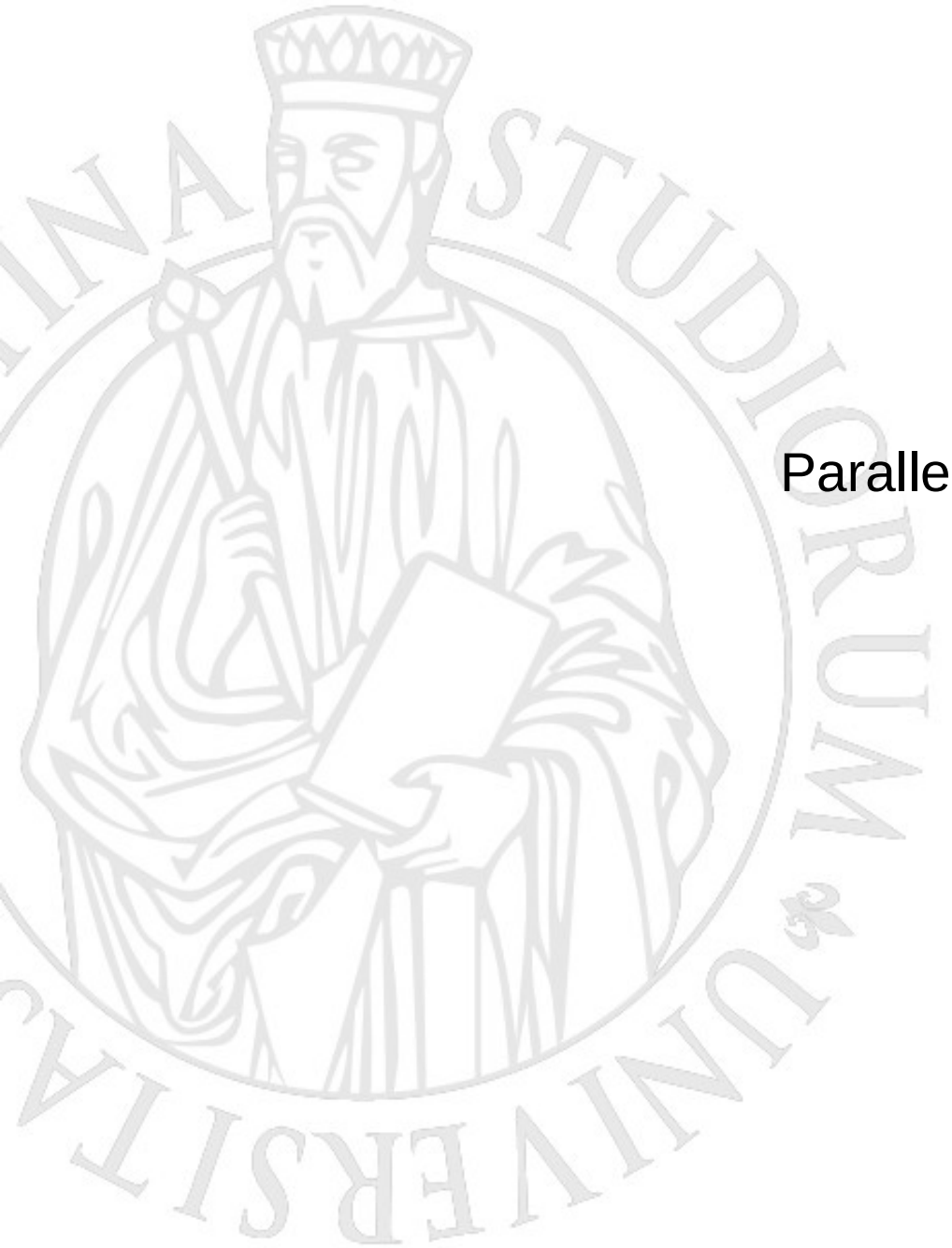


Random Maze Solver

Parallel Programming for Machine Learning

Sofia Galante

18/03/2021

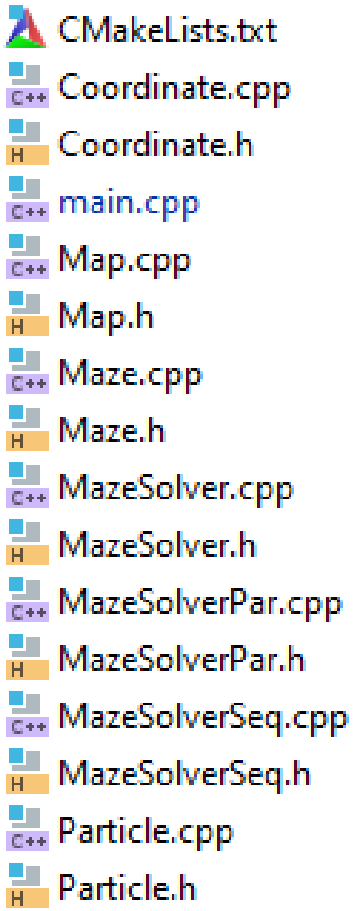
Introduzione

Il programma scritto per questo esperimento è un **Random Maze Solver**: un labirinto (generato in modo random) viene risolto da delle particelle che si muovono al suo interno, scegliendo il percorso da seguire randomicamente.
La particella più veloce (cioè che esce dal labirinto con il minor numero di passi) mostra il cammino corretto da seguire all'interno del labirinto.

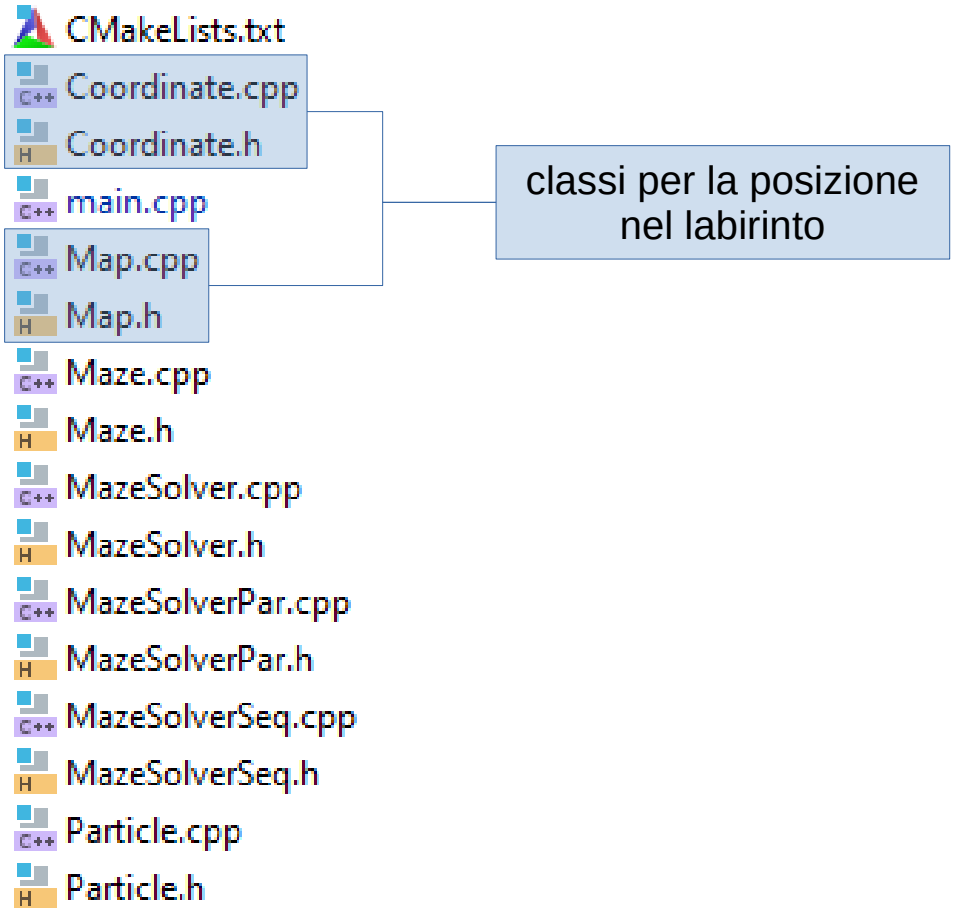
Si sono creati due diversi **Random Maze Solver**: uno di tipo *sequenziale* e uno di tipo *parallelo*.
Lo scopo di questo elaborato è quello di osservare lo **speedup** ottenuto nel secondo tipo di **Random Maze Solver** rispetto al primo.

