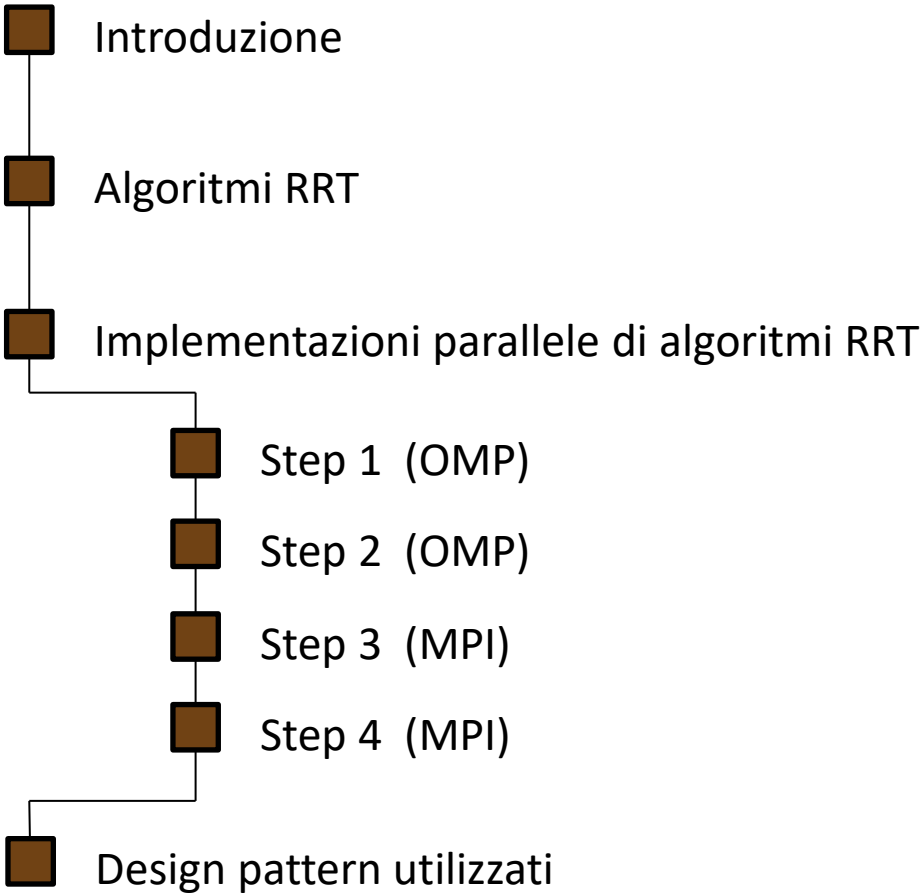
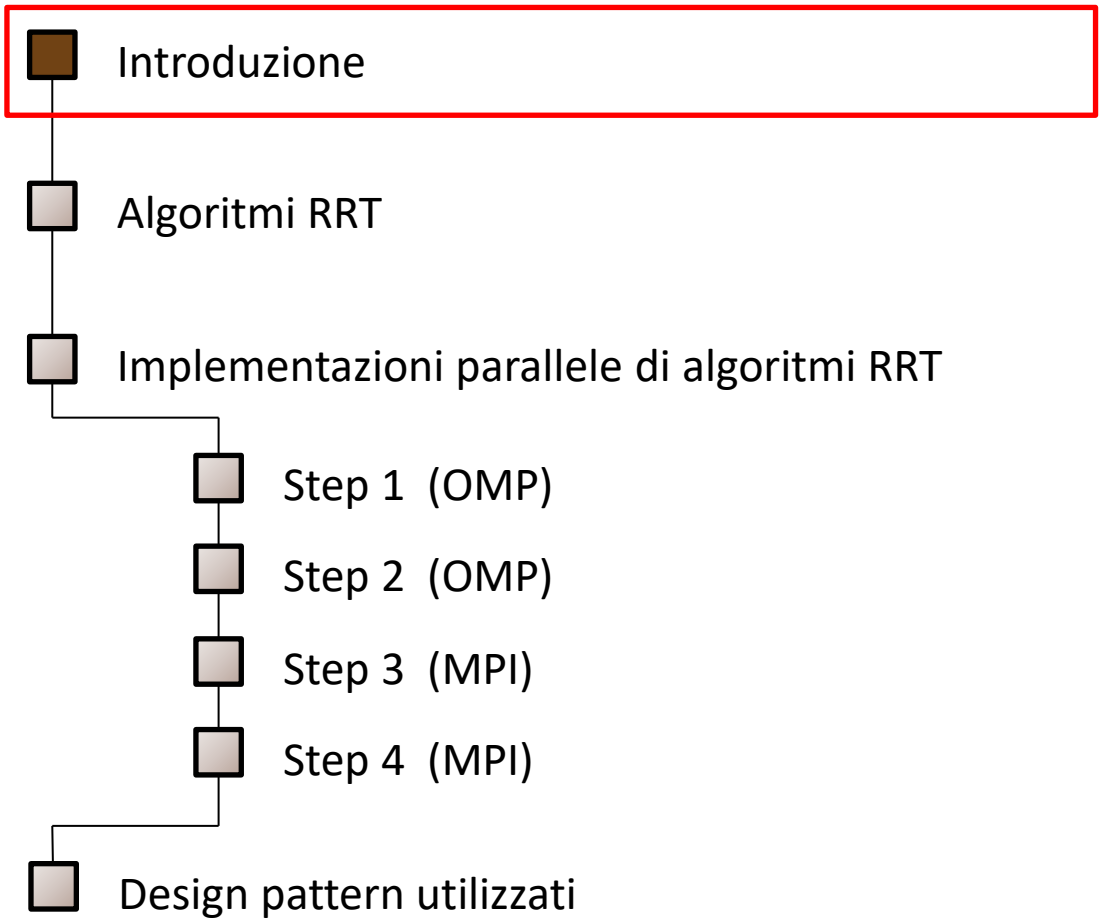
Andrea Casalino

Rapidly Random Tree (RRT) per Path Planning di manipolatori robotici, implementazioni parallele

Sommario

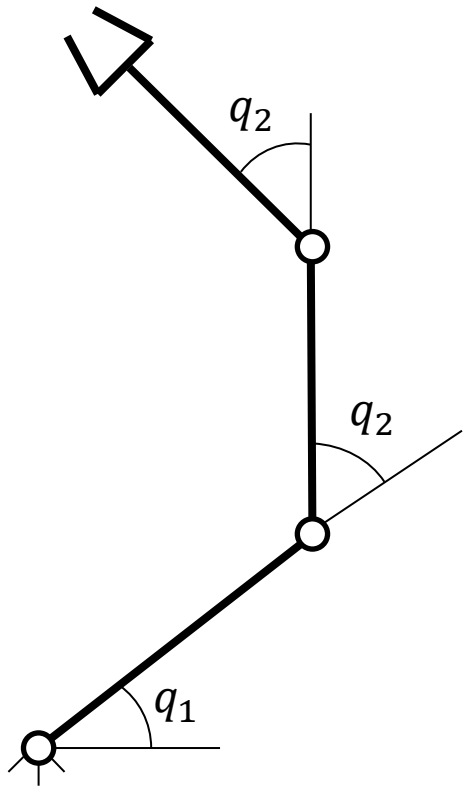

Introduzione
Algoritmi RRT
Implementazioni parallele di algoritmi RRT
Step 1 (OMP)
Step 2 (OMP)
Step 3 (MPI)
Step 4 (MPI)
Design pattern utilizzati

Sommario


Introduzione
Algoritmi RRT
Implementazioni parallele di algoritmi RRT
Step 1 (OMP)
Step 2 (OMP)
Step 3 (MPI)
Step 4 (MPI)
Design pattern utilizzati

Rapidly Random Tree (RRT) per Path Planning di manipolatori robotici

Manipolatore robotico

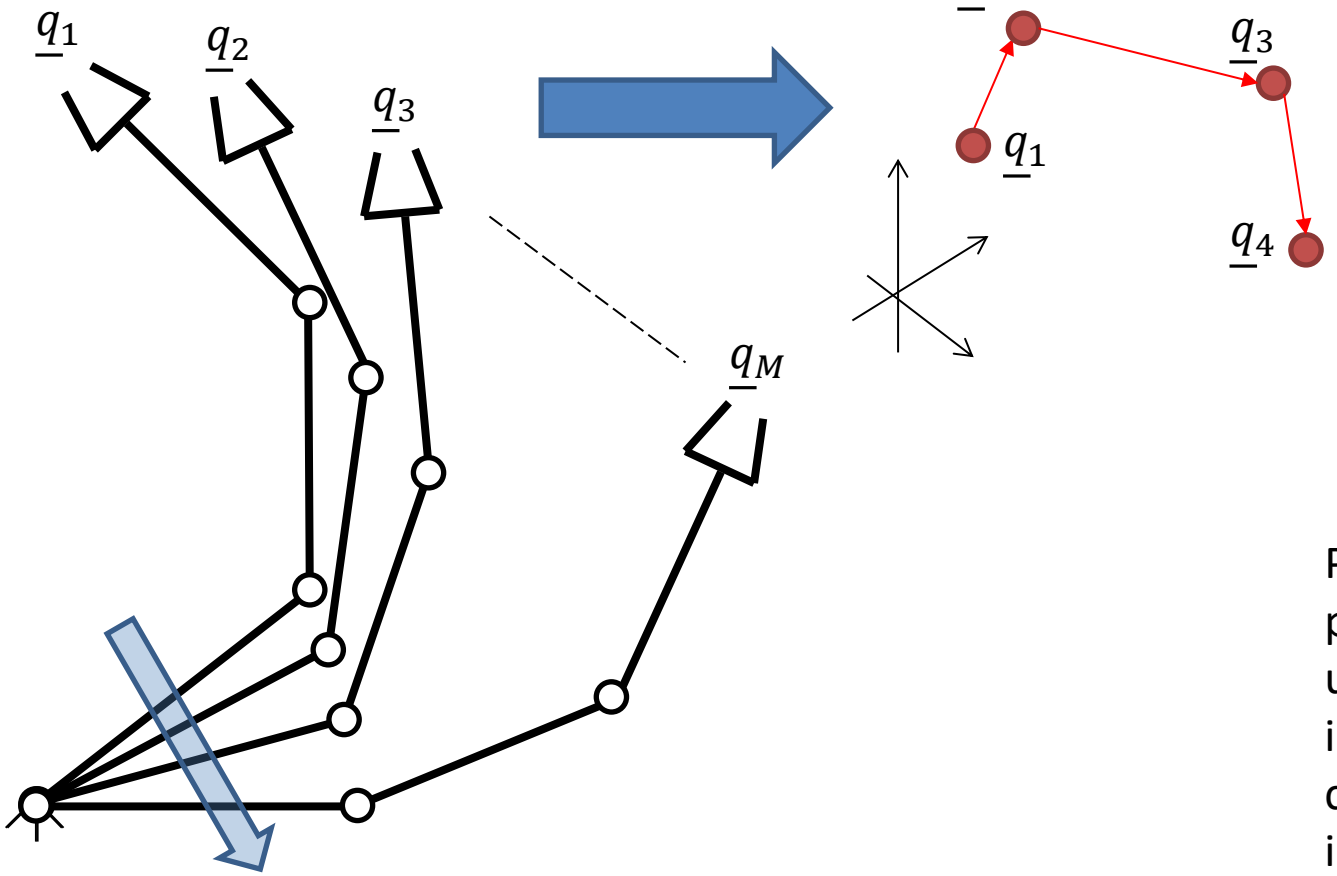


La configurazione (posa) del manipolatore è descritta da un vettore $\underline{q} = \begin{bmatrix} q_1 \\ \vdots \\ q_N \end{bmatrix}$, contenente il valore degli angoli di giunto per la posa considerata

Più in generale, il vettore $\underline{q}$ contiene quell'insieme di informazioni che è necessario conoscere per determinare univocamente lo stato del robot

Rapidly Random Tree (RRT) per Path Planning di manipolatori robotici

Un percorso è una successione di pose $\underline{q}_1 \rightarrow \underline{q}_2 \rightarrow \cdots \rightarrow \underline{q}_M$, che portano il manipolatore da una configurazione $\underline{q}_1$ iniziale ad una finale $\underline{q}_M$.

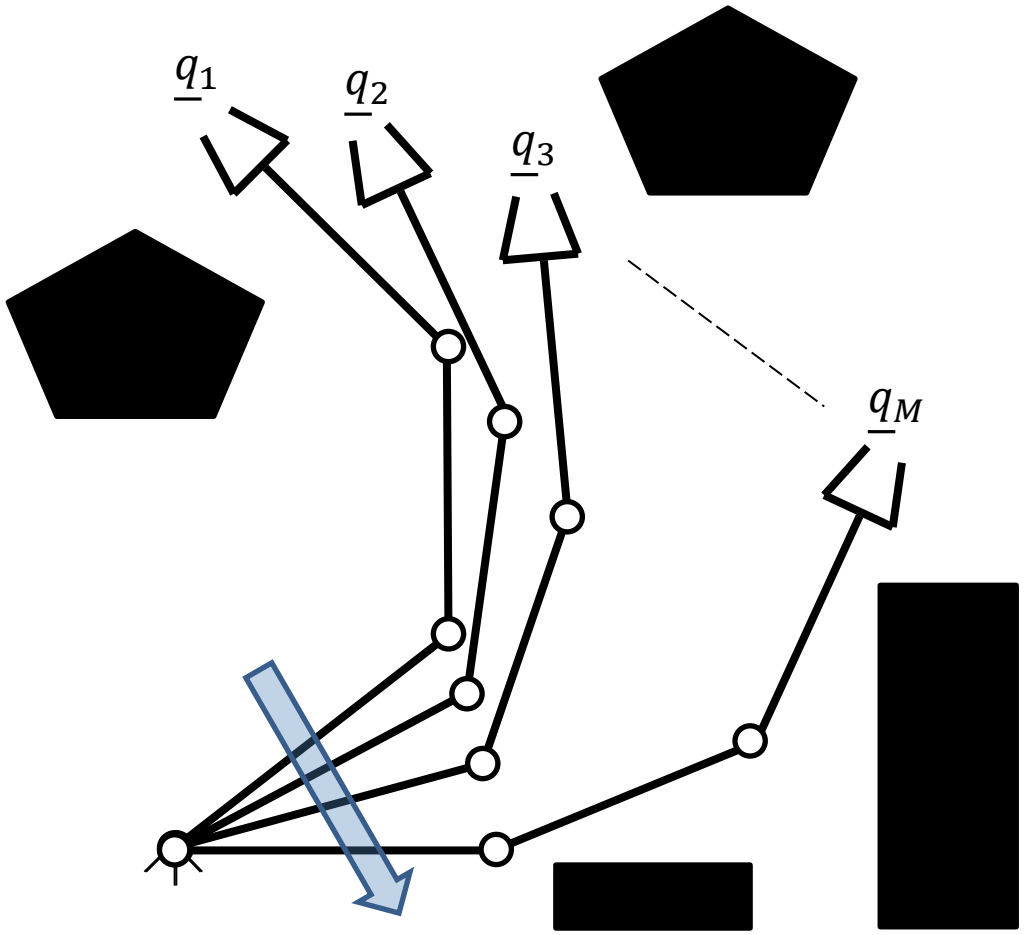


Il percorso seguito dal robot nel tempo può essere rappresentato come una spezzata nello spazio configurazionale (N dimensionale in generale)

Più in generale, il percorso è costituito da un insieme di traiettorie intermedie (ottimali), che congiungono gli stati intermedi

