Laboratorio Computazionale

Implementazione di un algoritmo distribuito per il disegno di grafi

Francesco Romeo 634705

PSEUDOCODICE

Per ogni iterazione in questo algoritmo ci mette Θ (| V |^2 + | E |), dove:

Forze Repulsive: O(|V|^2), dove |V| è il numero dei nodi.

Forze Attrattive: O(|E|), dove |E| è il numero degli archi.

Nella mia implementazione dell'algoritmo, ho migliorato le prestazioni utilizzando il parallelismo per distribuire il calcolo delle forze repulsive e attrattive tra vari core.

Questo approccio è stato fondamentale per ottimizzare l'esecuzione complessiva dell'algoritmo su macchine moderne con più core disponibili, o su cluster di macchine.

Aspetto la fine di tutti i core prima di procedere con l'aggiornamento delle posizioni dei nodi, garantendo che i calcoli paralleli siano completati prima di passare alla fase successiva.

```
area := W * L; { W and L are the width and length of the frame }
G := (V, E); { the vertices are assigned random initial positions }
k := sqrt(area/|V|);
function f_a(z) := begin return x^2/k end;
function f_r(z) := begin return k^2/z end;

for i := 1 to iterations do begin
    { calculate repulsive forces}
    for v in V do begin
        { each vertex has two vectors: .pos and .disp }
        v.disp := 0;
        for u in V do
            if (u # v) then begin
                { Δ is short hand for the difference}
                { vector between the positions of the two vertices )
                Δ := v.pos - u.pos;
                v.disp := v.disp + ( Δ /| Δ |) * f_r(| Δ |)
            end
    end

    { calculate attractive forces }
    for e in E do begin
        { each edge is an ordered pair of vertices .v and .u }
        Δ := e.v.pos – e.u.pos
        e.v.disp := e.v.disp – ( Δ/| Δ |) * f_a(| Δ |);
        e.u. disp := e.u.disp + ( Δ /| Δ |) * f_a(| Δ |)
    end

    { limit the maximum displacement to the temperature t }
    { and then prevent from being displaced outside frame}
    for v in V do begin
        v.pos := v.pos + ( v. disp/ |v.disp|) * min ( v.disp, t );
        v.pos.x := min(W/2, max(-W/2, v.pos.x));
        v.pos.y := min(L/2, max(–L/2, v.pos.y))
    end
    { reduce the temperature as the layout approaches a better configuration }
    t := cool(t)
end
```

Implementazione

I codici, in c, servono a visualizzare e simulare la creazione di un grafo utilizzando la versione multithread dell'algoritmo di Fruchterman-Reingold.

Ho usato librerie standard, più librerie extra come quella per visualizzare il grafo (SDL2), e per il calcolo parallelo (MPI).

Le strutture dati utilizzate sono le seguenti:

```c
typedef struct {
    double x;
    double y;
} Force;

typedef struct {
    double x;
    double y;
} Node;

typedef struct {
    size_t start;
    size_t end;
} Edge;
```

```c
typedef struct {
    Node* nodes;
    Edge* edges;
    size_t node_count;
    size_t edge_count;
} SimpleGraph;
```

La struttura **Force** rappresenta la forza esercitata su un nodo, con due variabili di tipo double, x e y, che indicano rispettivamente la componente della forza in direzione orizzontale e verticale.

La struttura **Node** descrive la posizione di un nodo nello schermo, utilizzando due variabili di tipo double, x e y, che specificano le coordinate del nodo nello schermo.

La struttura **Edge** rappresenta un arco che collega due nodi in un grafo, questa struttura contiene due variabili di tipo size_t, start e end, che indicano gli indici dei nodi di partenza e arrivo dell'arco.

La struttura **SimpleGraph** rappresenta un grafo composto da nodi e archi, include quattro variabili: node_count e edge_count, entrambe di tipo size_t, che indicano il numero totale di nodi e archi nel grafo, e due puntatori, Node* nodes e Edge* edges, che puntano rispettivamente agli array dei nodi e degli archi.