Il linguaggio di programmazione utilizzato è il C++ (con compiler MinGW) e la parallelizzazione è stata svolta con **OpenMP**.
Gli esperimenti si sono svolti su un PC con Windows 10 come sistema operativo e una CPU Intel Core i5-11400.

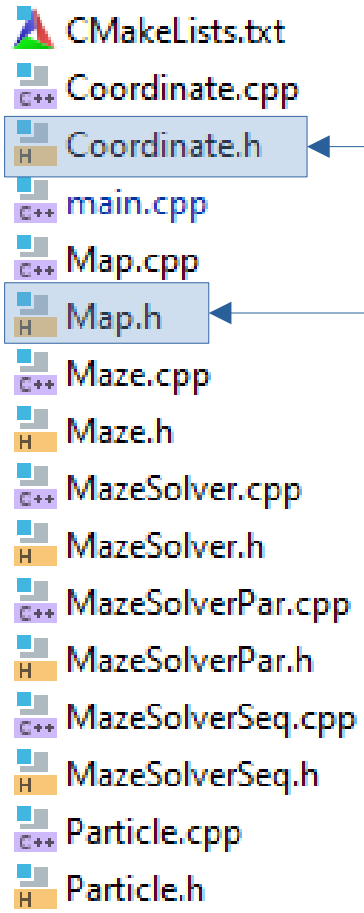
Organizzazione delle classi



Organizzazione delle classi



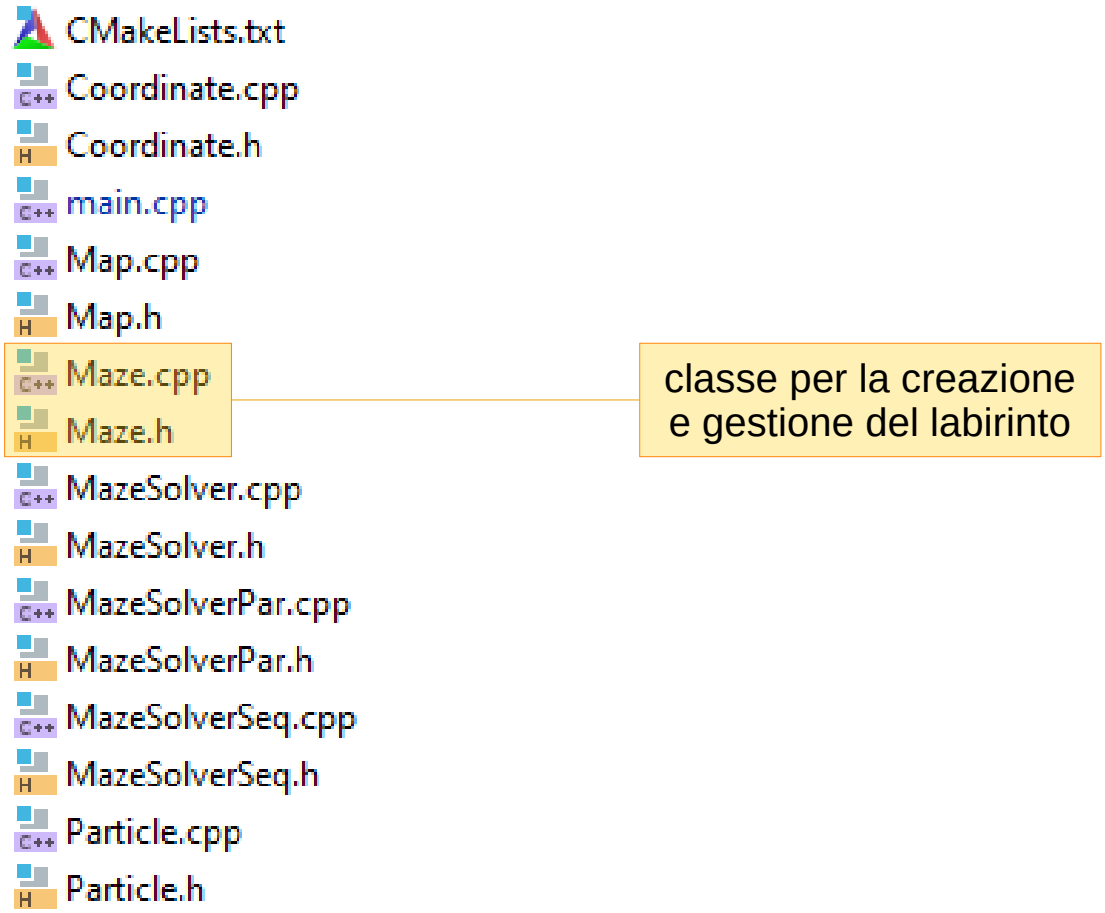
Organizzazione delle classi



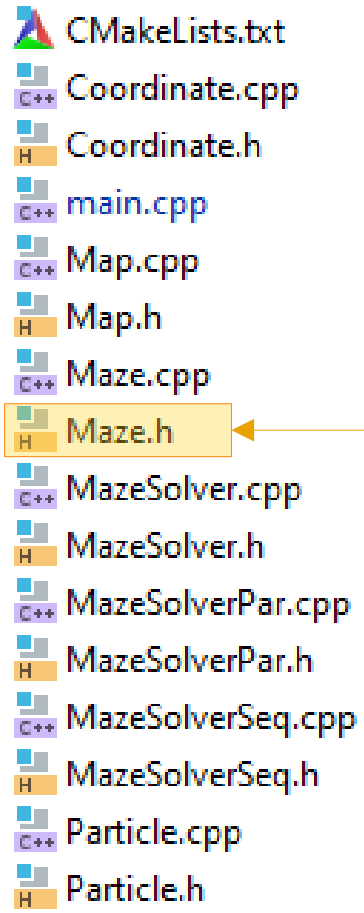

```cpp
class Coordinate{
public:
    explicit Coordinate(int x = 0, int y = 0);
    void setCoordinate(int x, int y);

    int getX() const{...}
    int getY() const{...}

    bool operator==(const Coordinate &right) const{...}
    bool operator!=(const Coordinate &right) const{...}

private:
    int x;
    int y;
};
```

```cpp
class Map {
public:
    explicit Map(int width, int height);

    int getValue(Coordinate p) const{...}

    void setValue(Coordinate p, int value){...}

    void incrValue(Coordinate p){...}


private:
    std::vector<int> map;
    int width;
    int height;
};
```