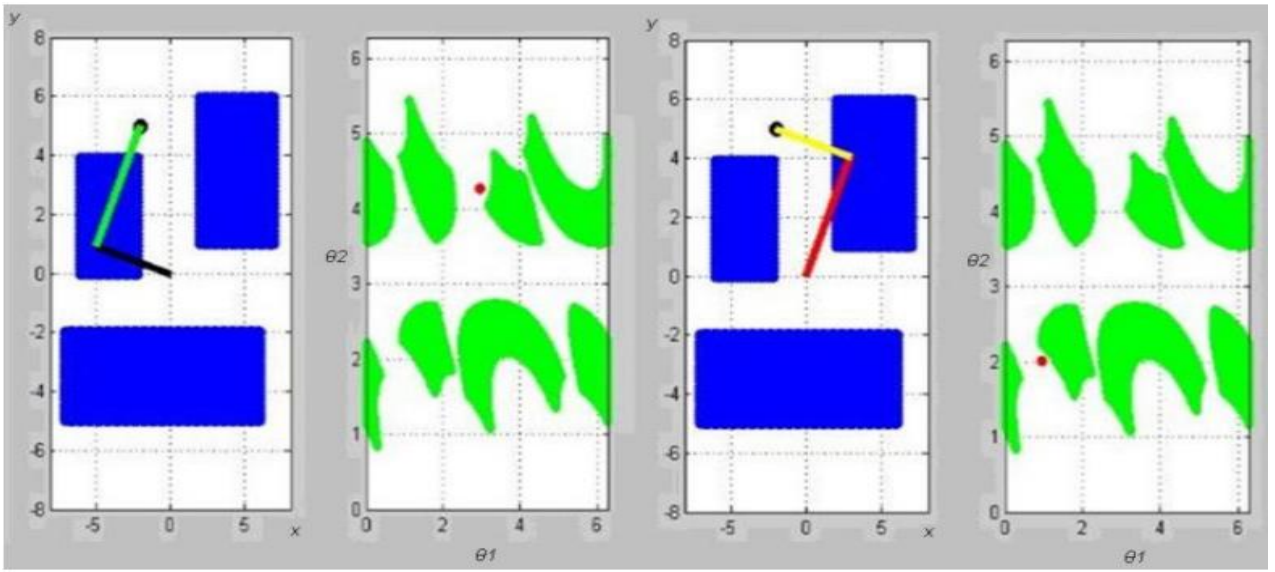
Rapidly Random Tree (RRT) per Path Planning di manipolatori robotici

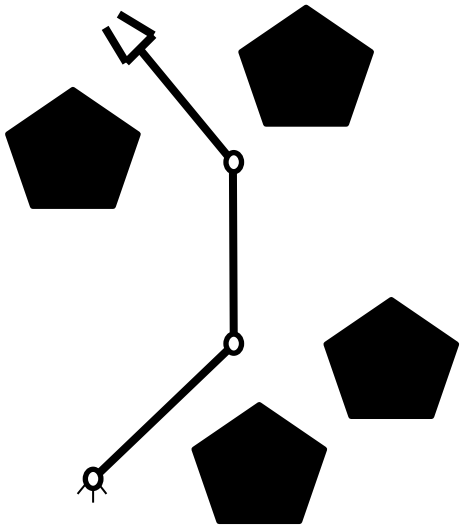
L'ambiente in cui il manipolatore si muove è popolato da una serie di ostacoli noti. I percorsi calcolati per il robot devono quindi essere privi di collisioni



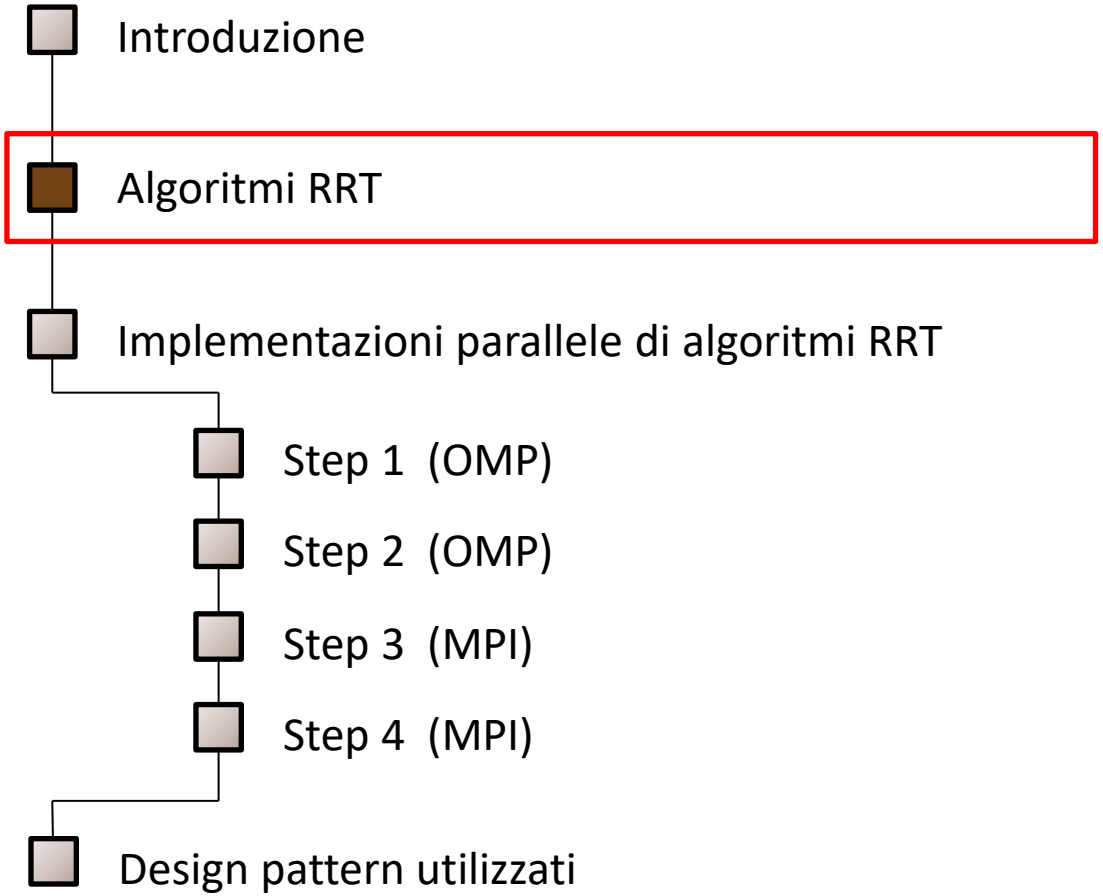
Rapidly Random Tree (RRT)
per Path Planning di manipolatori robotici

Supposto di conoscere la forma di tutti gli ostacoli che popolano la scena, è comunque computazionalmente proibitivo determinare la forma dell'intero insieme Q_{free}, se non per scenari molto semplificati. Risulta invece relativamente semplice determinare se una certa posa $\underline{q}$ appartiene o meno a Q_{free}.





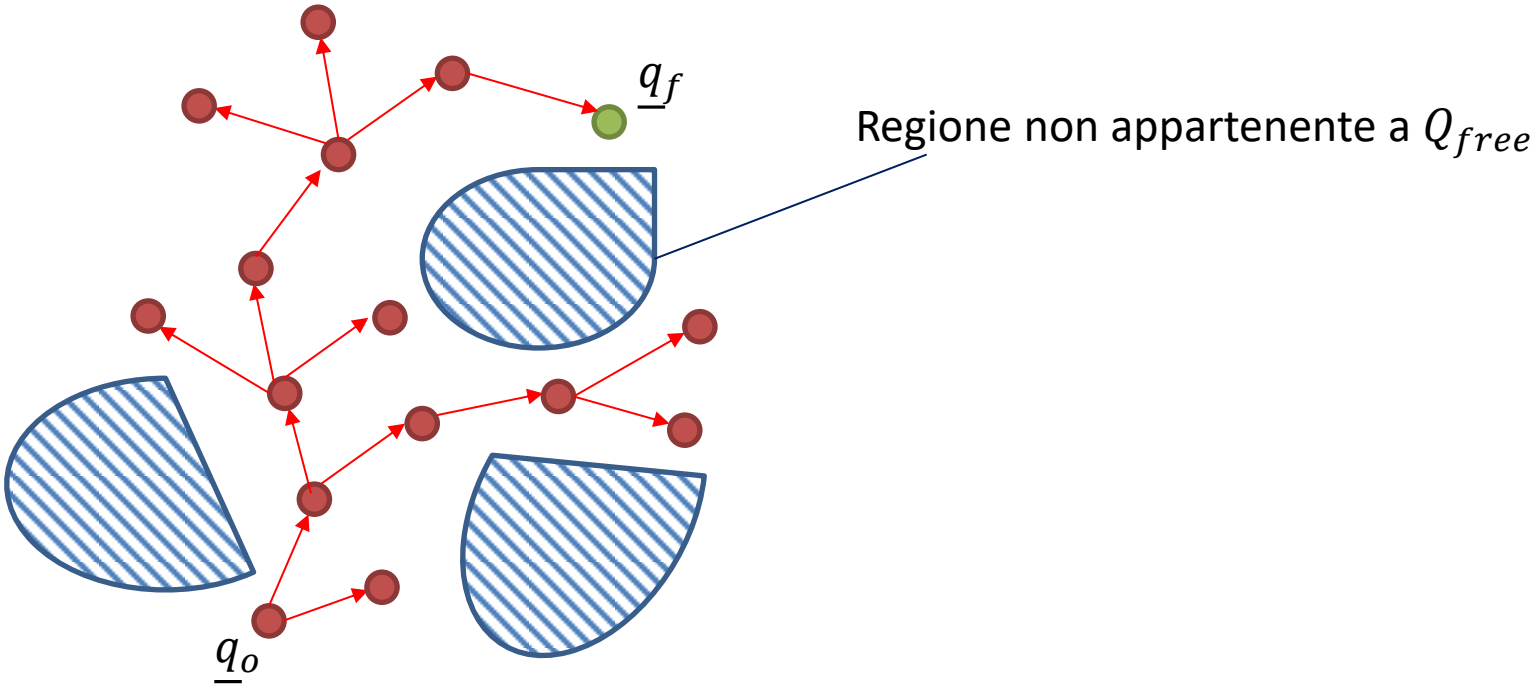
Sommario


Introduzione
Algoritmi RRT
Implementazioni parallele di algoritmi RRT
Step 1 (OMP)
Step 2 (OMP)
Step 3 (MPI)
Step 4 (MPI)
Design pattern utilizzati

Rapidly Random Tree (RRT) per Path Planning di manipolatori robotici

Lo scopo del path planning, è quello di determinare un percorso interamente contenuto nell'insieme Q_{free}, che porti il robot da una posa di partenza q_o ad una di arrivo q_f.

Gli algoritmi di ricerca randomica (RRT) del percorso, risolvono questo problema esplorando Q_{free} , costruendo in maniera iterativa un albero di ricerca. I nodi dell'albero sono degli stati raggiunti dal sistema (la sola posizione dei giunti, la posizione e la velocità, ecc..) a cui ci si riferirà con il termine generico q; mentre i rami sono delle traiettorie che li collegano (successioni di pose o sequenze di accelerazioni o coppie da imprimere ai giunti).



RRT, versione base (seriale)

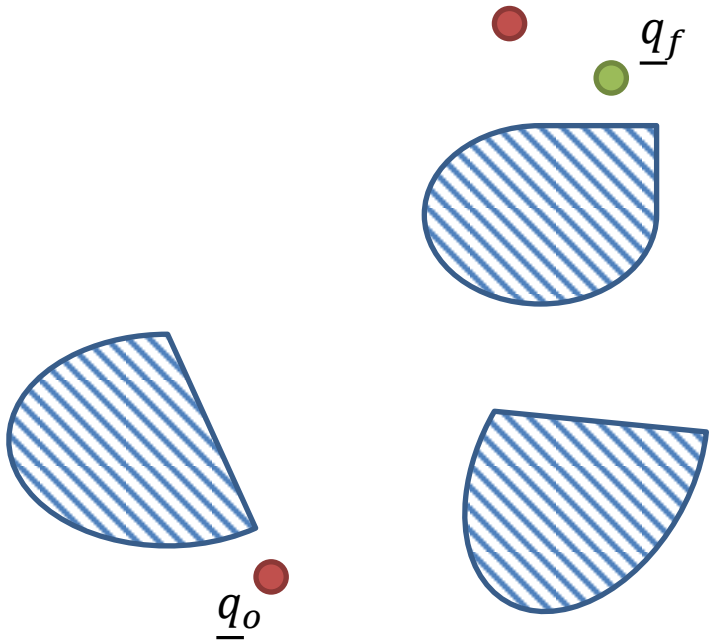
- La ricerca parte inizializzando l'albero con un nodo radice, il cui stato associato è $\underline{q_o}$

- Viene campionato in maniera randomica un nuovo stato $\underline{q_{rand}}$
- Viene trovato all'interno dell'albero esistente lo stato $\underline{q_{near}}$ che più è 'vicino' a $\underline{q_{rand}}$ secondo una qualche metrica (es. la distanza euclidea nello spazio delle configurazioni, il tempo speso per andare da uno stato all'altro seguendo una certa traiettoria ottimale che ignora la presenza di ostacoli, ecc..)
- Si calcola una posa $\underline{q_{extend}}$, raggiungibile seguendo traiettoria che porterebbe a $\underline{q_{rand}}$, ma 'distante' da $\underline{q_{near}}$ non più di un certo valore fissato (exploration degree).
- Si verifica se $\underline{q_{extend}}$ appartiene a Q_{free}:
  - Si: si aggiunge $\underline{q_{extend}}$ all'albero
  - No: $\underline{q_{extend}}$ viene scartato

- Si itera fino a che non si è trovato uno stato sufficientemente vicino a $\underline{q_f}$

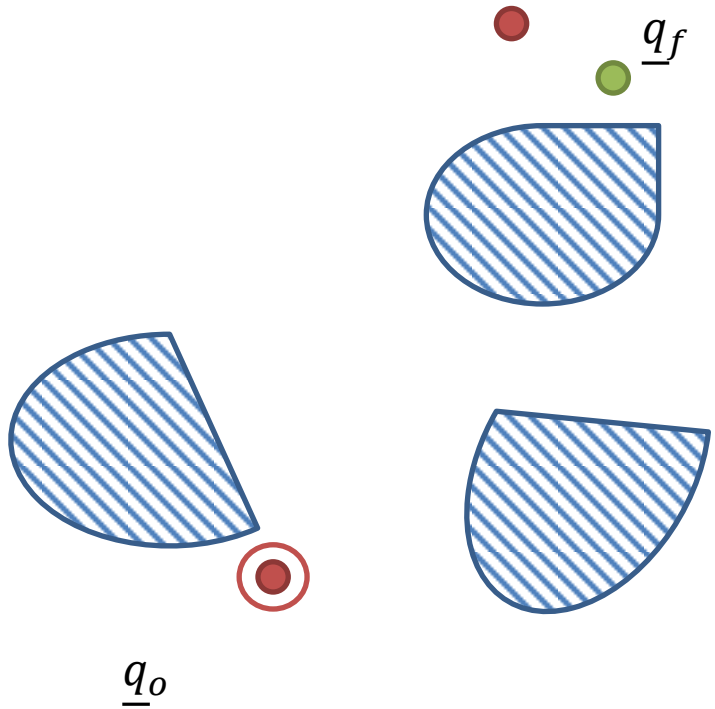
RRT, versione base (seriale)