Il codice richiede quattro variabili di input:

Il **nome del file**, che specifica da dove verranno letti i dati del grafo. Il programma verifica l'esistenza del file e, se non esiste, restituisce un errore e termina l'esecuzione.

Il **numero di iterazioni**, che determina quante volte le forze verranno applicate al grafo. Questo valore può essere incrementato durante l'esecuzione del programma se necessario, per garantire la stabilità del grafo.

La **temperatura**, che regola la libertà di movimento dei nodi nel grafo. All'inizio, fornisce un'ampia libertà di movimento, ma diminuisce gradualmente nel corso delle iterazioni, riducendo la mobilità dei nodi man mano che il grafo si stabilizza.

Il **peso**, che indica se nel file sono specificati i pesi degli archi. Se i pesi sono presenti, il programma li leggerà dal file; altrimenti, tratterà tutti gli archi come non pesati.

All’interno del main troviamo un ciclo che itera per un numero di iterazioni, **it**.

In ogni iterazione, viene eseguita una serie di operazioni per aggiornare il grafo.

Prima del ciclo, viene inizializzato a zero un vettore delle forze, **net_forces**, che conterrà le forze risultanti per ciascun nodo del grafo. Successivamente, si calcolano le posizioni massime dei nodi per ottenere una scala che si adatta allo schermo.

Per calcolare le forze repulsive tra i nodi, a ogni core viene dato un range di nodi.

Ogni core è responsabile di una porzione del grafo, determinata da **rank * graph.node_count / size e (rank + 1) * graph.node_count / size**.

Analogamente, la stessa procedura viene seguita per calcolare le forze attrattive tra i nodi collegati da un arco.

Una volta calcolate le forze, sincronizzo i core e invio i dati aggiornati al nodo radice,che si occuperà di aggiornare le posizioni dei nodi e inviare i dati aggiornati agli altri nodi.

Durante ogni iterazione, viene applicato un decadimento alla temperatura, utilizzato per ridurre l'intensità delle forze nel tempo.

Modelli di programmazione

L'interfaccia MPI (Message Passing Interface) rappresenta uno standard fondamentale nel campo del calcolo parallelo, fornendo strumenti avanzati per la sincronizzazione e la comunicazione tra processi all'interno di un sistema distribuito.
Questo protocollo è cruciale per consentire ai processi di cooperare efficacemente, scambiando dati in modo efficiente e gestendo complessi schemi di comunicazione.
La sua adozione è diffusa in ambiti scientifici e ingegneristici dove il calcolo parallelo è essenziale per affrontare problemi computazionali di dimensioni e complessità elevate.
OpenMP (Open Multi-Processing) è un'altra tecnologia fondamentale nel contesto del calcolo parallelo, ma differisce significativamente da MPI, mentre MPI si concentra sulla comunicazione tra processi in un ambiente distribuito, OpenMP è progettato per la programmazione parallela su sistemi con memoria condivisa, come i multi-core moderni.
OpenMP offre un modello di programmazione parallela basato su direttive pragma, che permettono agli sviluppatori di specificare in modo esplicito quali parti del codice devono essere eseguite parallelamente.

In questo caso, visto che usiamo un cluster di nodi, si è scelto di usare MPI.

Funzioni

```c
void calculateRepulsion(Force* net_forces, SimpleGraph* graph, size_t start, size_t end) {
    for (size_t i = start; i < end; i++) {
        for (size_t j = 0; j < graph->node_count; j++) {
            if (i != j) {
                double dx = graph->nodes[i].x - graph->nodes[j].x;
                double dy = graph->nodes[i].y - graph->nodes[j].y;
                double distance = sqrt(dx * dx + dy * dy);

                if (distance < MIN_DISTANCE) {
                    distance = MIN_DISTANCE;
                }

                double force_magnitude = k * k / distance;
                double force_x = force_magnitude * (dx / distance);
                double force_y = force_magnitude * (dy / distance);

                net_forces[i].x += force_x;
                net_forces[i].y += force_y;

            }
        }
    }
}

void calculateAttraction(Force* net_forces, SimpleGraph* graph, size_t start, size_t end) {
    for (size_t i = start; i < end; i++) {
        size_t node1 = graph->edges[i].start;
        size_t node2 = graph->edges[i].end;
        double dx = graph->nodes[node2].x - graph->nodes[node1].x;
        double dy = graph->nodes[node2].y - graph->nodes[node1].y;
        double distance = sqrt(dx * dx + dy * dy);
        double force = distance * distance / k;
        if (distance < MIN_DISTANCE) {
            distance = MIN_DISTANCE;
        }
        net_forces[node1].x += force * dx / distance;
        net_forces[node1].y += force * dy / distance;
        net_forces[node2].x -= force * dx / distance;
        net_forces[node2].y -= force * dy / distance;

    }
}
```