Organizzazione delle classi

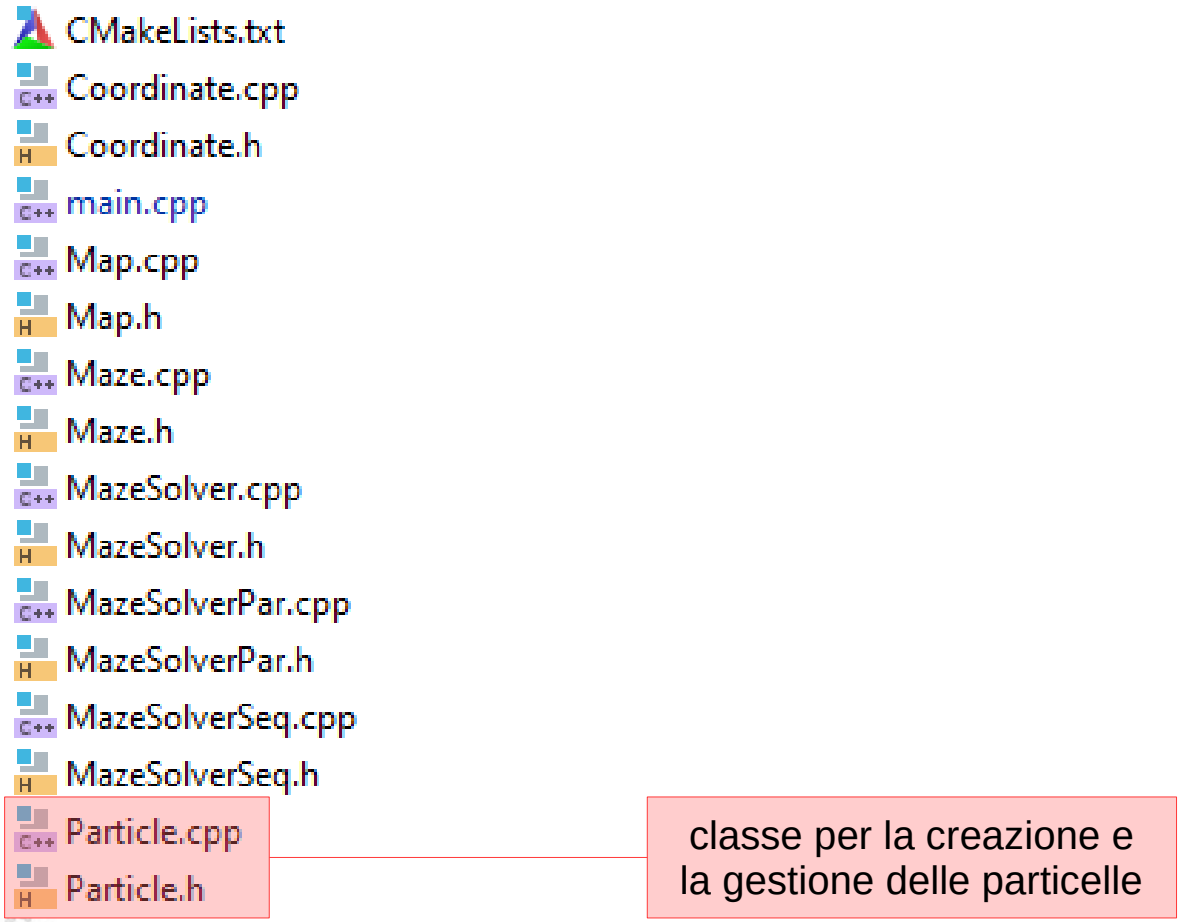


Organizzazione delle classi

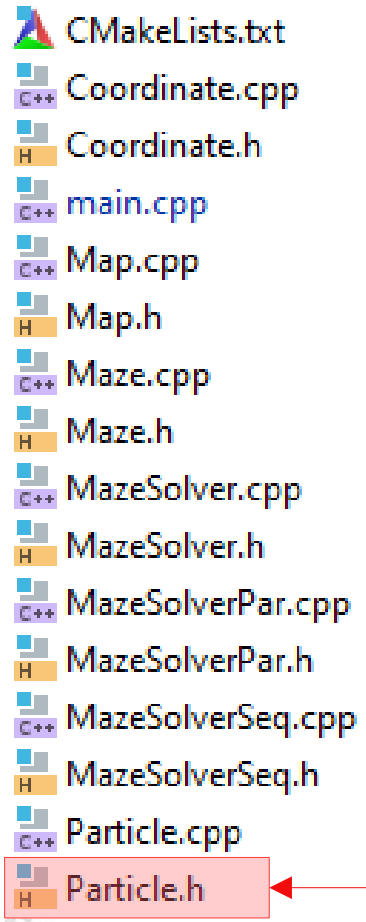



```
class Maze {
public:
    explicit Maze(int w, int h);

    Coordinate getStart() const{...}
    Coordinate getEnd() const{...}
    int getWidth() const{...}
    int getHeight() const{...}
    int getMaxSteps() const{...}
    Map getMap() const{...}

private:
    void createMaze();
    void setStartAndEnd();
    Coordinate setStartOrEnd(int wall);
    std::vector<Coordinate> validMoves(Coordinate now, bool inRecovery);
    bool isPointValid(Coordinate point, bool inRecovery);
    void placeWalls(Coordinate now);
    void placeWall(Coordinate p, int x, int y);
    Coordinate rewind(Coordinate now);

    int getDirection(Coordinate now, Coordinate prev);

    void recovery();
    std::vector<Coordinate> findWallsToRemove(Coordinate now);
    void print();

    int height;
    int width;
    Coordinate start;
    Coordinate end;
    int maxSteps;
    Map map; // -1 = wall, {1, 2, 3, 4} = {nord, est, sud, ovest}
};
```