Campionamento di un nuovo stato randomico

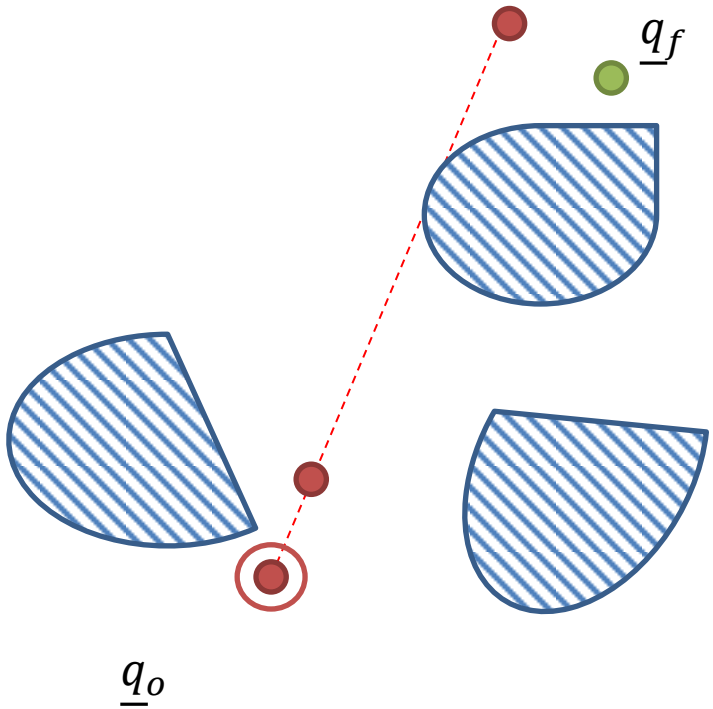


RRT, versione base (seriale)

Ricerca del nearest neighbour q_{near}



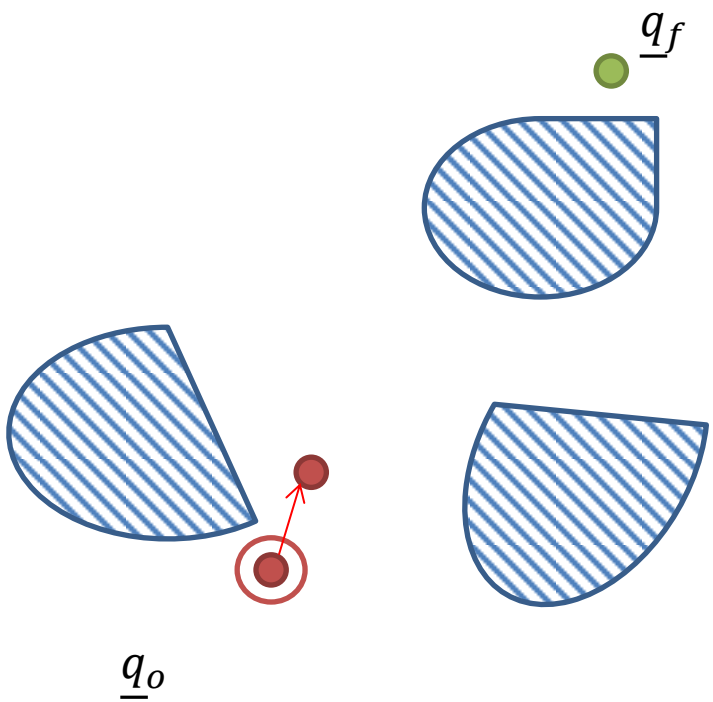
RRT, versione base (seriale)



Calcolo di q_{extend} e
Controllo sul fatto che q_{extend} appartenga a Q_{free}

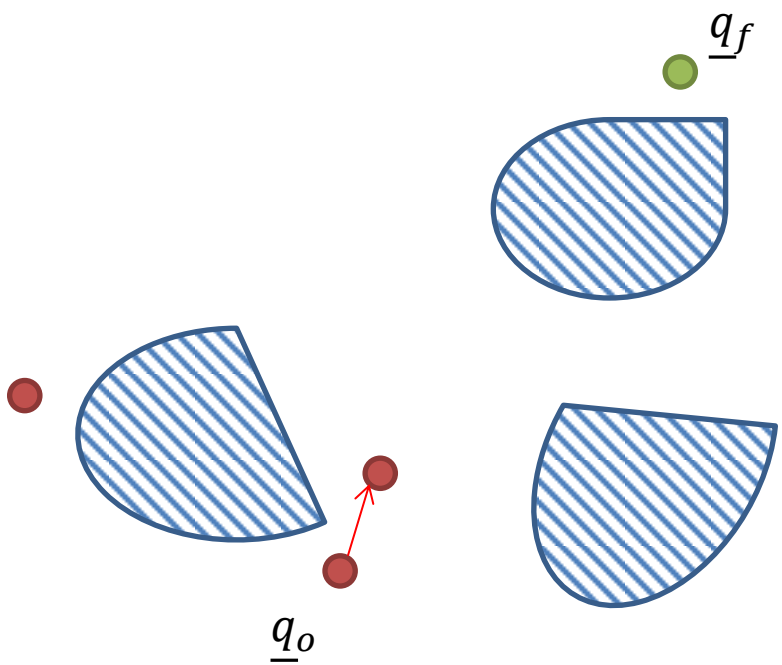
RRT, versione base (seriale)

Aggiunta di q_{extend} all'albero di ricerca



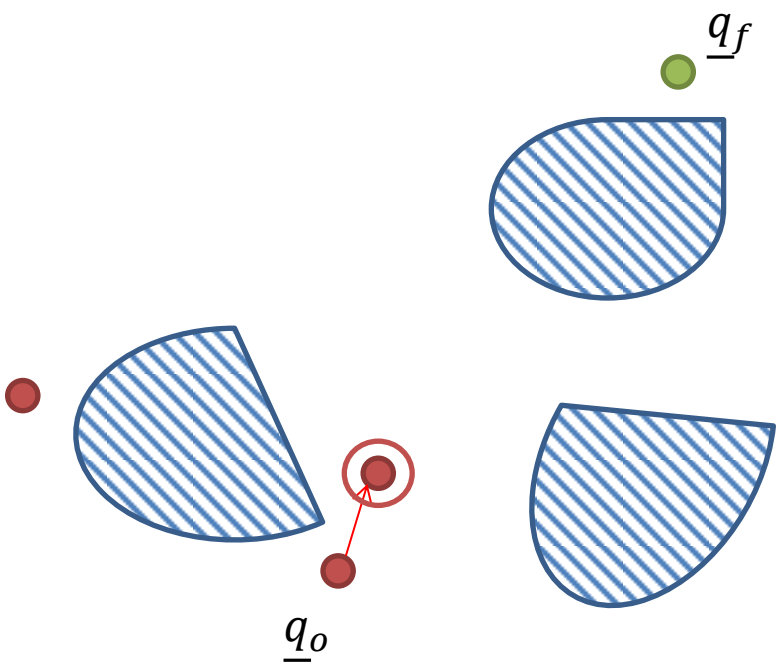
RRT, versione base (seriale)

Campionamento stato randomico

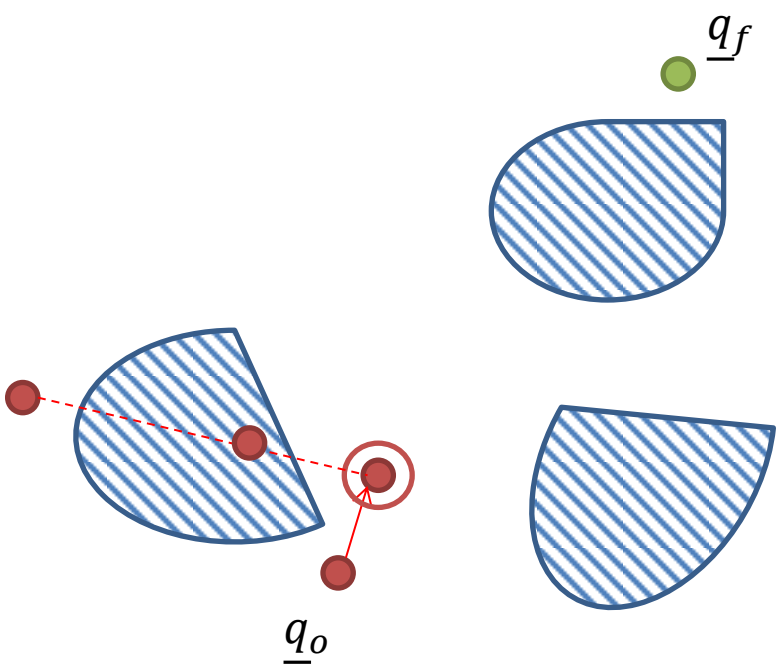


RRT, versione base (seriale)

Ricerca del nearest neighbour q_{near}



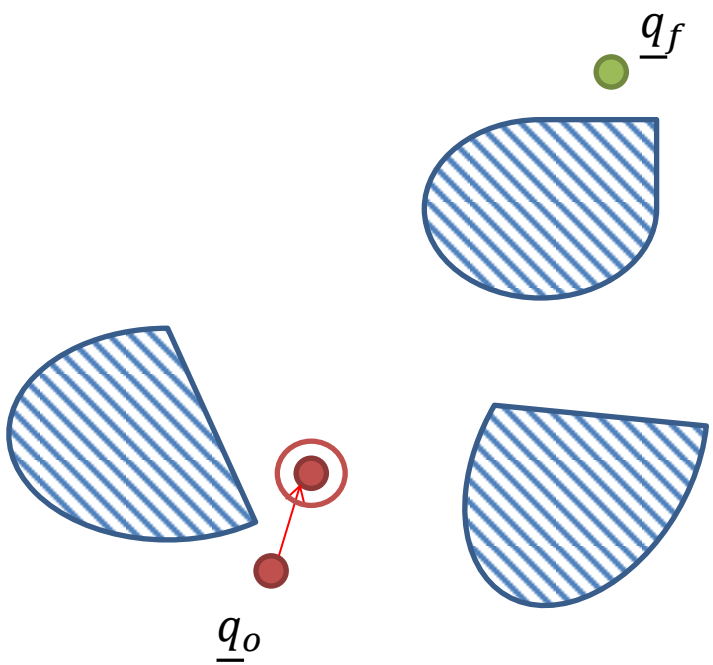
RRT, versione base (seriale)



Calcolo di q_{extend} e
Controllo sul fatto che q_{extend} appartenga a Q_{free}

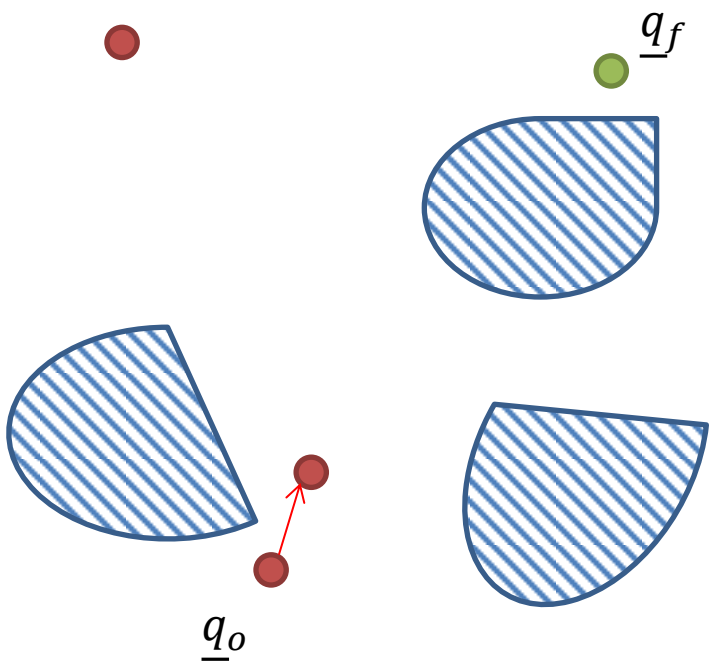
RRT, versione base (seriale)

Scarto q_{extend}



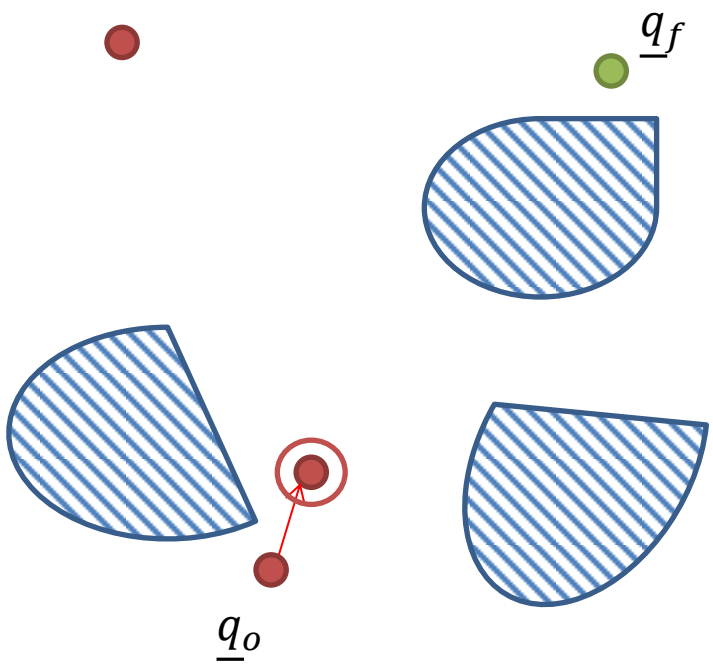
RRT, versione base (seriale)

Campionamento stato randomico

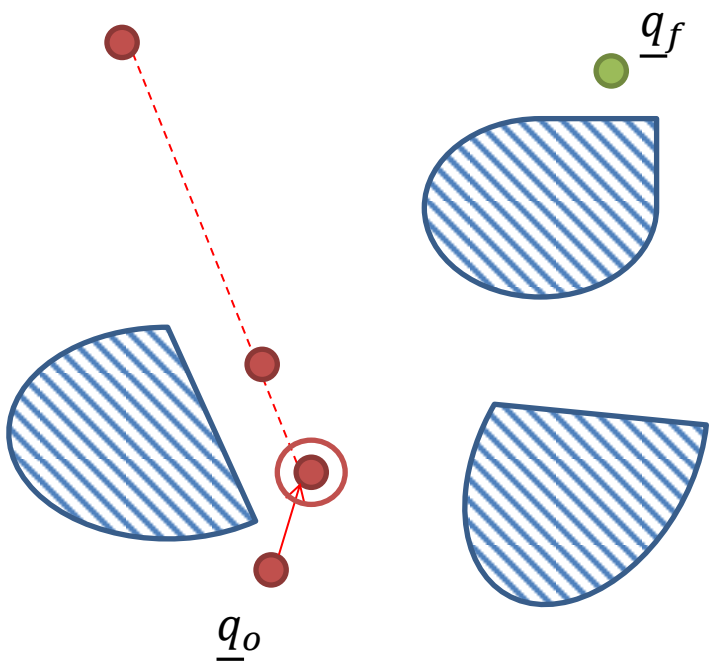


RRT, versione base (seriale)