La funzione **calculateRepulsion** si occupa di calcolare le forze repulsive tra i nodi all'interno di un intervallo specificato.
Se i e j sono nodi diversi, viene calcolata la distanza euclidea tra i due nodi.
Una volta calcolata la forza, la decompongo nella componente orizzontale e verticale e aggiorno net_force

La funzione **calculateAttraction** si occupa di calcolare e applicare le forze attrattive tra i nodi collegati da archi.
La forza è proporzionale alla distanza, il che significa che nodi molto distanti tendono ad avvicinarsi rapidamente, mentre nodi già vicini hanno una forza attrattiva quasi nulla.
Anche qua decompongo la forza nelle componenti orizzontali e verticali

La funzione **moveNodes** scorre ogni singolo nodo, calcola lo spostamento del nodo lungo l'asse x tenendo conto della forza netta che agisce su di esso e limitando lo spostamento massimo a temperature.

```
void moveNodes(Force* net_forces, SimpleGraph* graph, size_t start, size_t end) {
    for (size_t i = start; i < end; i++) {
        double dx = net_forces[i].x;
        double dy = net_forces[i].y;


        double displacement = sqrt(dx * dx + dy * dy);
        if (displacement > temperature) {
            dx = dx / displacement * temperature;
            dy = dy / displacement * temperature;
        }

        graph->nodes[i].x += dx;
        graph->nodes[i].y += dy;
    }
}
```

```
void getMaxNodeDimensions(SimpleGraph* graph, double* maxX, double* maxY) {
    *maxX = 0.0;
    *maxY = 0.0;
    for (size_t i = 0; i < graph->node_count; i++) {
        if (fabs(graph->nodes[i].x) > *maxX) {
            *maxX = fabs(graph->nodes[i].x);
        }
        if (fabs(graph->nodes[i].y) > *maxY) {
            *maxY = fabs(graph->nodes[i].y);
        }
    }
}
```

La funzione **getMaxNodesDimensions** semplicemente torna le coordinate massime dei nodi, in valore assoluto, per x e y

La funzione **readGraphFile** prende in input il nome di un file e legge come prima cosa il numero di nodi presenti nel file.

Per gestire gli archi, viene creato un array temporaneo **temp_edges** con una capacità iniziale di 100 elementi. La funzione utilizza due variabili **temp_capacity** e **temp_edge_count** per gestire la capacità e il conteggio degli archi temporanei. Se la capacità dell'array temporaneo viene superata, la funzione raddoppia la capacità dell'array usando realloc.

Se il peso non è presente, la funzione legge due valori alla volta dal file, altrimenti tre valori alla volta.

Successivamente, la funzione alloca memoria per i nodi e inizializza le loro posizioni su un cerchio unitario.

Infatti le coordinate x e y di ciascun nodo vengono calcolate usando le funzioni cos e sin, distribuendo i nodi uniformemente attorno a un cerchio.

E infine copio l’array temporaneo nell'array del grafo con memcpy e imposto **graph.edge_count** al valore di **temp_edge_count** per riflettere il numero totale di archi nel grafo.

Aspetti teorici

Il modello di Fruchterman-Reingold utilizza una metafora fisica per posizionare i nodi di un grafo:

-I nodi si respingono reciprocamente come cariche elettriche.

-Gli archi agiscono come molle, attraendo i nodi collegati.

L'obiettivo è minimizzare l'energia complessiva del sistema, raggiungendo una configurazione stabile dove le forze sono bilanciate.