Organizzazione delle classi

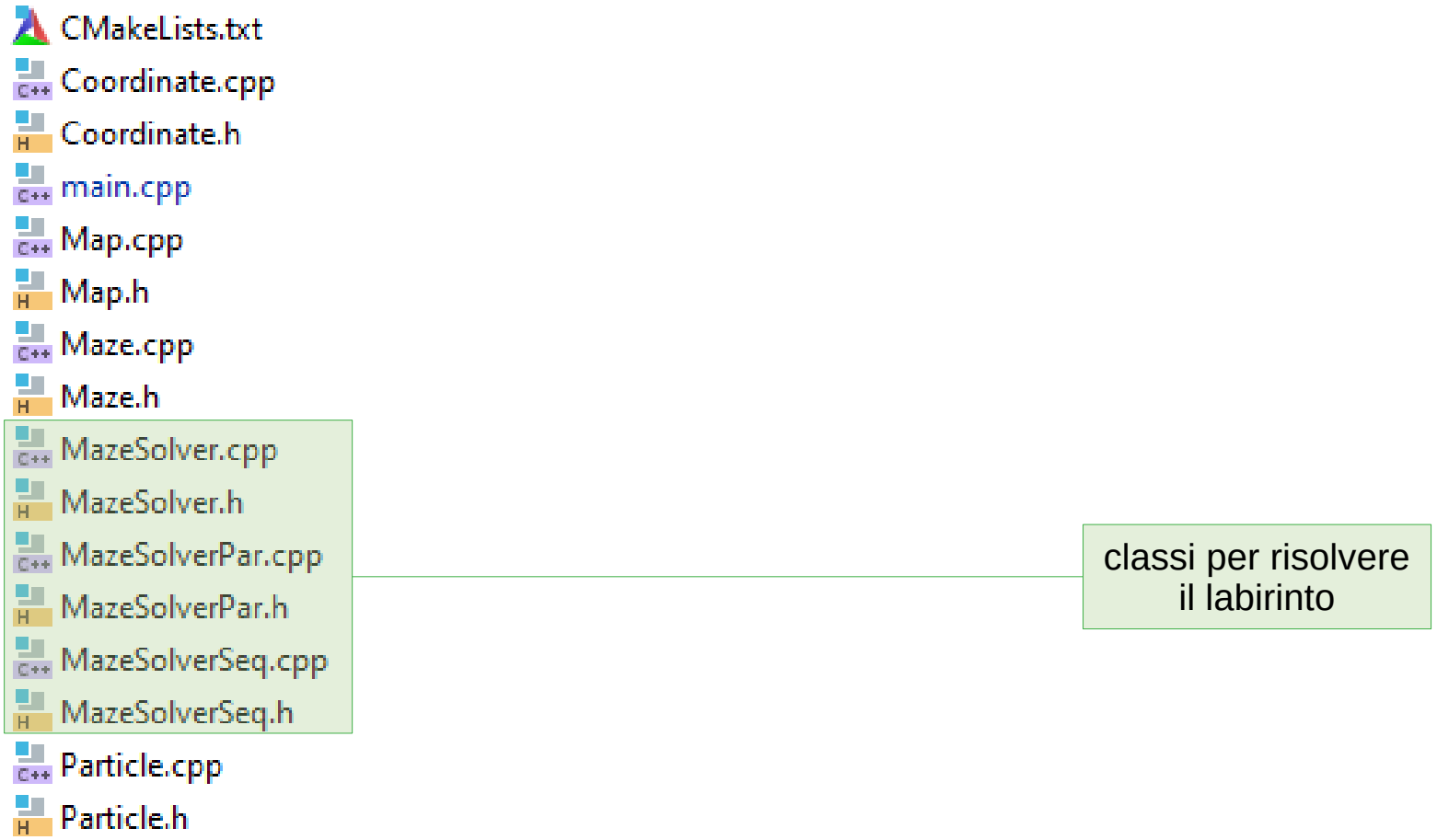


Organizzazione delle classi

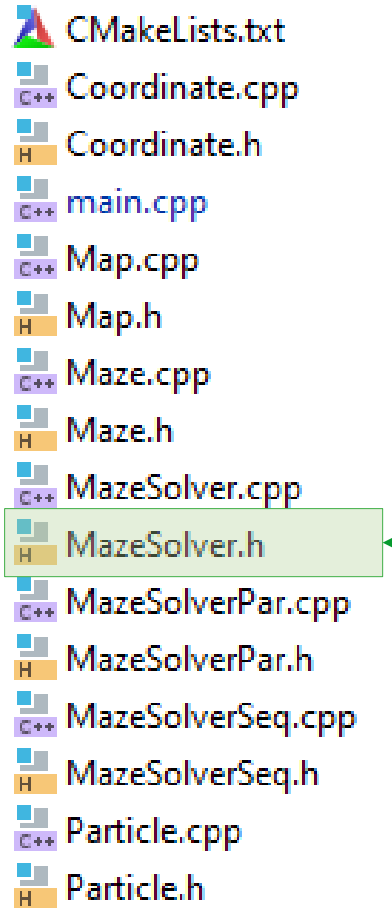



```
class Particle{
public:
    explicit Particle(Coordinate start, int id, Map map);

    void addStep(Coordinate point){...}
    Map getMap() const{...}
    int getSteps() const{...}
    int getID() const{...}

private:
    Map map;
    int steps;
    int ID;
};
```

Organizzazione delle classi

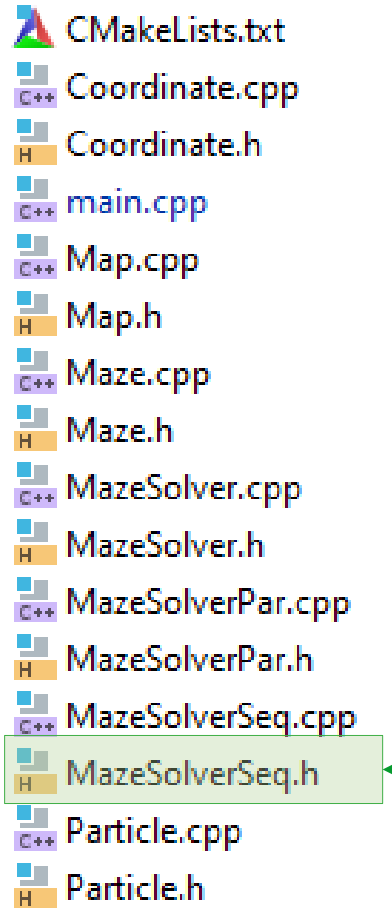


Organizzazione delle classi



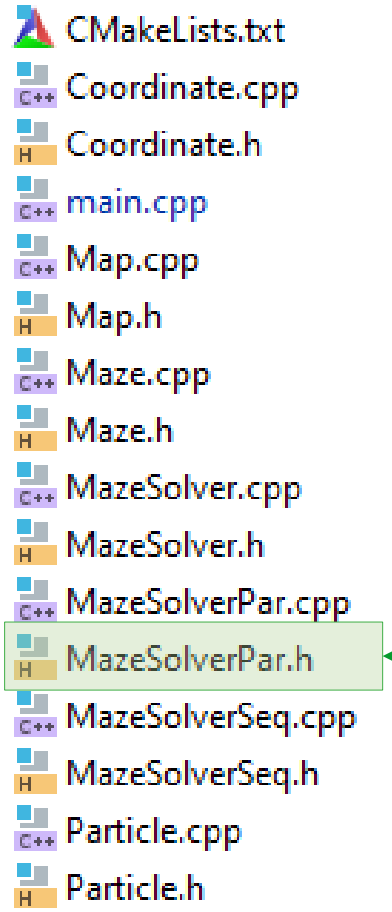

```
class MazeSolver{
public:
    explicit MazeSolver(Maze m) : maze(m){};
    virtual void solve(int numberOfParticles) = 0;
protected:
    bool moveParticle(Particle& p);
    virtual std::vector<Coordinate> validMoves(Coordinate now);
    bool isPointValid(Coordinate next);
    void print(Particle &p);
    Maze maze;
};
```

Organizzazione delle classi




```
class MazeSolverSeq : public MazeSolver {
public:
    explicit MazeSolverSeq(Maze m) : MazeSolver(m){};
    void solve(int numberOfParticles) override;

};
```

Organizzazione delle classi




```
class MazeSolverPar : public MazeSolver{
public:
    explicit MazeSolverPar(Maze m, int threads, bool lock) : MazeSolver(m), threads(threads), withLock(lock) {}
    void solve(int numberOfParticles) override;

private:
    std::vector<Coordinate> validMoves(Coordinate p) override;
    std::vector<Coordinate> validMovesLock(Coordinate p);

    int threads;
    bool withLock;
};
```

Algoritmo sequenziale

```
void MazeSolverSeq::solve(int numberOfParticles) {
    std::vector<Particle> particles;

    printf( format: "\n\nRisolvere il labirinto: modo sequenziale\n");
    //fa risolvere il labirinto a tutte le particelle
    for (int i = 0; i < numberOfParticles; i++) {
        printf( format: "Particella n %d entra nel labirinto\n", i+1);
        Map map( width: maze.getWidth(), height: maze.getHeight());
        Particle p( start: maze.getStart(), id: i+1, map);
        bool inTime = moveParticle(p);
        if(inTime){
            particles.push_back(p);
            printf( format: "Particella n %d uscita dal labirinto compiendo %d passi\n\n", i+1, p.getSteps());
        }
        else{
            printf( format: "La particella n %d non ha trovato l'uscita\n\n", i+1);
        }
    }

    if(particles.empty())
        printf( format: "Nessuna particella ha trovato l'uscita\n");
    else{
        //sceglie un vincitore
        auto winner : iterator<...> = particles.begin();
        for(auto particle : iterator<...> =  particles.begin()+1 ; particle != particles.end(); ++particle){
            if(particle->getSteps() < winner->getSteps())
                winner = particle;
        }
        //disegna il cammino del vincitore
        print( &: *winner);
    }
}
```

Algoritmo parallelo

Risoluzione del labirinto da parte delle particelle

```
void MazeSolverPar::solve(int numberOfParticles) {
    std::vector<Particle> particles;

    printf( format: "\n\nRisolvere il labirinto: modo parallelo\n");

    omp_set_nested(3);
    std::vector<int> seeds;

    //genera i seeds per la rand()
    for(int i = 0; i < numberOfParticles; i++){
        seeds.push_back(rand());
    }

#pragma omp parallel default(none) shared(particles) firstprivate(numberOfParticles, seeds, maze) num_threads(threads)
    {
        std::vector<Particle> localParticles;
#pragma omp for nowait
        for (int i = 0; i < numberOfParticles; i++) {
            srand( Seed: seeds[i]);
            printf( format: "THREAD %d -> Particella n %d entra nel labirinto\n", omp_get_thread_num(), i + 1);
            Map map( width: maze.getWidth(), height: maze.getHeight());
            Particle p( start: maze.getStart(), id: i + 1, map);
            bool inTime = moveParticle(p);
            if (inTime) {
                localParticles.push_back(p);
                printf( format: "THREAD %d -> Particella n %d uscita dal labirinto compiendo %d passi\n\n", omp_get_thread_num(),
                        i + 1, p.getSteps());
            } else {
                printf( format: "THREAD %d -> La particella n %d non ha trovato l'uscita\n\n", omp_get_thread_num(), i + 1);
            }
        }
#pragma omp critical
        particles.insert( position: particles.end(), first: localParticles.begin(), last: localParticles.end());
    }
```