Ricerca del nearest neighbour q_{near}



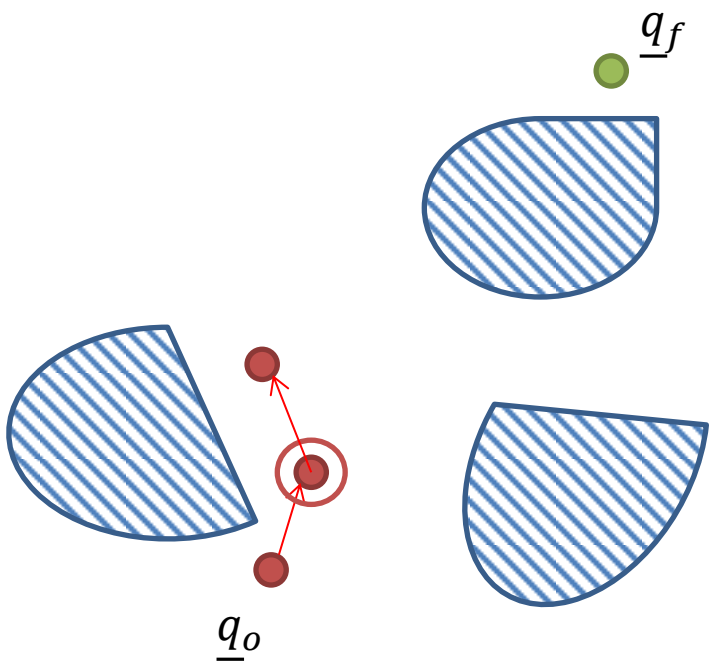
RRT, versione base (seriale)



Calcolo di q_{extend} e
Controllo sul fatto che q_{extend} appartenga a Q_{free}

RRT, versione base (seriale)

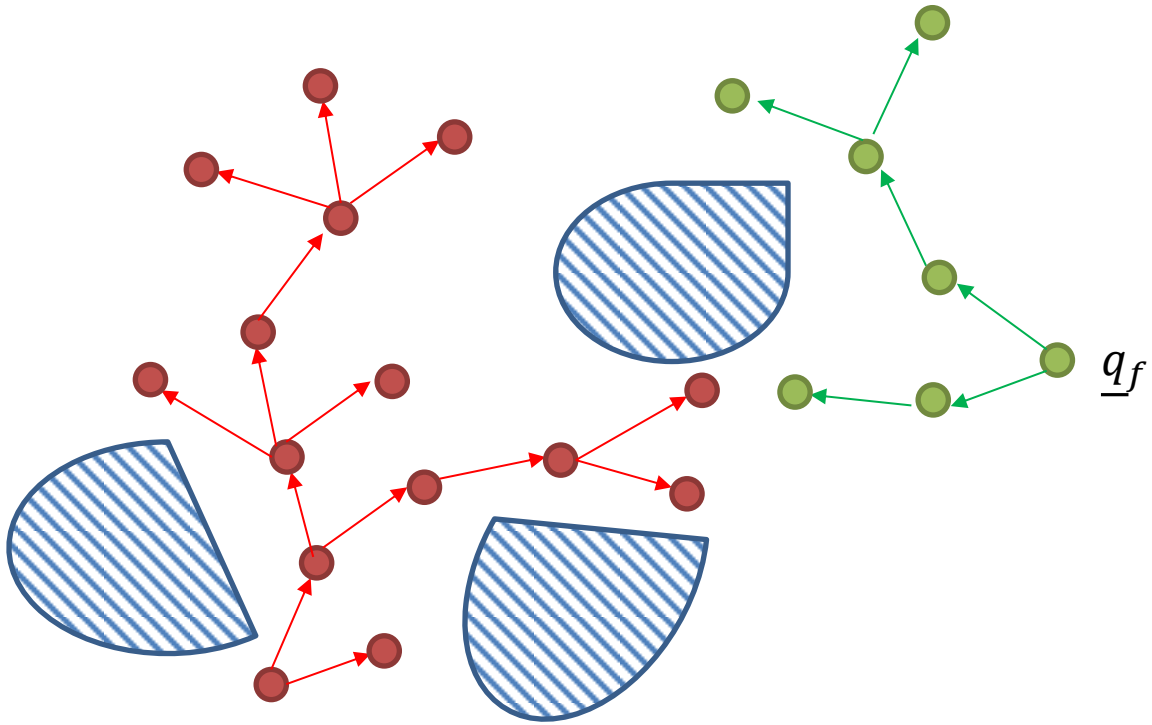
Aggiunta di q_{extend} all'albero di ricerca



E così via…

Variante dell' RRT: RRT bidirezionale

Vengono estesi in simultanea 2 alberi di ricerca: uno avente come nodo radice $\underline{q_o}$, e uno avente come radice $\underline{q_f}$. Ad ogni iterazione si considera un albero 'master' ed uno 'slave': il master viene esteso verso uno stato $\underline{q_{rand}}$ randomicamente generato e nel caso di estensione riuscita, si tenta di estendere lo slave verso il nodo q_{extend} aggiunto all'albero master. Per l'iterazione successiva i ruoli tra gli alberi vengono invertiti. L'algoritmo ha termine quando si riesce a determinare un collegamento fra i due alberi



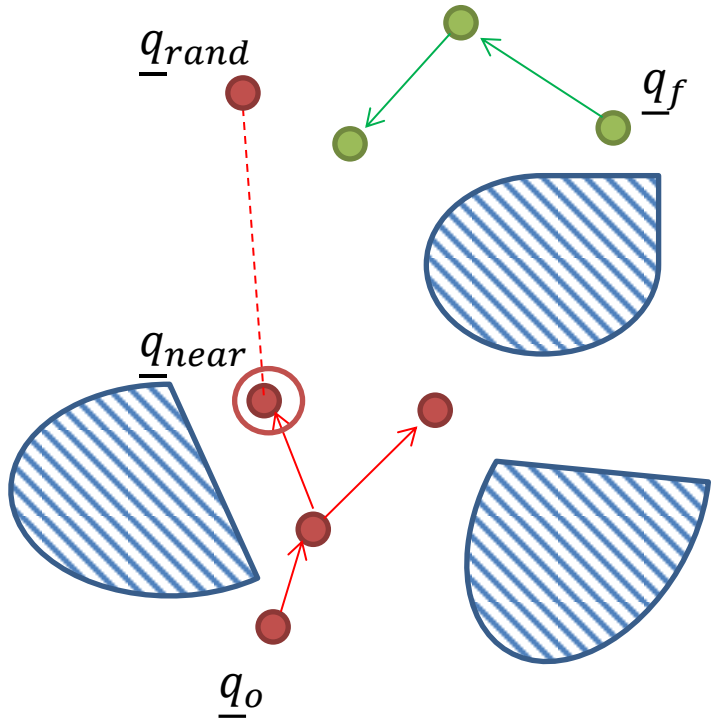
Variante dell' RRT: RRT bidirezionale

- La ricerca parte inizializzando due alberi : uno con radice $\underline{q_o}$, l'altro con radice $\underline{q_f}$
- Si assume il primo albero come master e il secondo come slave

- Viene campionato in maniera randomica un nuovo stato $\underline{q_{rand}}$
- Viene trovato all'interno dell'albero master lo stato $\underline{q_{near}}$ che più è 'vicino' a $\underline{q_{rand}}$
- Si calcola una posa $\underline{q_{extend}}$, raggiungibile seguendo traiettoria che porterebbe a $\underline{q_{rand}}$, ma 'distante' da $\underline{q_{near}}$ non più di un certo valore fissato (exploration degree).
- Si verifica se $\underline{q_{extend}}$ appartiene a Q_{free}:
  - Si: si aggiunge $\underline{q_{extend}}$ all'albero
  - No: $\underline{q_{extend}}$ viene scartato
- Viene trovato all'interno dell'albero slave lo stato $\underline{q'_{near}}$ più vicino a $\underline{q_{extend}}$
- Si calcola una posa $\underline{q'_{extend}}$, raggiungibile seguendo traiettoria che porterebbe a $\underline{q'_{near}}$, ma 'distante' da $\underline{q'_{near}}$ non più di un certo valore fissato (exploration degree).
- Si invertono master e slave

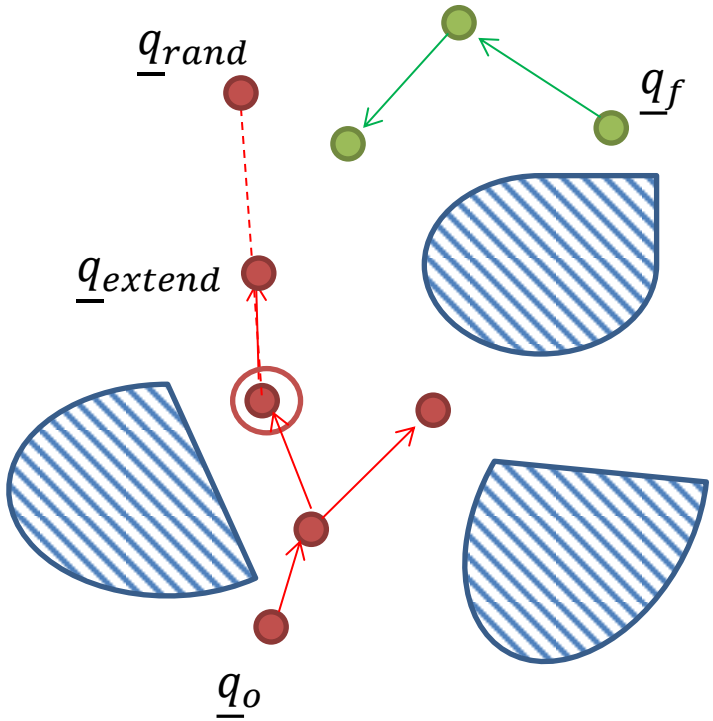
- Si itera fino a che non si è trovato un $\underline{q'_{extend}}$ sufficientemente vicino a $\underline{q_{extend}}$: in tal caso i due alberi si sono incontrati

Variante dell' RRT: RRT bidirezionale



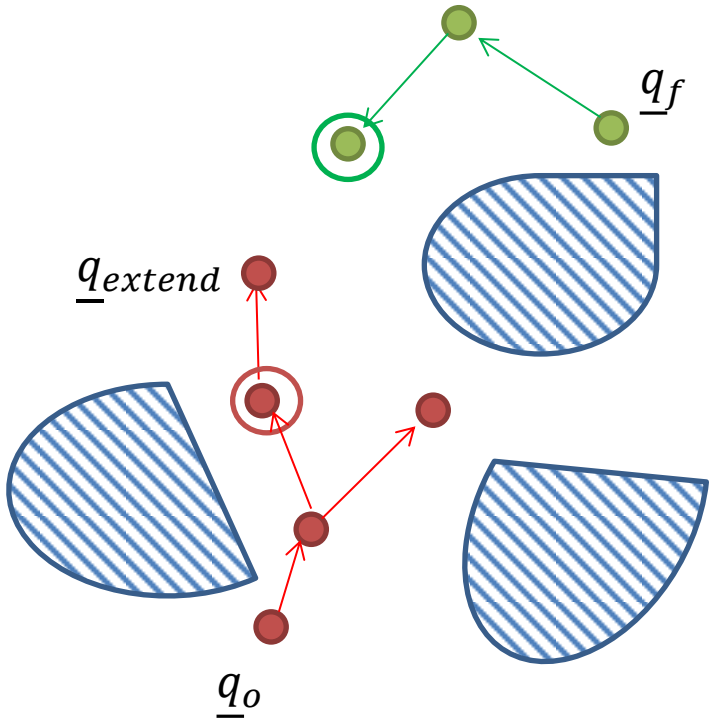
Generazione di q_{rand} e ricerca di q_{near}

Variante dell' RRT: RRT bidirezionale



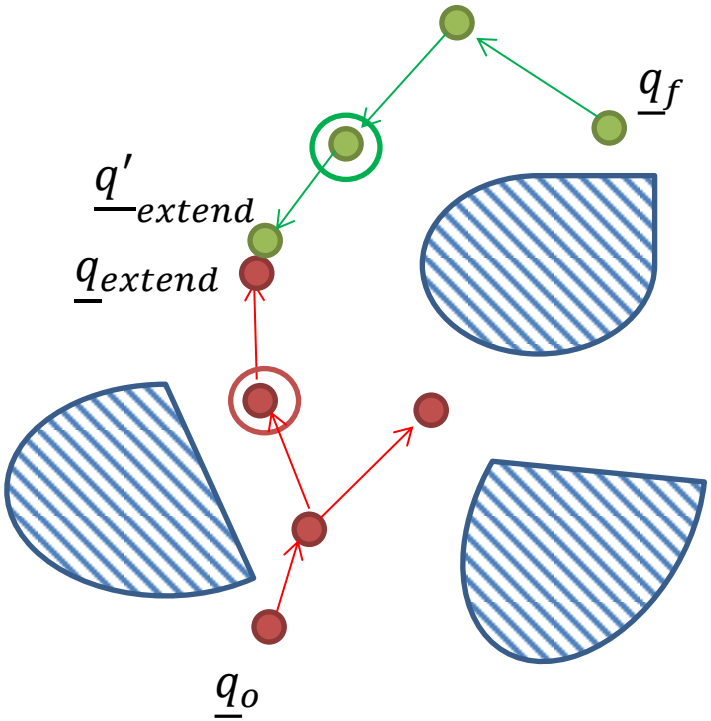
Determinazione di q_{extend}

Variante dell' RRT: RRT bidirezionale



Ricerca di q'_{near}

Variante dell' RRT: RRT bidirezionale

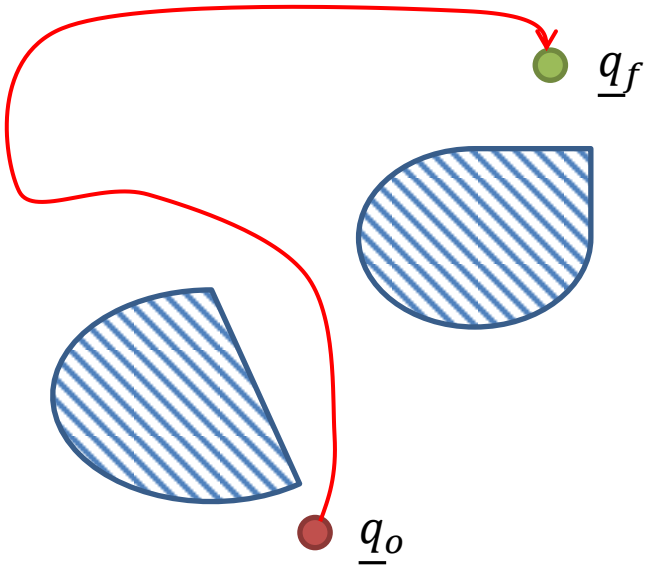


Determinazione di q'_{extend}

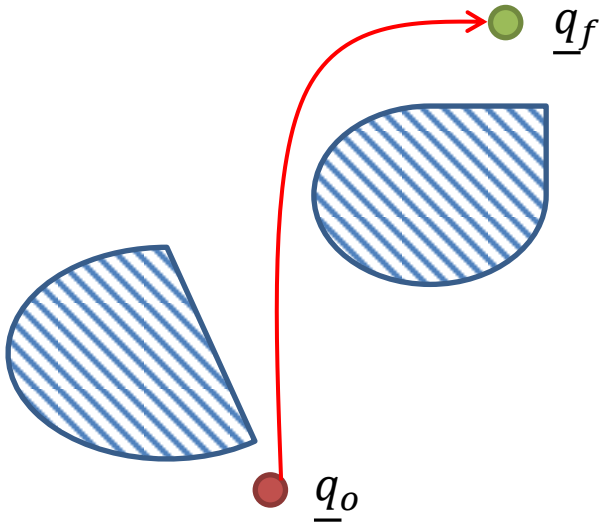
Variante dell' RRT: RRT*

Gli algoritmi RRT sono in grado di trovare un percorso interamente contenuto in Q_{free} che congiunge uno stato di partenza voluto con uno di arrivo. Tuttavia la soluzione trovata è sub-ottimale con probabilità = 1. La variante RRT* è in grado di ovviare a questo problema, essendo in grado di ottenere un percorso fattibile (assenza collisioni) minimizzante una certa cifra di merito (ad esempio lo spazio complessivamente percorso).