La scalabilità forte si riferisce alla capacità di un sistema parallelo di mantenere le prestazioni costanti quando si aumenta il numero di processori, mantenendo fisso il carico di lavoro totale.
Se aumento il numero di processori oltre un certo limite, le prestazioni peggiorano a causa dell'overhead tra i nodi, come la contesa delle risorse.

La scalabilità debole invece valuta come le prestazioni di un sistema parallelo variano quando aumenta il numero di processori, mentre la dimensione del problema cresce proporzionalmente.
Anche in questo caso, possiamo osservare che adattando la quantità di processori proporzionalmente alla dimensione del problema, i risultati rimangono costanti, a patto di rispettare i principi della scalabilità forte.
La qualità dei risultati viene valutata in termini di stabilità del grafo e della disposizione dei nodi.

Informazioni su ogni nodo

Versione GCC

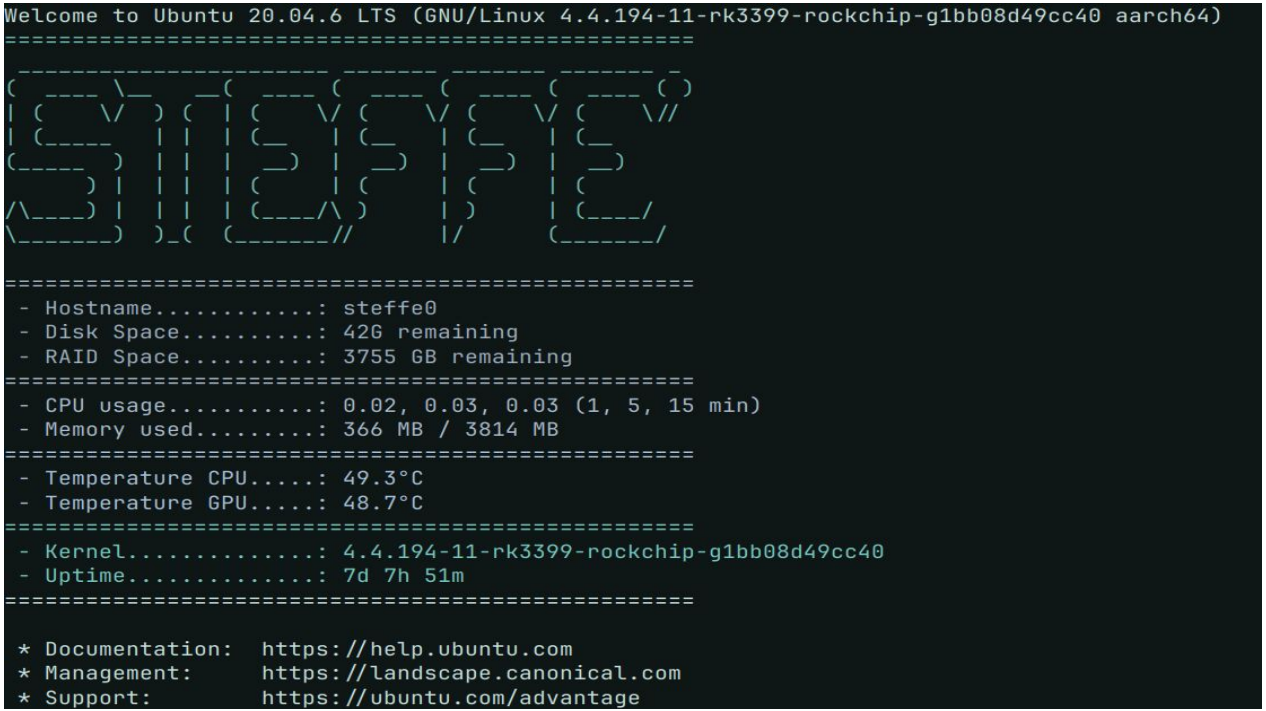
```
(Ubuntu
9.4.0-1ubuntu1~20.04.2) 9.4.0
```

Come compilare

```
mpicc -g -o main main.c
-lm per il file main.c

gcc -o disegna disegna.c
-lSDL2 -lm per il file
disegna.c

oppure semplicemente
usando il Makefile con:
make
```