Algoritmo parallelo

Risoluzione del labirinto da parte delle particelle

```
void MazeSolverPar::solve(int numberOfParticles) {
    std::vector<Particle> particles;

    printf( format: "\n\nRisolvere il labirinto: modo parallelo\n");

    omp_set_nested(3);
    std::vector<int> seeds;

    //genera i seeds per la rand()
    for(int i = 0; i < numberOfParticles; i++){
        seeds.push_back(rand());
    }

#pragma omp parallel default(none) shared(particles) firstprivate(numberOfParticles, seeds, maze) num_threads(threads)
    {
        std::vector<Particle> localParticles;
#pragma omp for nowait
        for (int i = 0; i < numberOfParticles; i++) {
            srand( Seed: seeds[i]);
            printf( format: "THREAD %d -> Particella n %d entra nel labirinto\n", omp_get_thread_num(), i + 1);
            Map map( width: maze.getWidth(), height: maze.getHeight());
            Particle p( start: maze.getStart(), id: i + 1, map);
            bool inTime = moveParticle(p);
            if (inTime) {
                localParticles.push_back(p);
                printf( format: "THREAD %d -> Particella n %d uscita dal labirinto compiendo %d passi\n\n", omp_get_thread_num(),
                        i + 1, p.getSteps());
            } else {
                printf( format: "THREAD %d -> La particella n %d non ha trovato l'uscita\n\n", omp_get_thread_num(), i + 1);
            }
        }
#pragma omp critical
        particles.insert( position: particles.end(), first: localParticles.begin(), last: localParticles.end());
    }
```

generazione dei seeds

Algoritmo parallelo

Risoluzione del labirinto da parte delle particelle

```
void MazeSolverPar::solve(int numberOfParticles) {
    std::vector<Particle> particles;

    printf( format: "\n\nRisolvere il labirinto: modo parallelo\n");

    omp_set_nested(3);
    std::vector<int> seeds;

    //genera i seeds per la rand()
    for(int i = 0; i < numberOfParticles; i++){
        seeds.push_back(rand());
    }

#pragma omp parallel default(none) shared(particles) firstprivate(numberOfParticles, seeds, maze) num_threads(threads)
    {
        std::vector<Particle> localParticles;
#pragma omp for nowait
        for (int i = 0; i < numberOfParticles; i++) {
            srand( Seed: seeds[i]);
            printf( format: "THREAD %d -> Particella n %d entra nel labirinto\n", omp_get_thread_num(), i + 1);
            Map map( width: maze.getWidth(), height: maze.getHeight());
            Particle p( start: maze.getStart(), id: i + 1, map);
            bool inTime = moveParticle(p);
            if (inTime) {
                localParticles.push_back(p);
                printf( format: "THREAD %d -> Particella n %d uscita dal labirinto compiendo %d passi\n\n", omp_get_thread_num(),
                        i + 1, p.getSteps());
            } else {
                printf( format: "THREAD %d -> La particella n %d non ha trovato l'uscita\n\n", omp_get_thread_num(), i + 1);
            }
        }
#pragma omp critical
        particles.insert( position: particles.end(), first: localParticles.begin(), last: localParticles.end());
    }
```

parallelizzazione

Algoritmo parallelo

Risoluzione del labirinto da parte delle particelle

```
void MazeSolverPar::solve(int numberOfParticles) {
    std::vector<Particle> particles;

    printf( format: "\n\nRisolvere il labirinto: modo parallelo\n");

    omp_set_nested(3);
    std::vector<int> seeds;

    //genera i seeds per la rand()
    for(int i = 0; i < numberOfParticles; i++){
        seeds.push_back(rand());
    }

#pragma omp parallel default(none) shared(particles) firstprivate(numberOfParticles, seeds, maze) num_threads(threads)
    {
        std::vector<Particle> localParticles;
#pragma omp for nowait
        for (int i = 0; i < numberOfParticles; i++) {
            srand( Seed: seeds[i]);
            printf( format: "THREAD %d -> Particella n %d entra nel labirinto\n", omp_get_thread_num(), i + 1);
            Map map( width: maze.getWidth(), height: maze.getHeight());
            Particle p( start: maze.getStart(), id: i + 1, map);
            bool inTime = moveParticle(p);
            if (inTime) {
                localParticles.push_back(p);
                printf( format: "THREAD %d -> Particella n %d uscita dal labirinto compiendo %d passi\n\n", omp_get_thread_num(),
                        i + 1, p.getSteps());
            } else {
                printf( format: "THREAD %d -> La particella n %d non ha trovato l'uscita\n\n", omp_get_thread_num(), i + 1);
            }
        }
#pragma omp critical
        particles.insert( position: particles.end(), first: localParticles.begin(), last: localParticles.end());
    }
```

gestione della sezione critica

Algoritmo parallelo

Scelta del vincitore

```
    if(particles.empty())
        printf( format: "Nessuna particella ha trovato l'uscita\n");
    else{
        //sceglie un vincitore
        Particle winner = *particles.begin();
#pragma omp parallel for default(none) firstprivate(particles) shared(winner) \
        num_threads(threads)
        for(int i = 0; i < particles.size(); i++){
#pragma omp flush(winner)
            if(particles[i].getSteps() < winner.getSteps()) {
#pragma omp critical
                winner = particles[i];
#pragma omp flush(winner)
            }
        }
        //disegna il cammino del vincitore
        print( &: winner);
    }
}
```

Algoritmo parallelo

Scelta del vincitore

parallel for

```
    if(particles.empty())
        printf( format: "Nessuna particella ha trovato l'uscita\n");
    else{
        //sceglie un vincitore
        Particle winner = *particles.begin();
#pragma omp parallel for default(none) firstprivate(particles) shared(winner) \
        num_threads(threads)
        for(int i = 0; i < particles.size(); i++){
#pragma omp flush(winner)
            if(particles[i].getSteps() < winner.getSteps()) {
#pragma omp critical
                winner = particles[i];
#pragma omp flush(winner)
            }
        }
        //disegna il cammino del vincitore
        print( &: winner);
    }
}
```

Algoritmo parallelo

Scelta del vincitore

```
    if(particles.empty())
        printf( format: "Nessuna particella ha trovato l'uscita\n");
    else{
        //sceglie un vincitore
        Particle winner = *particles.begin();
#pragma omp parallel for default(none) firstprivate(particles) shared(winner) \
        num_threads(threads)
        for(int i = 0; i < particles.size(); i++){
#pragma omp flush(winner)
            if(particles[i].getSteps() < winner.getSteps()) {
#pragma omp critical
                winner = particles[i];
#pragma omp flush(winner)
            }
        }
        //disegna il cammino del vincitore
        print( &: winner);
    }
}
```

gestione della sezione critica

Algoritmo parallelo

Inserimento del lock

```
std::vector<Coordinate> MazeSolver::validMoves(Coordinate now) {
    std::vector<Coordinate> moves;
    std::vector<Coordinate> rightMoves;
    moves.emplace_back( x: now.getX(),  y: now.getY()+1);
    moves.emplace_back( x: now.getX()+1,  y: now.getY());
    moves.emplace_back( x: now.getX(),  y: now.getY()-1);
    moves.emplace_back( x: now.getX()-1,  y: now.getY());

    for(auto it : iterator<...>  = moves.begin(); it != moves.end(); ++it)
        if(isPointValid( next: *it))
            rightMoves.push_back(*it);

    return rightMoves;
}
```

```
std::vector<Coordinate> MazeSolverPar::validMovesLock(Coordinate now) {
    std::vector<Coordinate> moves;
    std::vector<Coordinate> rightMoves;
    moves.emplace_back( x: now.getX(),  y: now.getY()+1);
    moves.emplace_back( x: now.getX()+1,  y: now.getY());
    moves.emplace_back( x: now.getX(),  y: now.getY()-1);
    moves.emplace_back( x: now.getX()-1,  y: now.getY());

    omp_lock_t lock;
    omp_init_lock(&lock);

#pragma omp parallel for default(none) firstprivate(moves, maze) shared(rightMoves, lock) num_threads(4)
    for(int i = 0; i < 4; i++){
        if(isPointValid( next: moves[i])){
            omp_set_lock(&lock);
            rightMoves.push_back(moves[i]);
            omp_unset_lock(&lock);
        }
    }
    omp_destroy_lock(&lock);

    return rightMoves;
}
```

