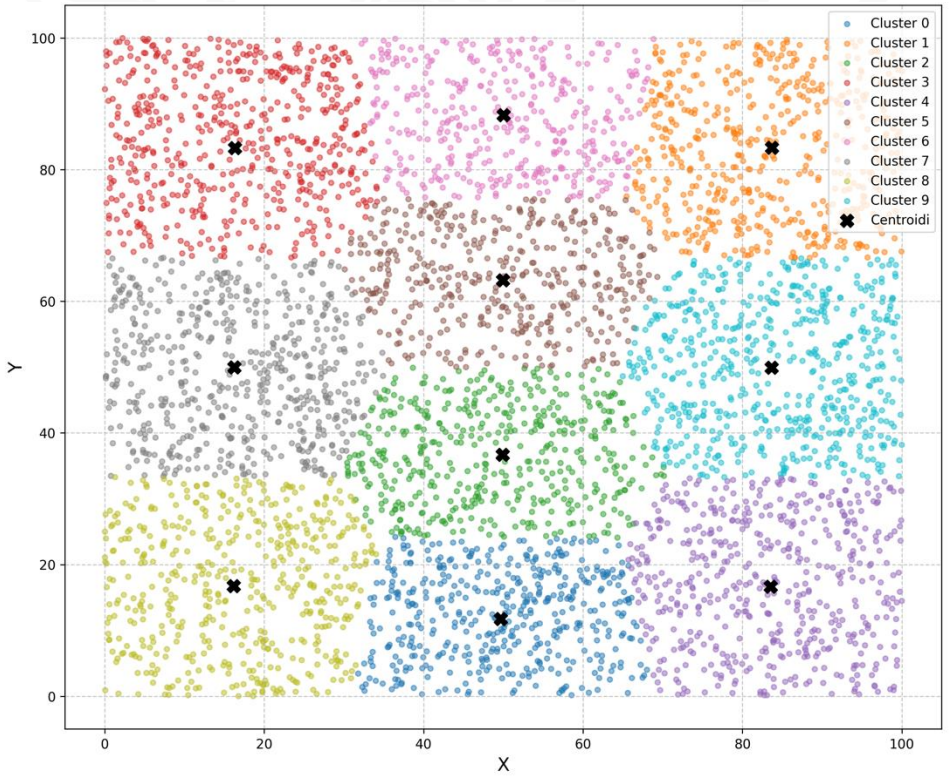
OBIETTIVO → KMEANS → CODICE → RISULTATI → CONSCLUSIONI

- **IMPLEMENTARE** l'algorimo K-means clustering in C++
- **PARALLELIZZARE** l'algoritmo
- **VALUTARE** speedup ed efficienza rispetto alla versione sequenziale

KMEANS

Dato un dataset di punti e un numero k di cluster assegna i punti ai cluster e genera la posizione dei centroidi.

- **Inizializzazione**: scelta k centroidi casuali
- **Assegnazione**: assegna ogni punto al centroide più vicino
- **Aggiornamento**: ricalcola i centroidi come media dei punti assegnati
- **Iterazione**: ripetere fino a convergenza



IMPLEMENTAZIONE SEQUENZIALE

```
// Assegna punti ai cluster
for (auto& point : points) {
    double minDist = numeric_limits<double>::max
        ();
    int bestCluster = -1;

    for (int j = 0; j < k; j++) {

        double dist = point.distanceSquared(
            centroids[j]);
        if (dist < minDist) {
            minDist = dist;
            bestCluster = j;
        }
    }

    if (point.cluster != bestCluster) {
        point.cluster = bestCluster;
        changed = true;
    }
}
```

```
// Aggiorna centroidi
if (changed) {
    vector<double> sumX(k, 0.0), sumY(k, 0.0);
    vector<int> count(k, 0);

    for (const auto& point : points) {
        sumX[point.cluster] += point.x;
        sumY[point.cluster] += point.y;
        count[point.cluster]++;
    }

    for (int j = 0; j < k; j++) {
        if (count[j] > 0) {
            centroids[j].x = sumX[j] / count[j];
            centroids[j].y = sumY[j] / count[j];
        }
    }
}
```

IMPLEMENTAZIONE PARALLELA

```
#ifdef USE_OPENMP
    omp_set_num_threads(numThreads);
#endif

#pragma omp parallel
{
    vector<double> localSumX(k, 0.0), localSumY(k
        , 0.0);
    vector<int> localCount(k, 0);
    bool localChanged = false;

    #pragma omp for
    for (size_t i = 0; i < points.size(); i++) {
        // Point assignment logic here
    }
}
```