Soluzione sub-ottimale



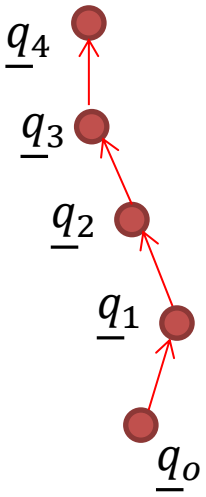
Soluzione ottimale



L'implementazione di un algoritmo di tipo RRT* prevede di possedere una funzione $C(q_1, q_2)$ in grado di calcolare il cost-to-go della traiettoria che procede da q_1 e termina in q_2.

Variante dell' RRT: RRT*

Il cost-to-go dal nodo radice dell'albero a un generico nodo q_1, verrà indicato con C(q_1). C(q_1) non è altro che una somma costi C(q_{k-1}, q_k):



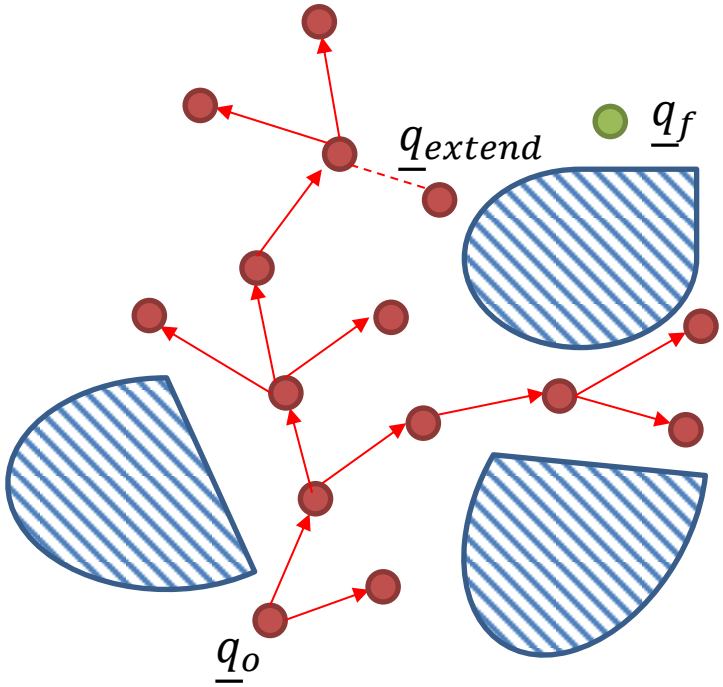
$$C(q_4)=\sum_{k=1}^{4} C(q_{k-1}, q_k)$$

La conoscenza del 'padre' da cui discende ogni nodo nell'albero, è fondamentale per lo svolgimento delle iterazioni associate alla versione RRT*.

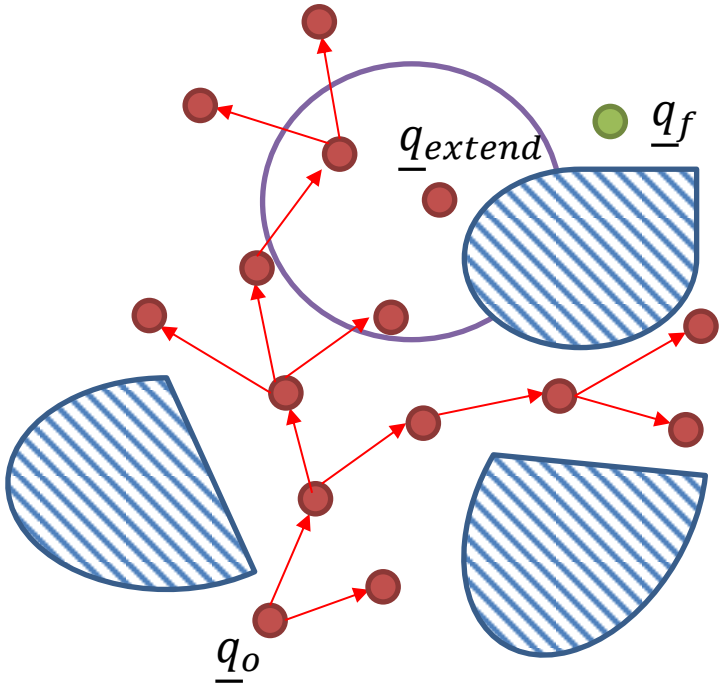
Variante dell' RRT: RRT*

- La ricerca parte inizializzando l'albero con un stato radice, la cui posa associata è $\underline{q}_o$
- Viene campionato in maniera randomica un nuovo stato $\underline{q}_{rand}$
- Viene trovato all'interno dell'albero esistente lo stato $\underline{q}_{near}$
- Si calcola una stato $\underline{q}_{extend}$ che giace lungo la traiettoria che porterebbe a $\underline{q}_{rand}$, per cui il tratto $\underline{q}_{near}$ -> $\underline{q}_{extend}$ è interamente privo di collisioni
- L'estensione viene accetata (almeno un $\underline{q}_{extend}$ viene trovato):
  - Si: si aggiunge $\underline{q}_{extend}$ all'albero
  - No
- Si determina l'insieme $Near = \{ \| \underline{q}_i - \underline{q}_{extend} \| \leq \left(\frac{\log(N)}{N}\right)^d \}$, dove N è il numero di nodi presenti nell'albero e d è la dimensione dello stato che caratterizza il generico nodo $\underline{q}$. La distanza $\underline{q}_i - \underline{q}_{extend}$, è da intendersi ancora una volta come una metrica associata alla traiettoria che porta da $\underline{q}_i$ a $\underline{q}_{extend}$. $Near$ rappresenta un insieme di stati candidati a cui collegare $\underline{q}_{extend}$.
- Si ordina $Near$ in maniera crescente rispetto ai valori di $C(\underline{q}_i)$
- $\underline{q}_{extend}$ viene collegato al primo stato $\underline{q}_{i\,A}$ in $Near$ per cui la traiettoria da quel nodo a $\underline{q}_{extend}$ è completamente contenuta in Q_{free} e tale nodo viene rimosso da $Near$.
- Per ogni nodo in $Near$, se risulta verificato che C($\underline{q}_{extend}$) + C($\underline{q}_{extend}$, $\underline{q}_i$) < C($\underline{q}_{i\,A}$) + C($\underline{q}_{i\,A}$, $\underline{q}_i$) e la traiettoria $\underline{q}_{extend} \rightarrow \underline{q}_i$ è completamente contenuta in Q_{free}, allora $\underline{q}_i$ viene collegato a $\underline{q}_{extend}$. Si effettua un 'rewird'.

Variante dell' RRT: RRT*

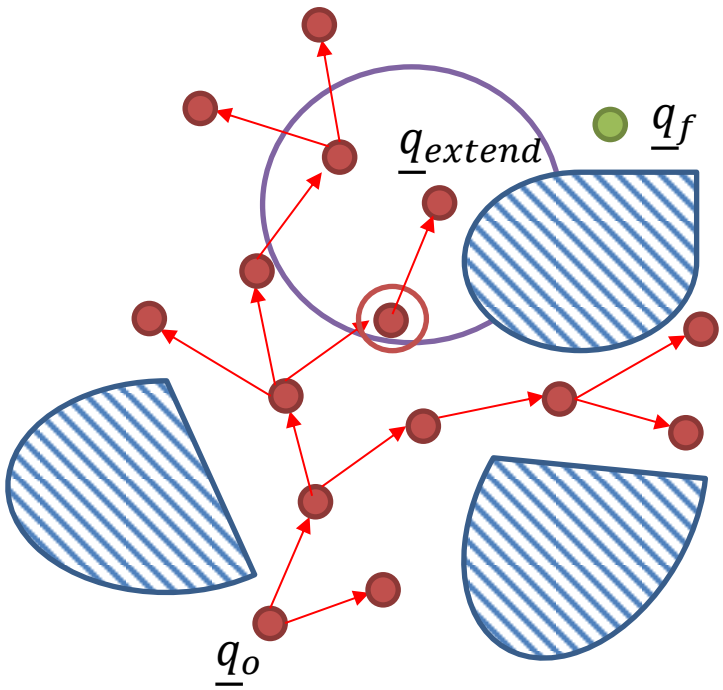


Variante dell' RRT: RRT*



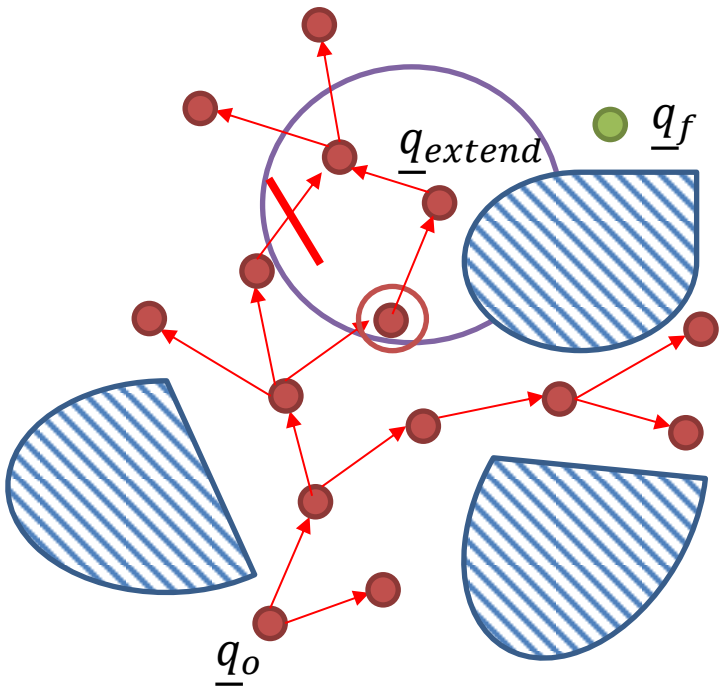
Determinazione del $Near$ set

Variante dell' RRT: RRT*



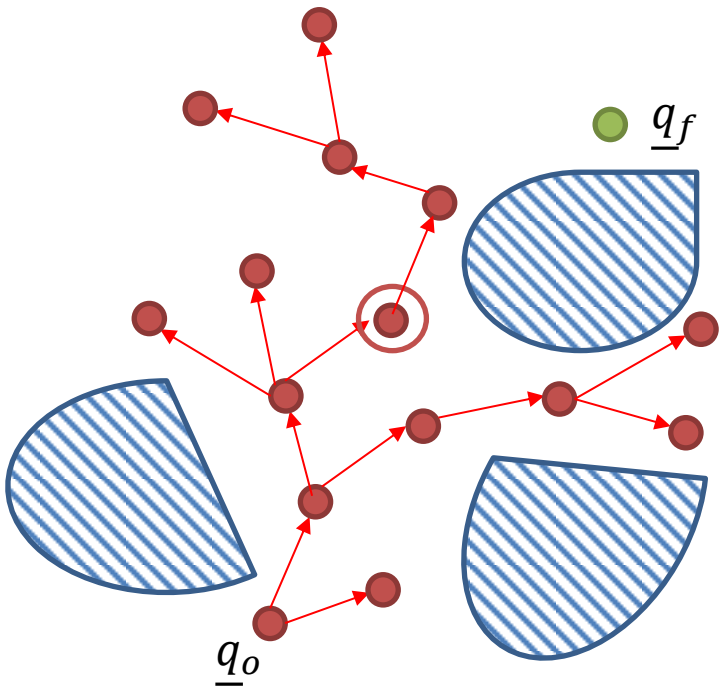
Selezione del nodo per cui la connessione è fattibile (assenza collisioni) e il cost-to-go risulta minimizzato

Variante dell' RRT: RRT*



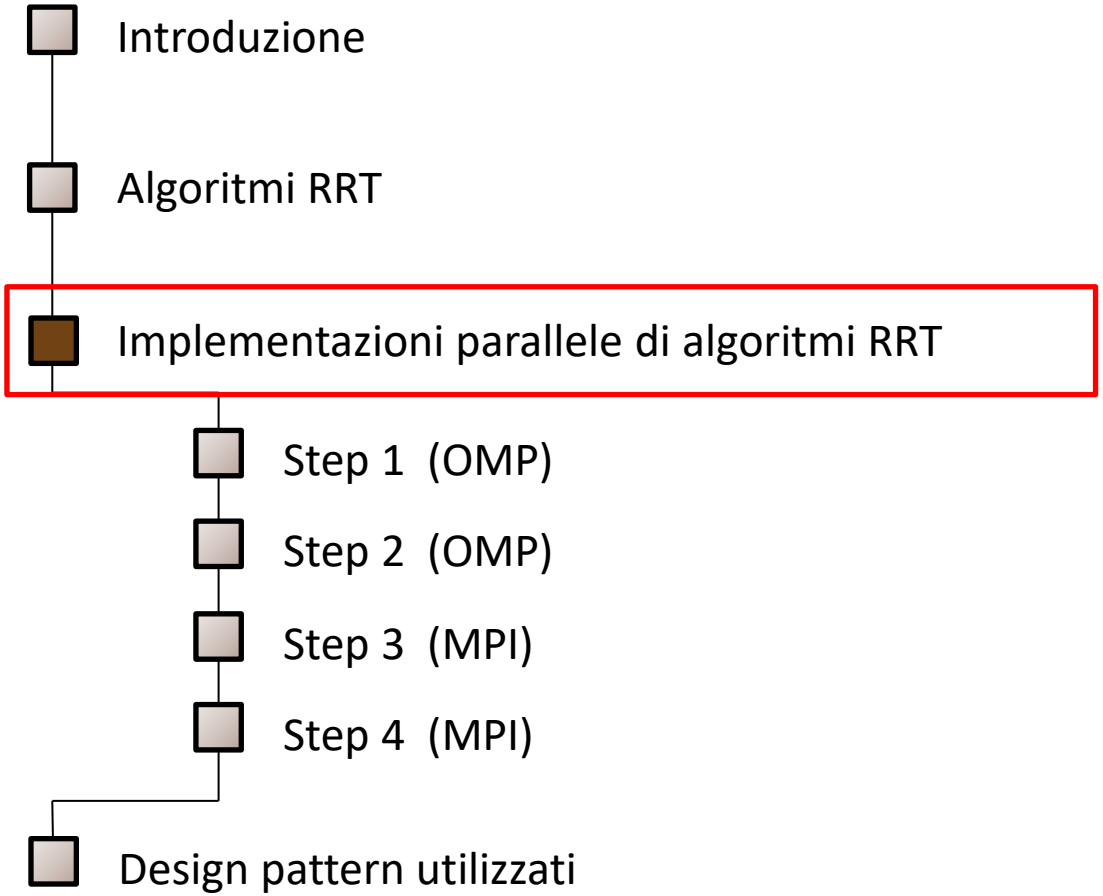
Rewird: valutazione su collegamenti alternativi per i nodi nel $Near$ set