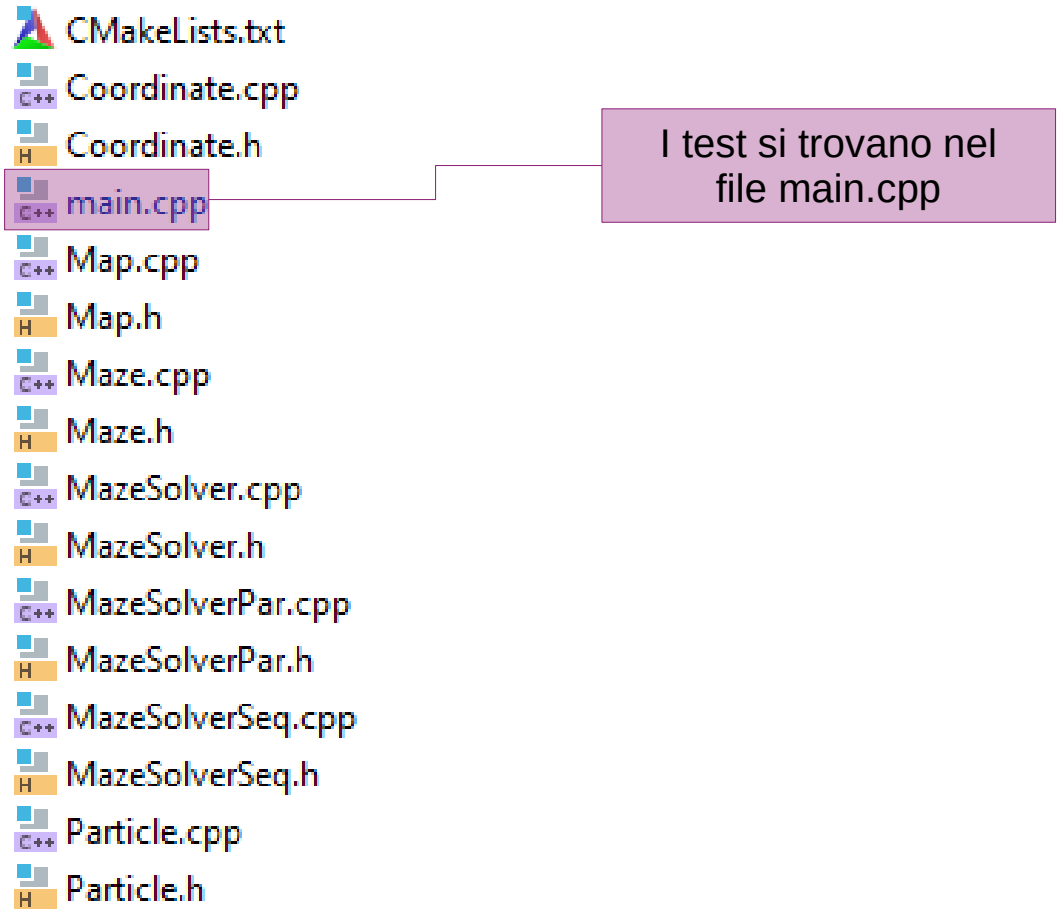
Algoritmo parallelo

Inserimento del lock

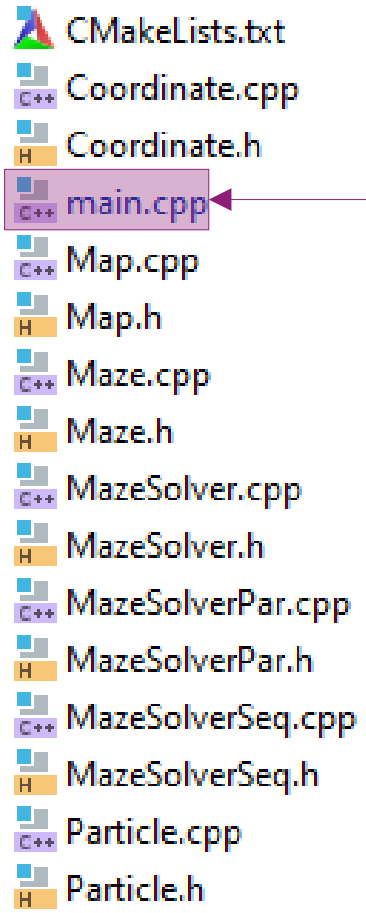
```
std::vector<Coordinate> MazeSolver::validMoves(Coordinate now) {
    std::vector<Coordinate> moves;
    std::vector<Coordinate> rightMoves;
    moves.emplace_back( x: now.getX(),  y: now.getY()+1);
    moves.emplace_back( x: now.getX()+1,  y: now.getY());
    moves.emplace_back( x: now.getX(),  y: now.getY()-1);
    moves.emplace_back( x: now.getX()-1,  y: now.getY());

    for(auto it : iterator<...>  = moves.begin(); it != moves.end(); ++it)
        if(isPointValid( next: *it))
            rightMoves.push_back(*it);

    return rightMoves;
}
```

```
std::vector<Coordinate> MazeSolverPar::validMovesLock(Coordinate now) {
    std::vector<Coordinate> moves;
    std::vector<Coordinate> rightMoves;
    moves.emplace_back( x: now.getX(),  y: now.getY()+1);
    moves.emplace_back( x: now.getX()+1,  y: now.getY());
    moves.emplace_back( x: now.getX(),  y: now.getY()-1);
    moves.emplace_back( x: now.getX()-1,  y: now.getY());

    omp_lock_t lock;
    omp_init_lock(&lock);

#pragma omp parallel for default(none) firstprivate(moves, maze) shared(rightMoves, lock) num_threads(4)
    for(int i = 0; i < 4; i++){
        if(isPointValid( next: moves[i])){
            omp_set_lock(&lock);
            rightMoves.push_back(moves[i]);
            omp_unset_lock(&lock);
        }
    }
    omp_destroy_lock(&lock);

    return rightMoves;
}
```

Test


CMakeLists.txt
Coordinate.cpp
Coordinate.h
main.cpp
Map.cpp
Map.h
Maze.cpp
Maze.h
MazeSolver.cpp
MazeSolver.h
MazeSolverPar.cpp
MazeSolverPar.h
MazeSolverSeq.cpp
MazeSolverSeq.h
Particle.cpp
Particle.h
I test si trovano nel file main.cpp