Performance

Utilizzeremo il filo bio-dmela per testare le performance (7394 nodi).

```
GNU nano 4.8
#!/bin/bash
#SBATCH --job-name=mpi_test
#SBATCH --output=mpi_test_output.txt
#SBATCH --ntasks=9
#SBATCH --time=00:10:00
#SBATCH --ntasks-per-node=3
#SBATCH --cpus-per-task=2


module load openmpi
srun prova bio-dmela 100 500 0
```

-> 2.45 minuti

1.23 minuti <-

```
GNU nano 4.8
#!/bin/bash
#SBATCH --job-name=mpi_test
#SBATCH --output=mpi_test_output.txt
#SBATCH --ntasks=20
#SBATCH --time=00:10:00
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=3


module load openmpi
srun prova bio-dmela 100 500 0
```

```
#!/bin/bash
#SBATCH --job-name=mpi_test
#SBATCH --output=mpi_test_output.txt
#SBATCH --ntasks=54
#SBATCH --time=00:10:00
#SBATCH --ntasks-per-node=6
#SBATCH --cpus-per-task=1


module load openmpi
srun prova bio-dmela 100 500 0
```

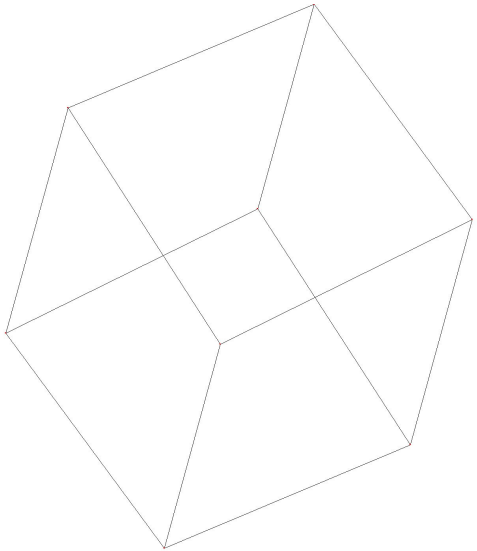
-> 46 secondi

Test

Questi test saranno eseguiti utilizzando diversi tipi di input, come temperatura, file e iterazioni. Lo scopo di questi test è verificare la robustezza, l'affidabilità e l'efficacia del programma in condizioni diverse.

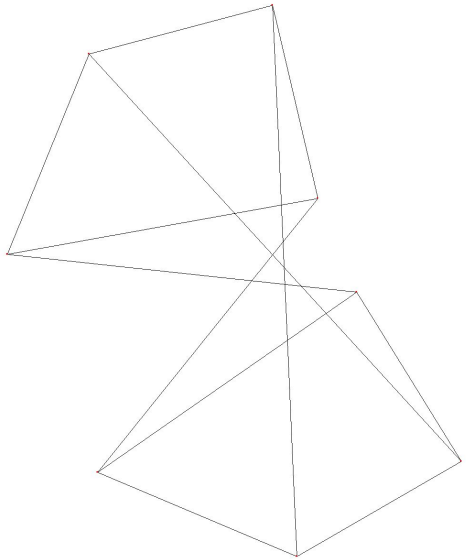
```
./main cube 100 500 0
```

Con il comando sopra (file, iterazioni, temperature e peso) otteniamo il seguente grafo(dopo aver fatto ./disegna out.txt) ->