Variante dell' RRT: RRT*



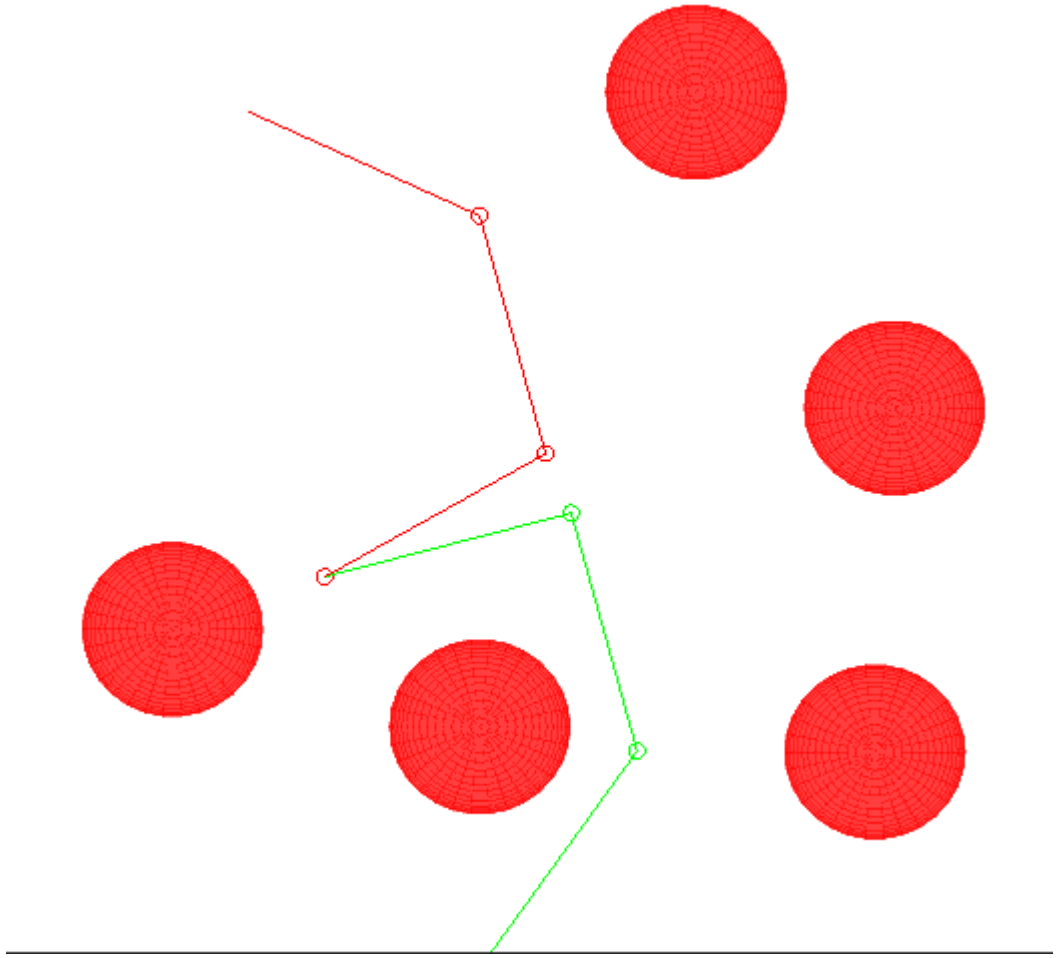
Dopo aver eseguito eventuali ricollegamenti, l'albero è pronto per nuove espansioni

Sommario


Introduzione
Algoritmi RRT
Implementazioni parallele di algoritmi RRT
Step 1 (OMP)
Step 2 (OMP)
Step 3 (MPI)
Step 4 (MPI)
Design pattern utilizzati

RRT-RRT*, parallel implementation

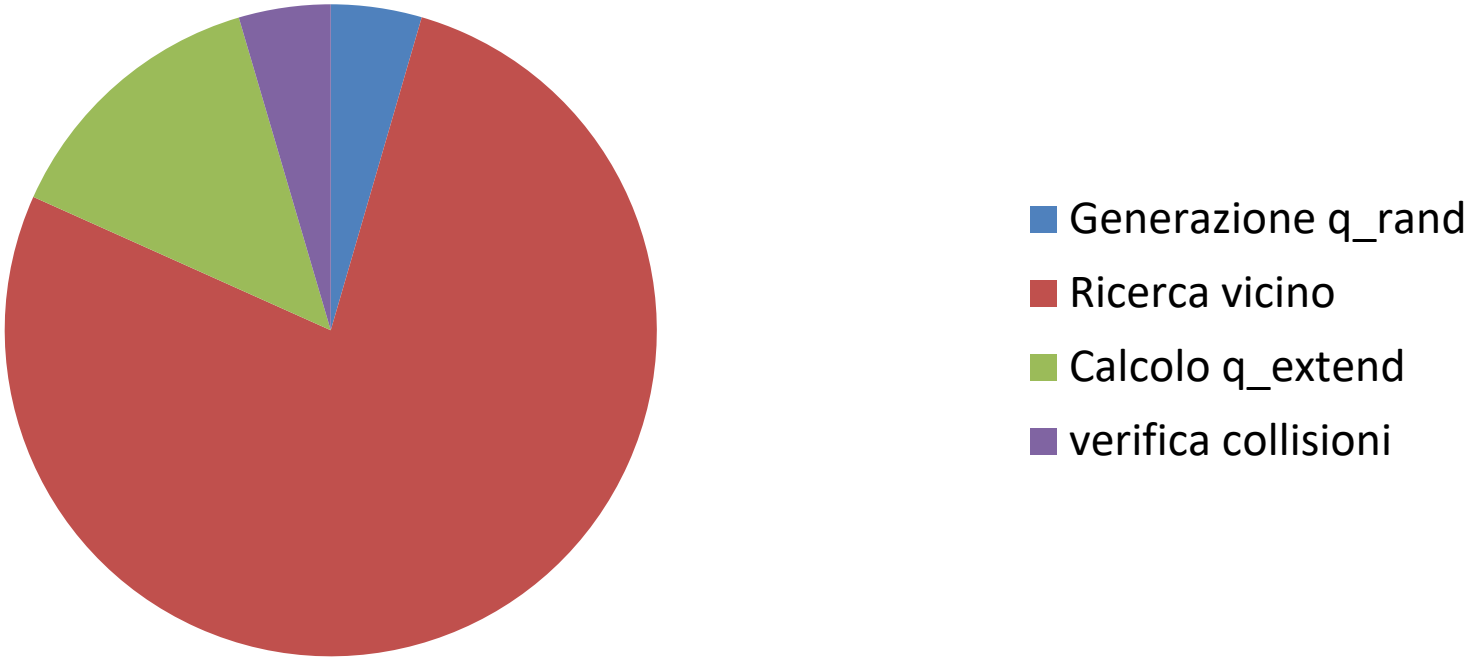
Sarà considerato il seguente problema. Si vuole trovare il percorso che porta un robot dotato di 3 gradi di libertà, dalla configurazione iniziale (in rosso) a quella finale (in verde). Le sfere di colore rosso sono ostacoli fissi (noti), contro cui deve essere impedita la collisione.



RRT, profiling

RRT

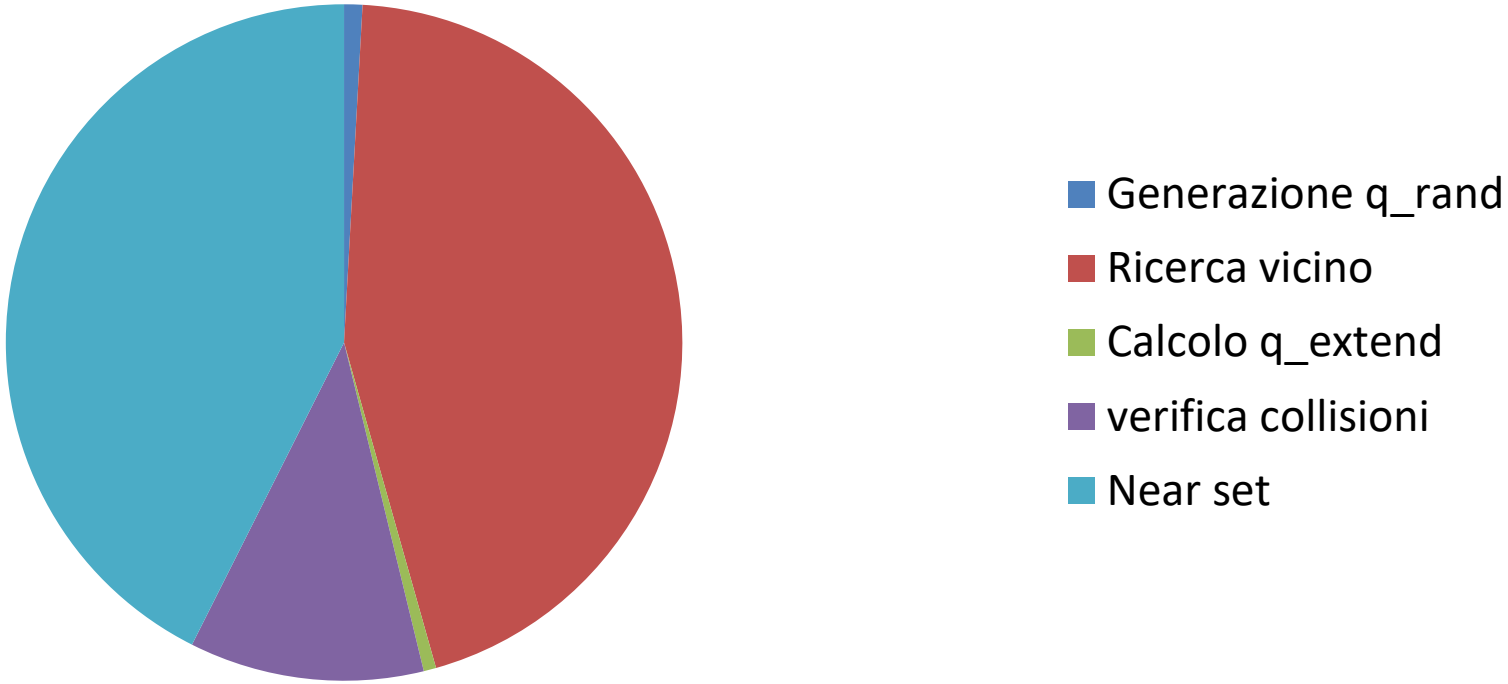
Routine	%tempo
Generazione $\underline{q}_{rand}$	4.50
Ricerca del nodo vicino	77.16
Calcolo della posa $\underline{q}_{extend}$	13.73
Verifica collisioni	4.55



RRT*, profiling

RRT*

Routine	%tempo
Generazione q_{rand}	0,88
Ricerca del nodo vicino	45,10
Calcolo della posa q_{extend}	0,60
Verifica collisioni	11,30
Determinazione $Near$ set	42,92

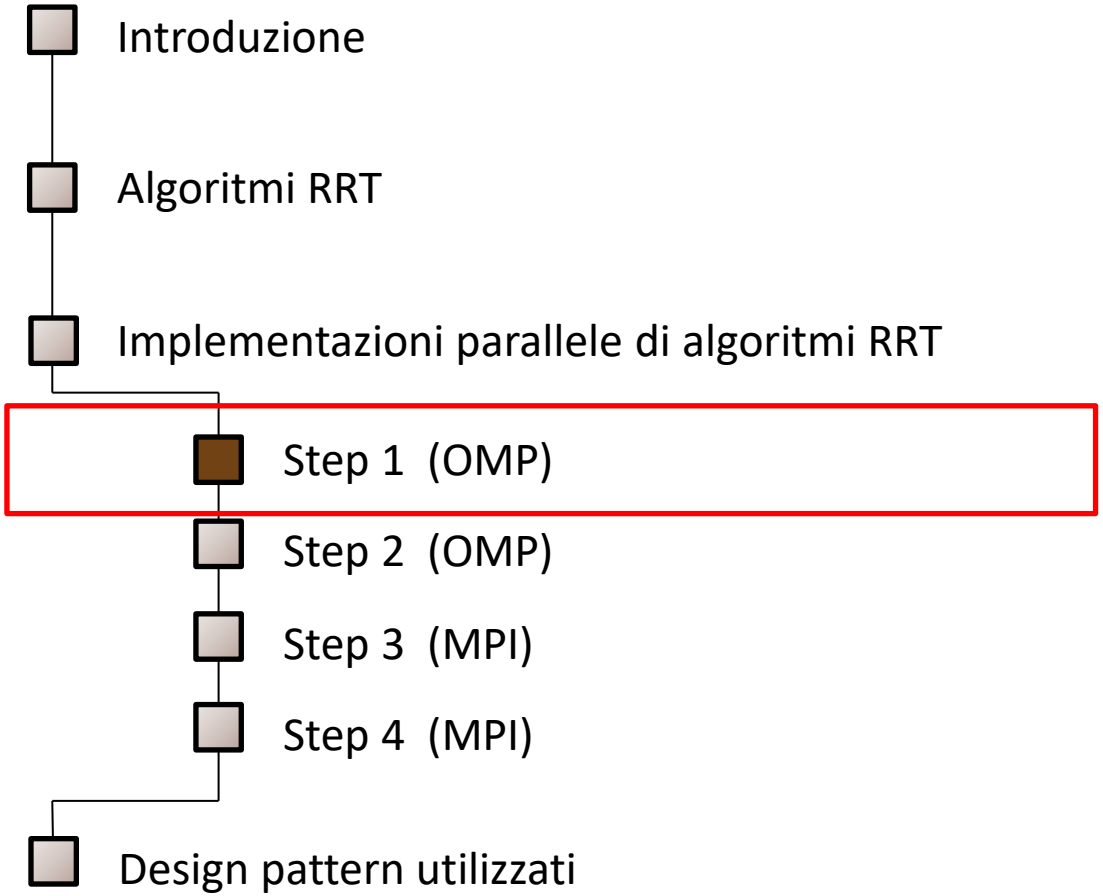


RRT-RRT*, parallel implementation

Nei risultati che verranno presentati si farà variare oltre al numero di threads/processi, anche il tempo di calcolo per ottenere un singolo nodo. La dicitura corrisponde al tempo impiegato per ottenere lo stato q_{extend}, noto il nodo q_{near}. E' una quantità che può variare moltissimo a seconda del tipo di scenario considerato: più ostacoli popolano la scena e più tempo di calcolo è richiesto per determinare se una certa posa presenta collisioni o no.

Si noti come invece, il tempo di copiatura di un nodo (fra threads o processi), non cambia al variare dello scenario, a parità di manipolatore considerato. Il rapporto fra il tempo di ottenimento di q_{extend} e quello di copiatura di un nodo diventa quindi cruciale per valutare la bontà degli algoritmi di parallelizzazione.