Test

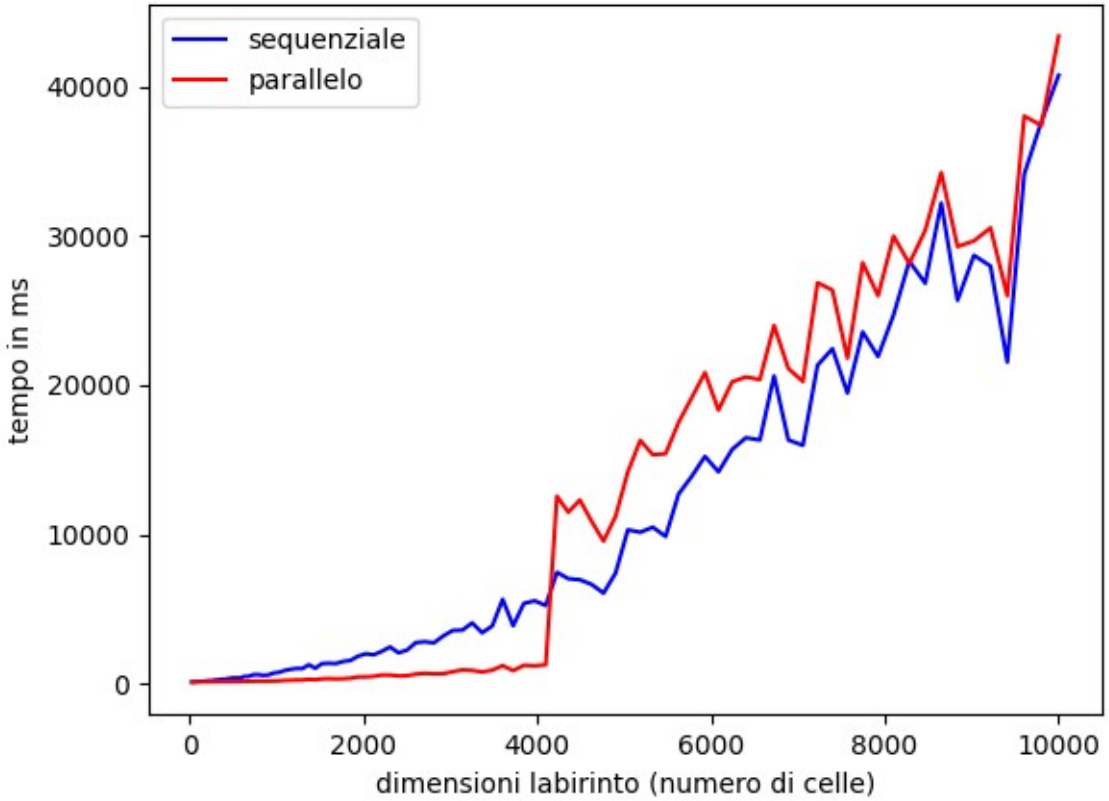



```
void dimTest(std::string testName){...}
void particlesTest(std::string testName){...}
void threadsTest(std::string testName){...}
void threadsTestV2(std::string testName){...}
void lockTest(std::string testName) {...}

int main(int argc, char *argv[]) {
    if(argc != 2){
        printf( format: "Manca il parametro\n");
        return 1;
    }
    srand( Seed: time( Time: NULL));

    std::string testName = argv[1];
    dimTest(testName);
    particlesTest(testName);
    threadsTest(testName);
    threadsTestV2(testName);
    lockTest(testName);
    return 0;
}
```

Test 1

Aumento della dimensione del labirinto

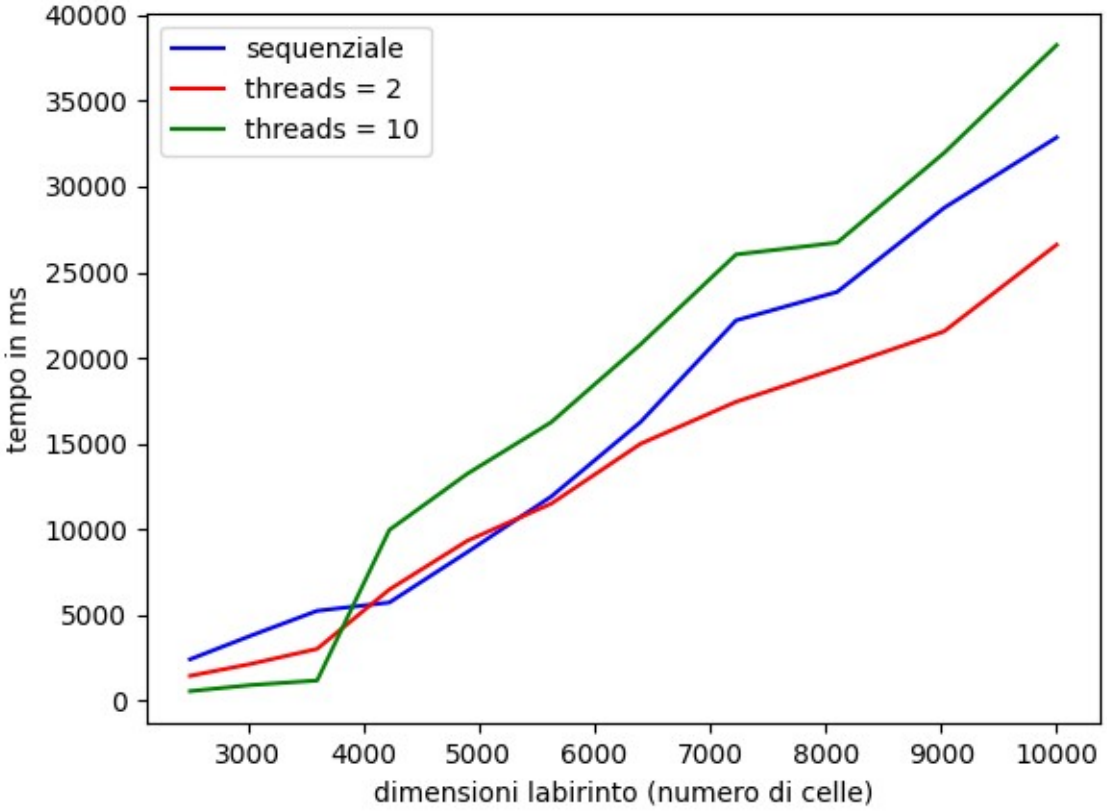
- numero di particelle = 50
- numero di thread per la versione parallela = 10
- dimensione del labirinto che varia da 5x5 fino a 100x100



Test 1 v2

Aumento della dimensione del labirinto (diversi numeri di thread)

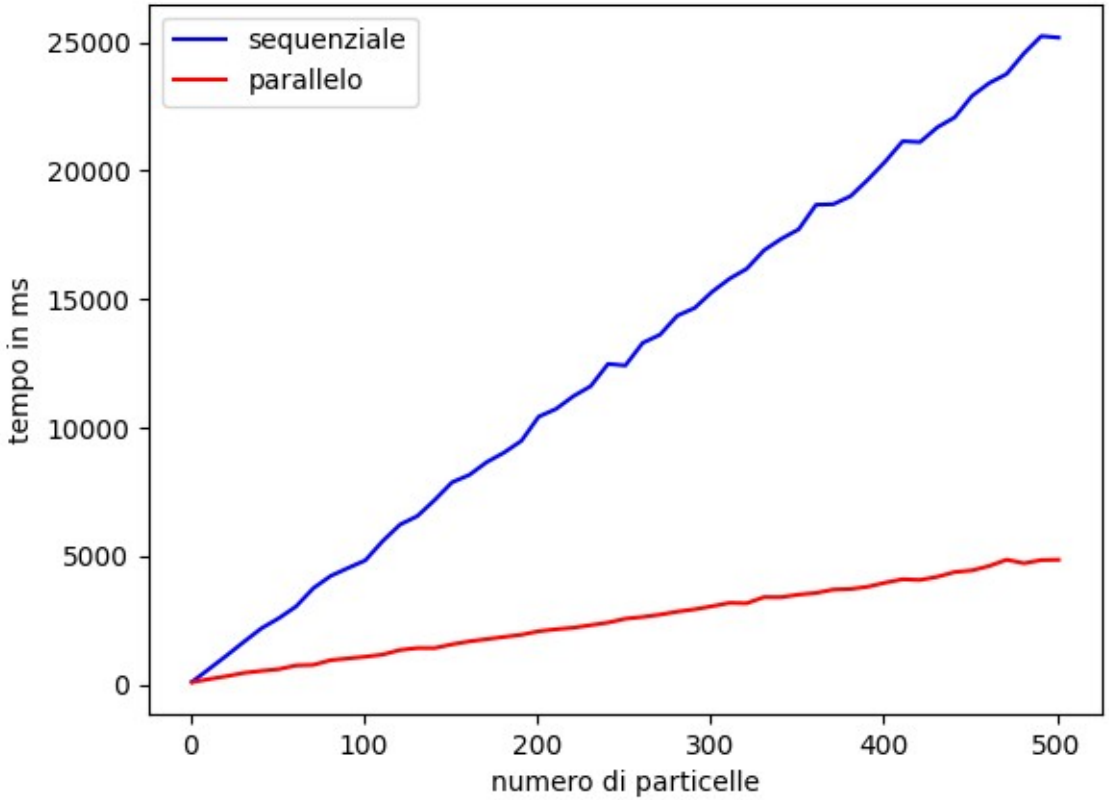
- numero di particelle = 50
- numero di thread per la versione parallela = {2, 10}
- dimensione del labirinto che varia da 50x50 fino a 100x100 (aumentando il lato di 5 in 5)