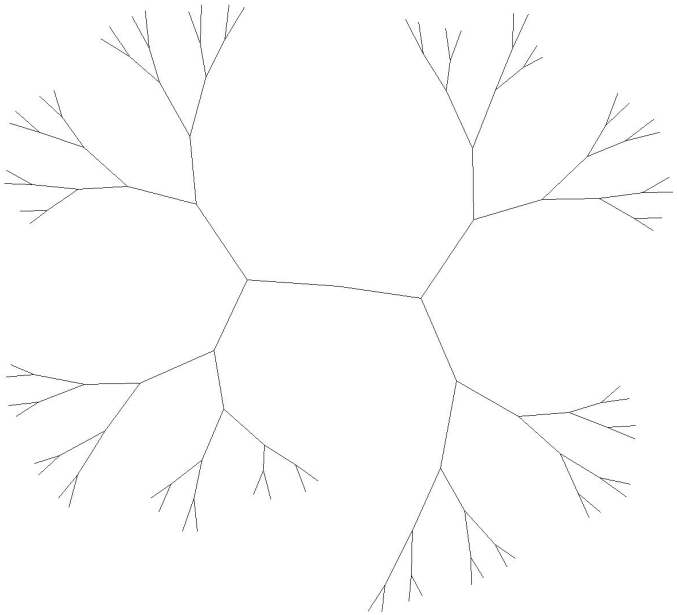


Ma si può vedere come modificando un po il comando, per esempio abbassando il numero di iterazioni, cambi il risultato

```
./main cube 60 500 0    |
                        V
```

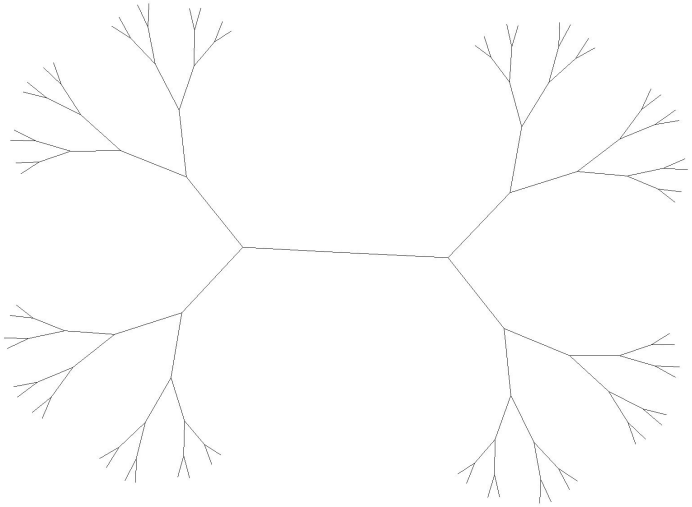


```
./main 127binary-tree 100 500 0
```

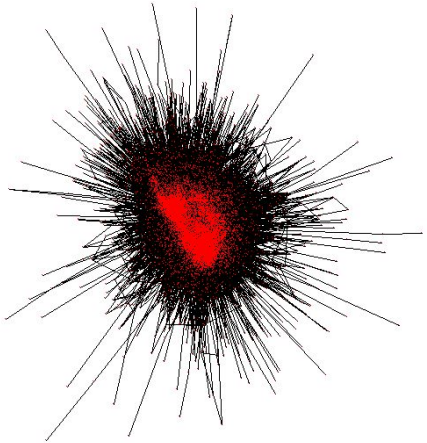


E se aumentiamo il numero di iterazioni, possiamo vedere come si stabilizza del tutto