Crea i thread paralleli

Variabili locali per thread: evitano race condition

```
#pragma omp for
for (size_t i = 0; i < points.size(); i++) {
    double minDist = numeric_limits<double>::max
        ();
    int bestCluster = -1;

    for (int j = 0; j < k; j++) {
        double dist = points[i].distanceSquared(
            centroids[j]);
        if (dist < minDist) {
            minDist = dist;
            bestCluster = j;
        }
    }

    if (points[i].cluster != bestCluster) {
        points[i].cluster = bestCluster;
        localChanged = true;
    }

    localSumX[bestCluster] += points[i].x;
    localSumY[bestCluster] += points[i].y;
    localCount[bestCluster]++;
}
```

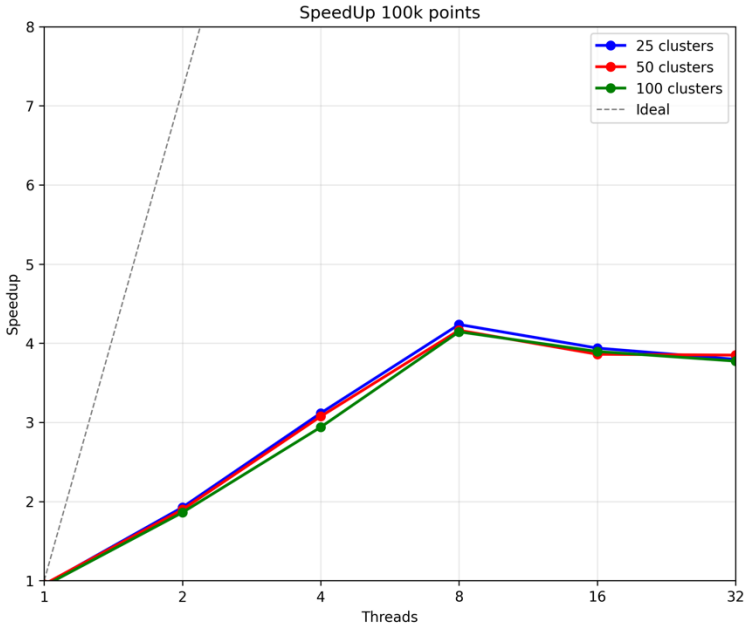
Parallelizza il loop principale

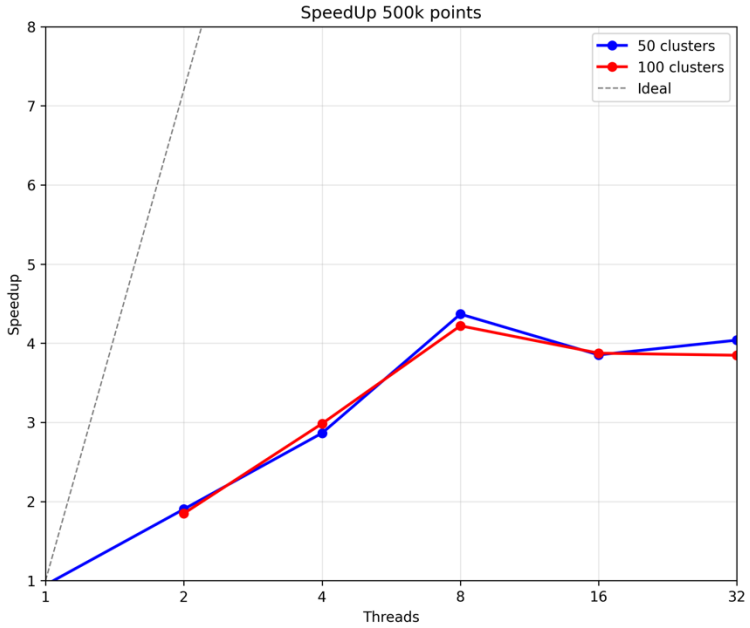
```
#pragma omp critical
{
    if (localChanged) changed = true;
    for (int j = 0; j < k; j++) {
        sumX[j] += localSumX[j];
        sumY[j] += localSumY[j];
        count[j] += localCount[j];
    }
}
```

Sincronizzazione per combinare i risultati

RISULTATI

Model	MacBook Pro 13-inch, M1, 2020
Processor	Apple M1 chip (8-core CPU)
	4 performance + 4 efficiency cores
Memory	16 GB unified memory





Efficienza

2 thread	**95.5%**
4 thread	**76.8%**
8 thread	**53.4%**
> 8 thread	**diminuzione**

CONCLUSIONI

- **Speedup ottimale con 8 thread** corrispondente ai core fisici del sistema
- **Degradazione delle prestazioni** oltre 8 thread dovuta all'overhead di gestione
- **Parallelizzazione efficace** del clustering K-means