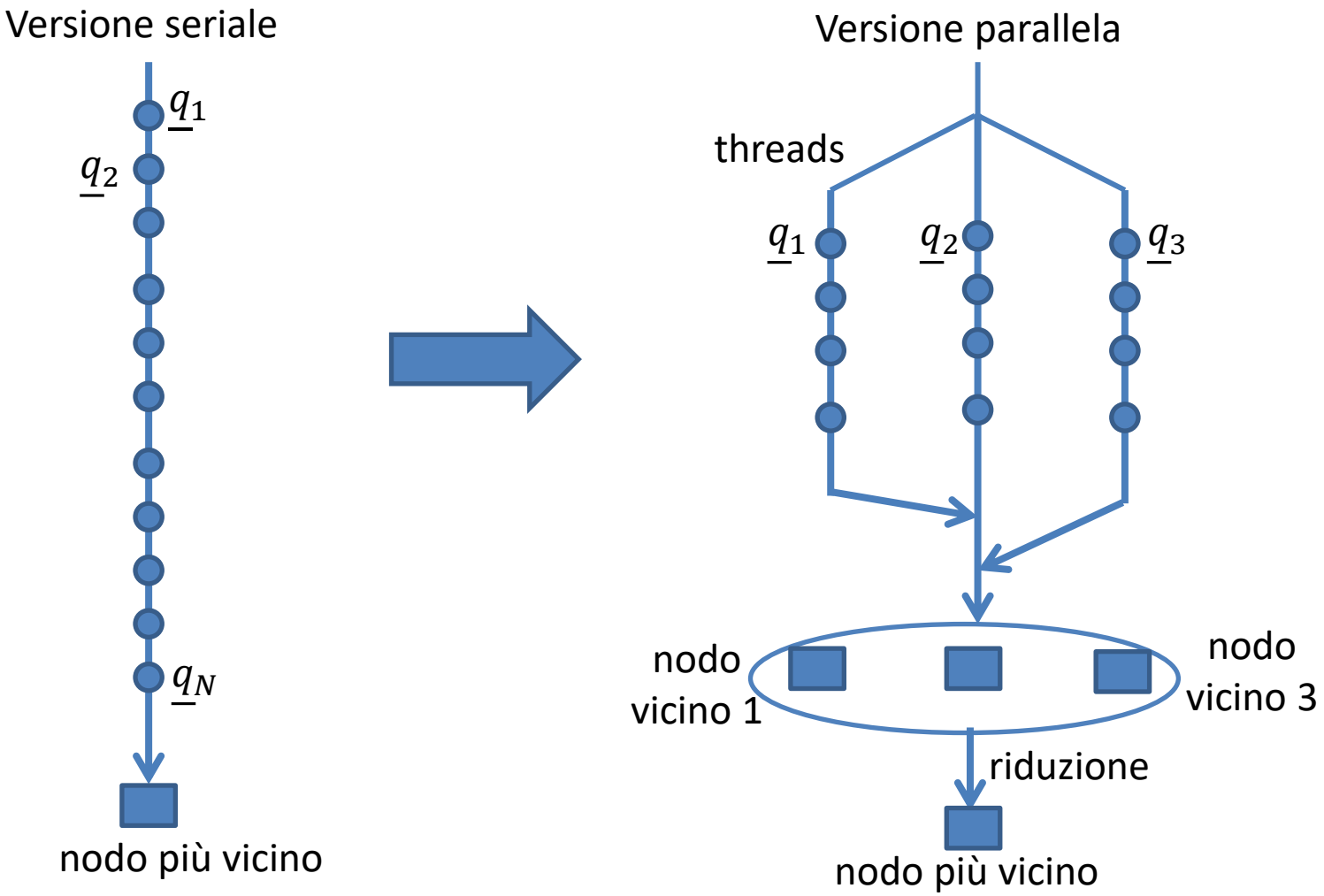
Sommario


Introduzione
Algoritmi RRT
Implementazioni parallele di algoritmi RRT
Step 1 (OMP)
Step 2 (OMP)
Step 3 (MPI)
Step 4 (MPI)
Design pattern utilizzati

RRT-RRT*, parallel implementation

Step 1: parallelizzare la ricerca di $\underline{q}_{near}$ usando openMP

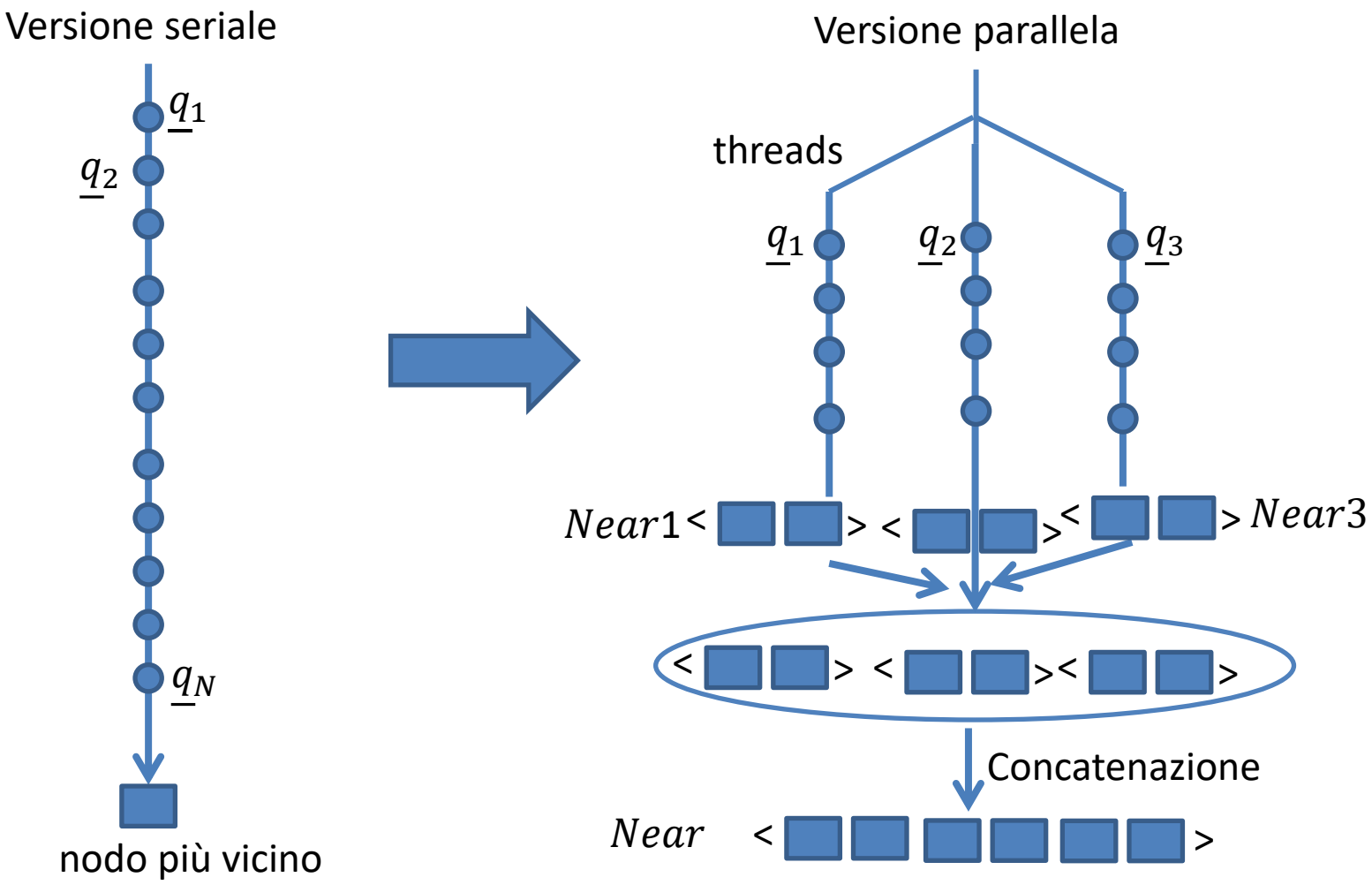
Scorrendo la lista dei nodi dell'albero = $< \underline{q}_1, \underline{q}_2, \cdots, \underline{q}_N >$, si tiene traccia del nodo più vicino facendo confronti successivi della distanza tra i vari nodi e $\underline{q}_{rand}$.



RRT-RRT*, parallel implementation

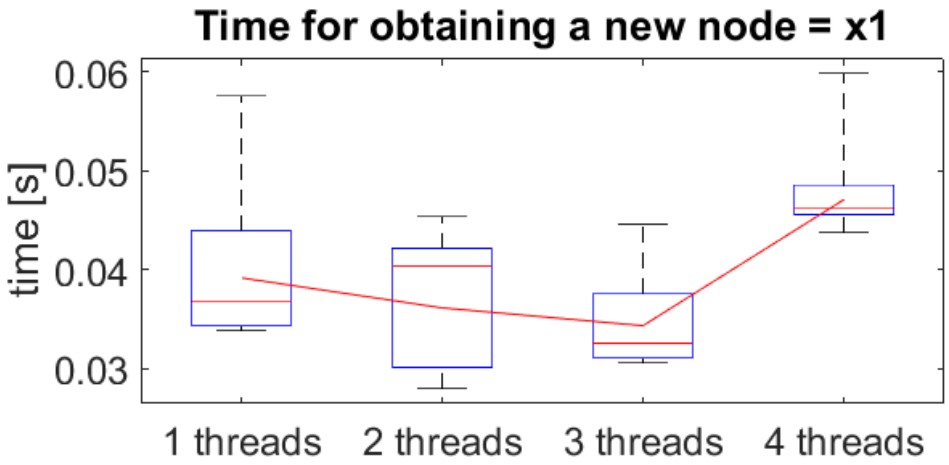
Step 1: parallelizzare il calcolo del set $Near$ usando openMP

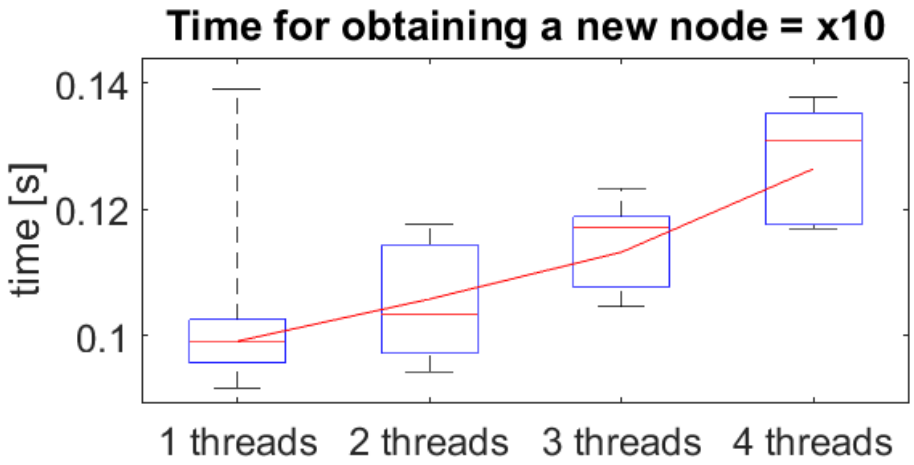
Scorrendo la lista dei nodi dell'albero = $< \underline{q}_1, \underline{q}_2, \cdots, \underline{q}_N >$ in thread paralleli producono n liste di vicini, che vengono successivamente concatenate