Test 2

Aumento del numero di particelle

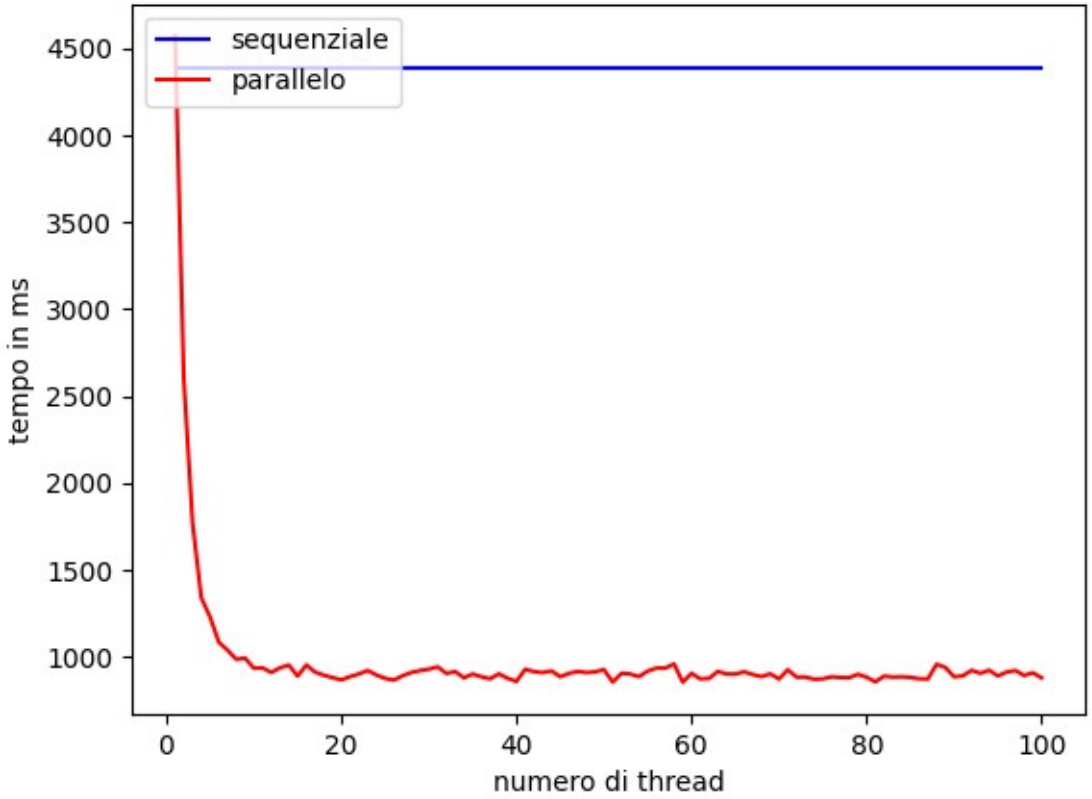
- numero di particelle che varia da 1 a 501 (aumenta di 10 in 10)
- numero di thread per la versione parallela = 10
- dimensione del labirinto = 50 x 50



Test 3

Aumento del numero di thread

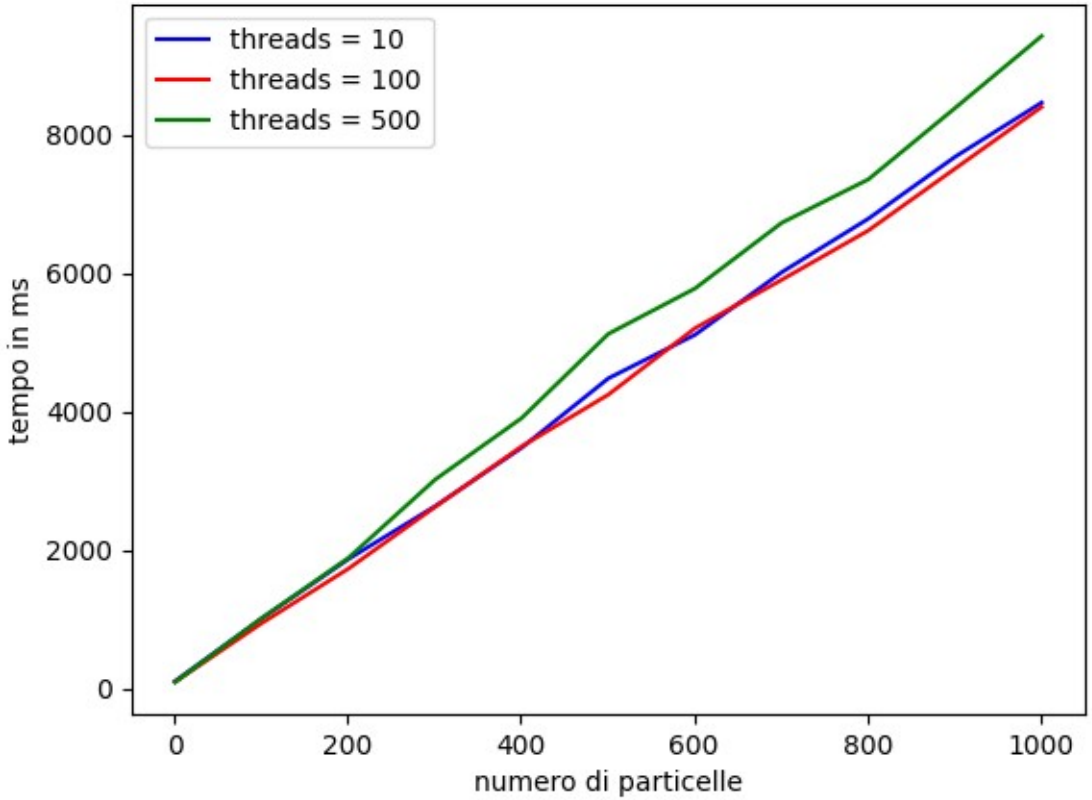
- numero di particelle = 100
- numero di thread per la versione parallela che varia da 1 a 100
- dimensione del labirinto = 50 x 50



Test 3 v2

Aumento del numero delle particelle (diverso numero di thread)

- numero di particelle che varia da 1 a 1001 (aumenta di 100 in 100)
- numero di thread per la versione parallela = {10, 100, 500}
- dimensione del labirinto = 50 x 50



Test 4

Inserimento del lock

- numero di particelle = 10
- numero di thread per la versione parallela = 10
- dimensione del labirinto = 20 x 20

Sequenziale	Parallelo senza lock	Parallelo con lock
82.478 ms	44.832 ms	2306.714 ms