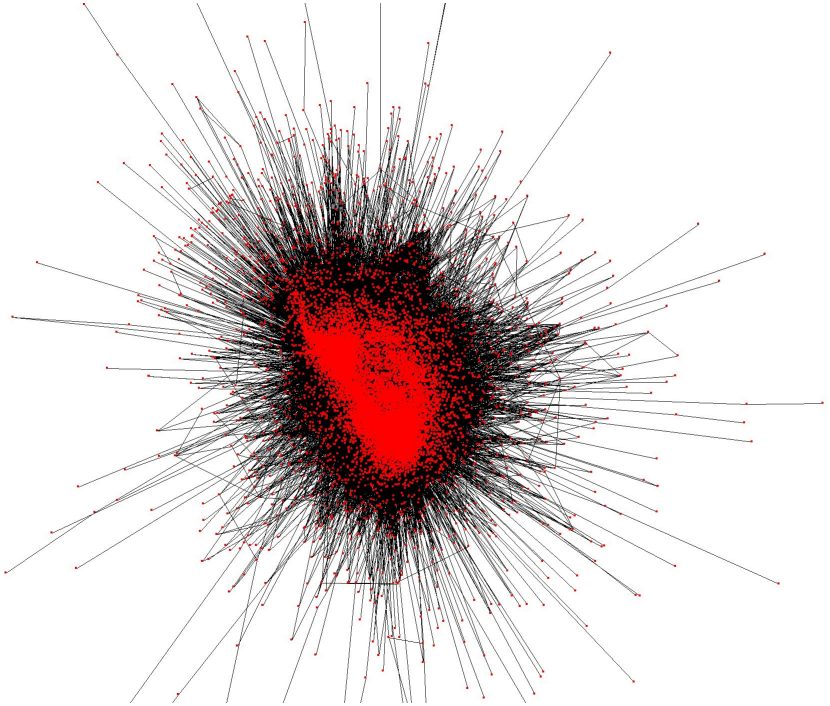
```
./main 127binary-tree 300 500 0
```

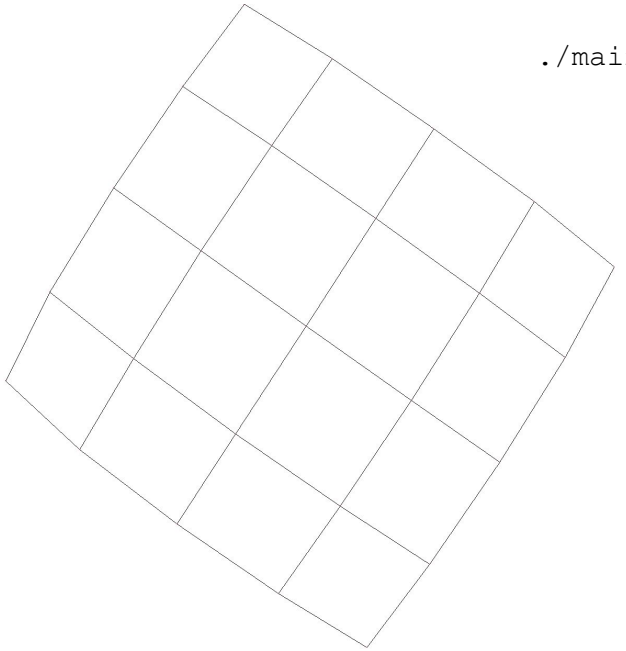


Se proviamo ad usare un file più grande `./main bio-human-gene1 100 500 1 (22284)`



Otteniamo qualcosa del genere, ovviamente andando ad effettuare lo zoom possiamo vedere meglio i singoli nodi (6.50 minuti)

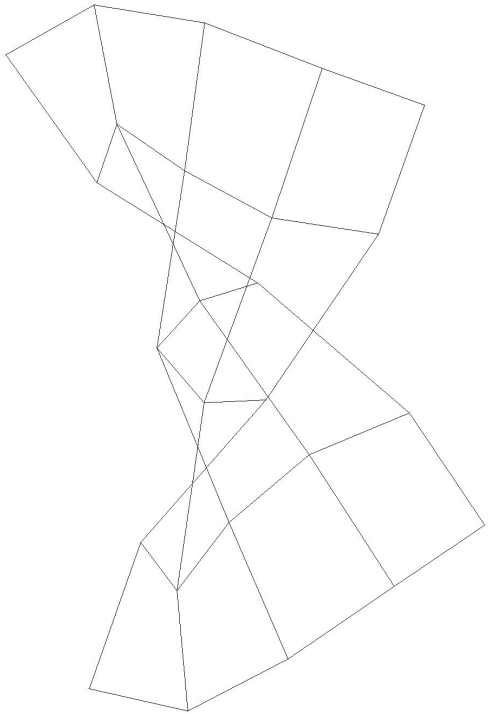




```
./main 5grid 300 500 0
```

Anche qua è possibile vedere come, al diminuire delle iterazioni il grafico non raggiunge lo stato finale

```
./main 5grid 100 500 0
```



Codice: https://github.com/Francesco7602/Laboratorio-Computazionale/

BIBLIOGRAFIA

THOMAS M. J. FRUCHTERMAN* AND EDWARD M. REINGOLD : Graph Drawing by Force-directed Placement

Christopher Mueller, Douglas Gregor, Andrew Lumsdaine : Distributed Force-Directed Graph Layout and Visualization