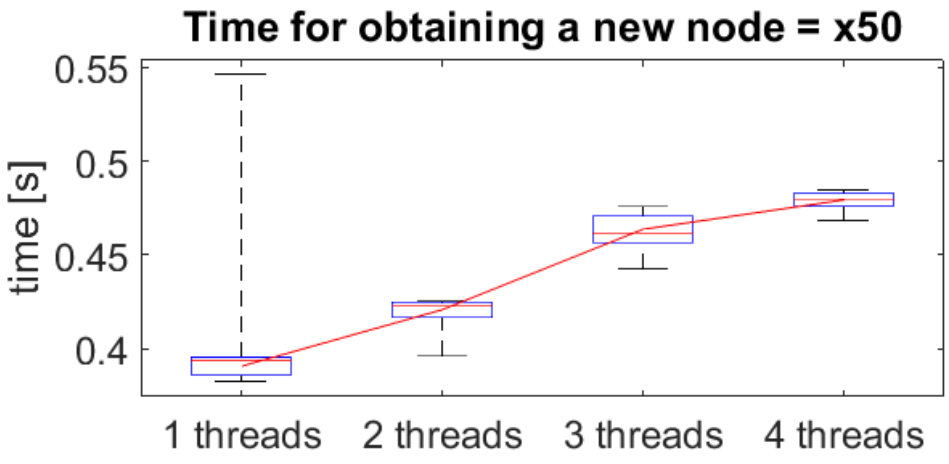


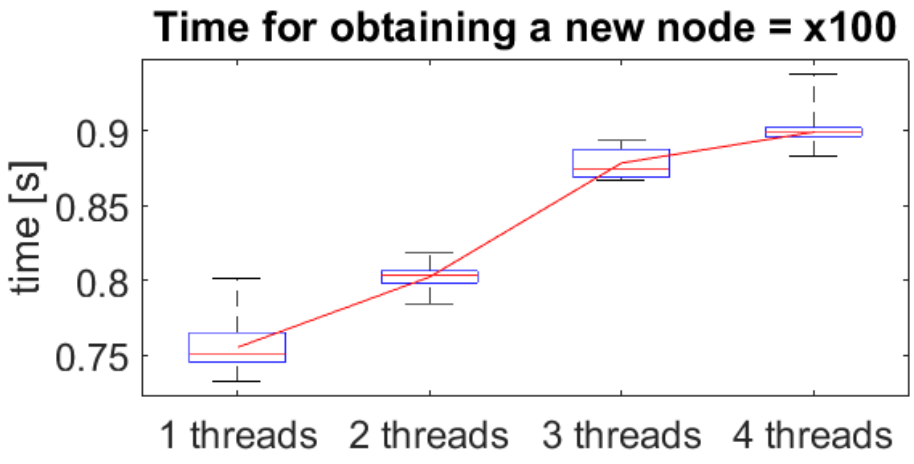
RRT-RRT*, parallel implementation

Step 1: results RRT



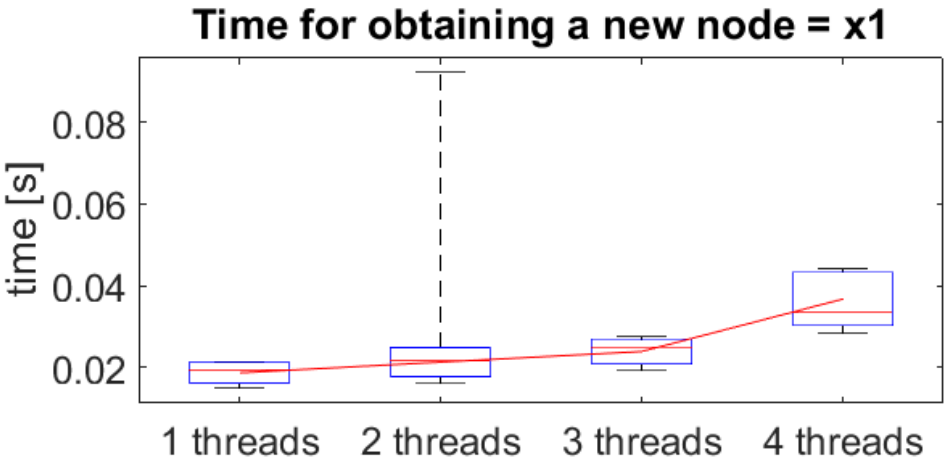


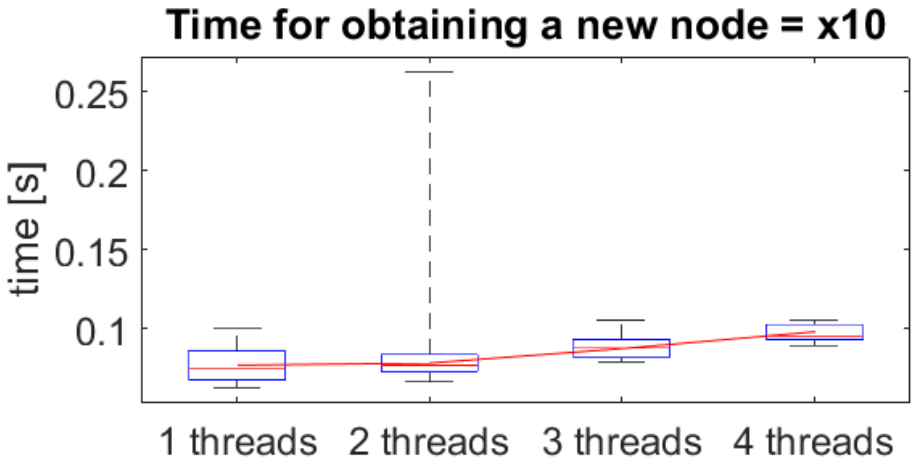


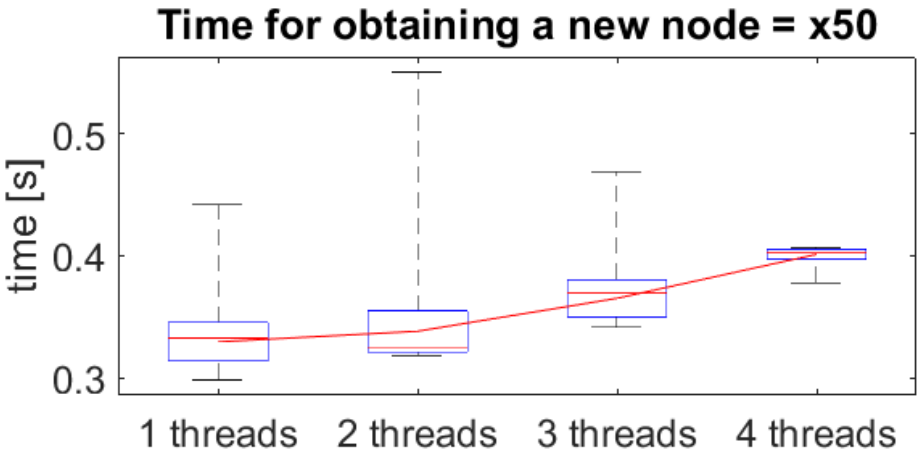


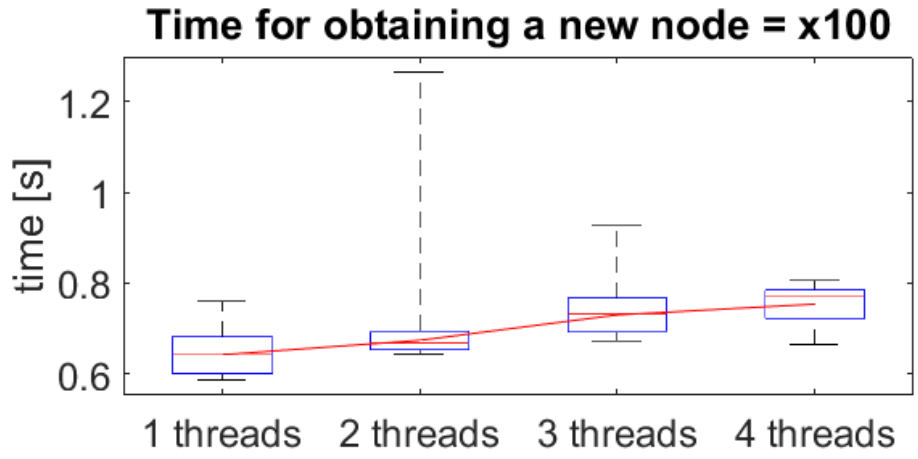
RRT-RRT*, parallel implementation

Step 1: results RRT bidirezionale





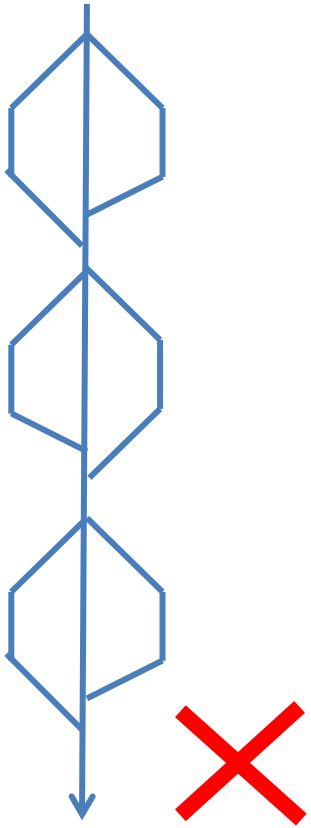


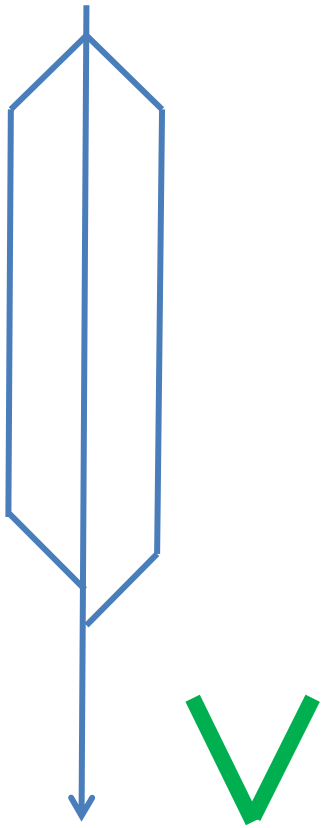


RRT, parallel implementation

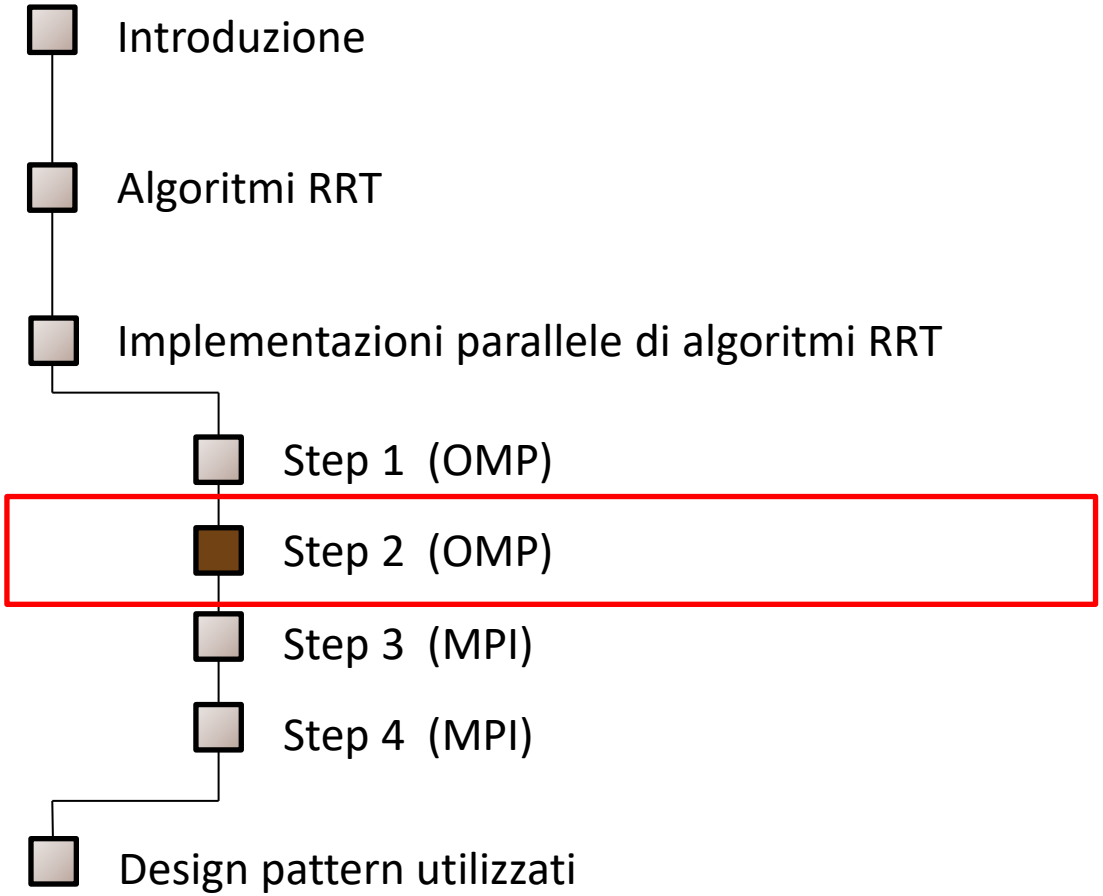
La strategia non risulta efficiente. Questo perché la sola parallelizzazione dei metodi di ricerca del vicino e quello di calcolo del $Near$ set, impongono al programma di entrare e uscire continuamente da una sezione parallela, perdendo molto tempo nell'aspettare che i vari thread terminino il lavoro.

Sarebbe invece auspicabile fare in modo che una sezione parallela permanga per tutta la durata del programma di ricerca del percorso.





Sommario


Introduzione
Algoritmi RRT
Implementazioni parallele di algoritmi RRT
Step 1 (OMP)
Step 2 (OMP)
Step 3 (MPI)
Step 4 (MPI)
Design pattern utilizzati

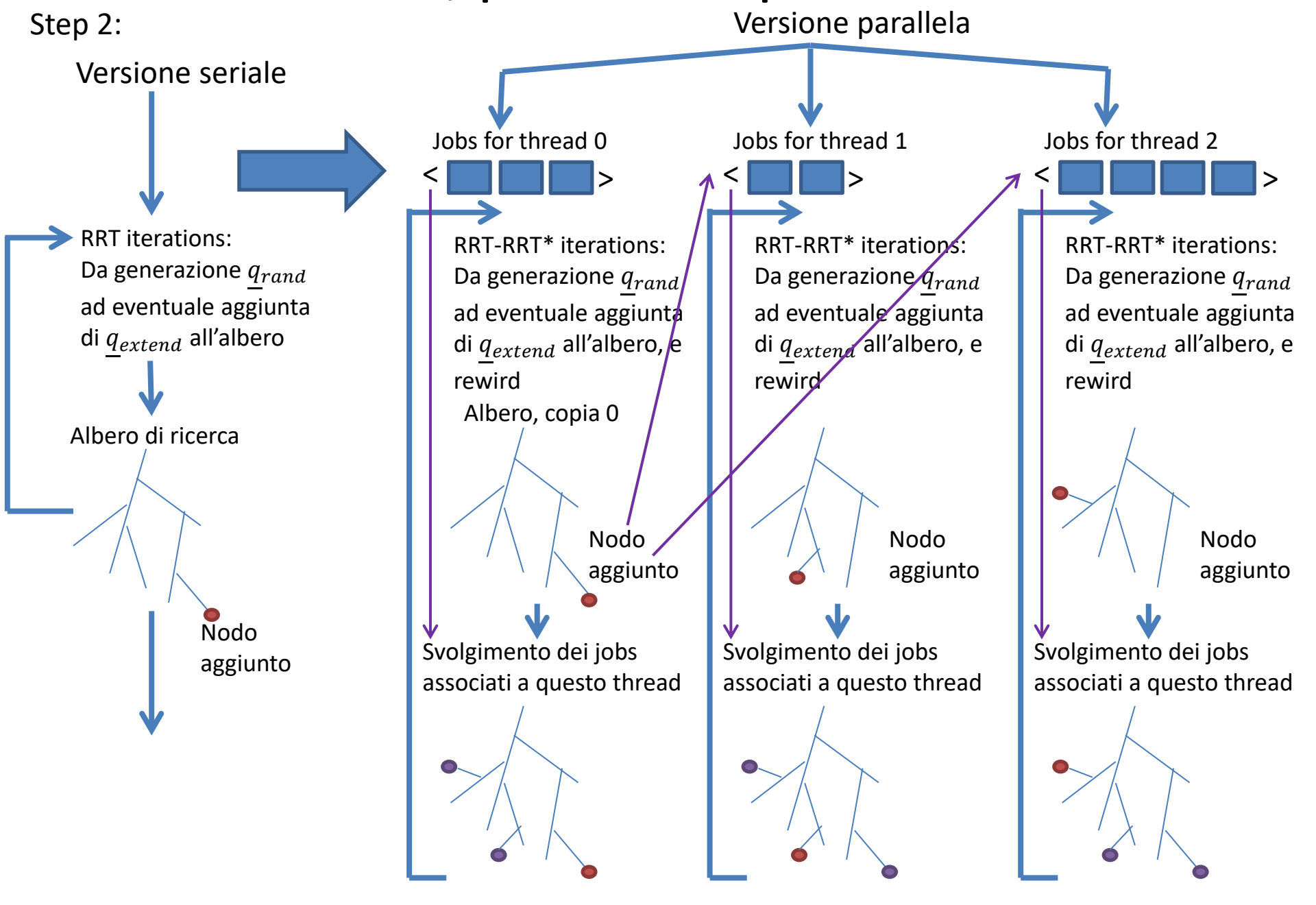
RRT-RRT*, parallel implementation

Step 2: Si parallelizza l'espansione dell'albero di ricerca, nei vari thread. Ogni thread ha visibilità di tutto l'albero, attraverso l'accesso ad una copia locale dello stesso. I vari thread, dopo aver aggiunto dei nodi alla loro copia locale, si incaricano di aggiungere delle copie degli stessi ad una lista di 'job' da fare presente negli altri thread. Ogni thread alterna le normali iterazioni dell'RRT allo svolgimento dei job provenienti dagli altri thread (inserendo le copie nel proprio albero).

Nel caso della versione RRT*, oltre ai nodi trovati, nella lista di job da fare vengono inseriti anche gli eventuali rewird eseguiti, da replicare negli alberi degli altri thread.

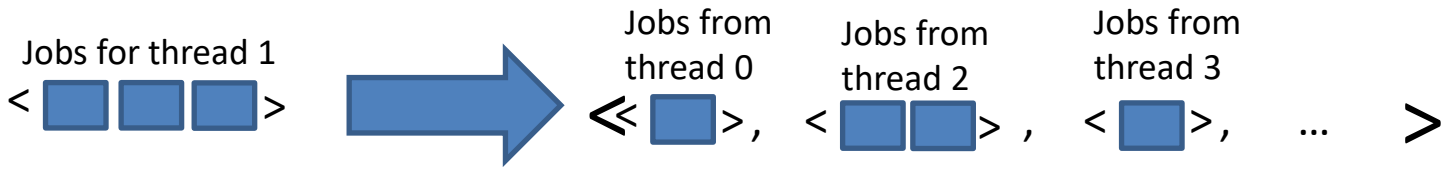
Si noti come non sia necessaria alcuna sincronizzazione dei thread, in quanto questi non devono mai modificare simultaneamente il contenuto di variabili condivise.

RRT-RRT*, parallel implementation

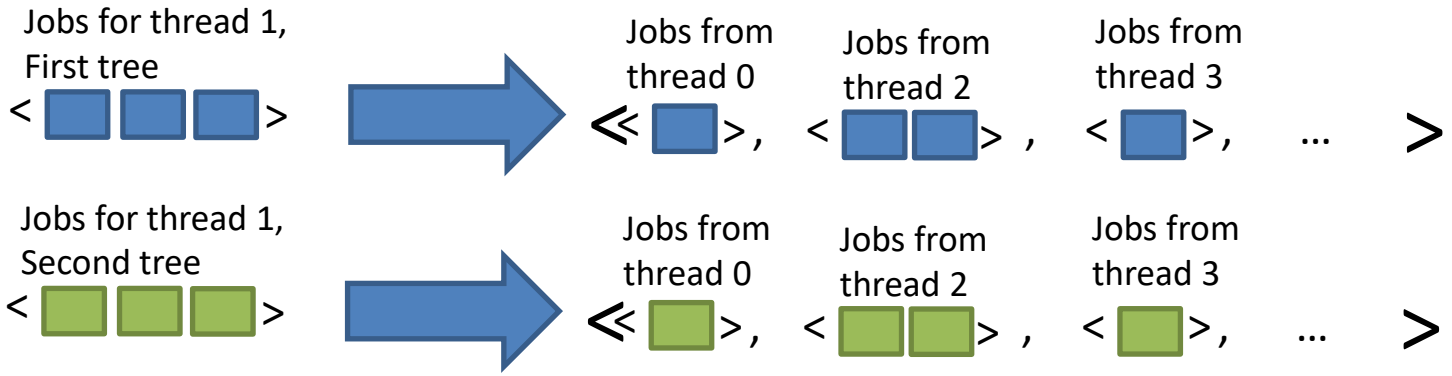


RRT-RRT*, parallel implementation

Step 2: La lista di job è in realtà un accostamento di liste, una per ogni altro thread. In questa maniera i threads non devono sincronizzarsi per aggiungere jobs, perché ognuno lo fa su una lista diversa.

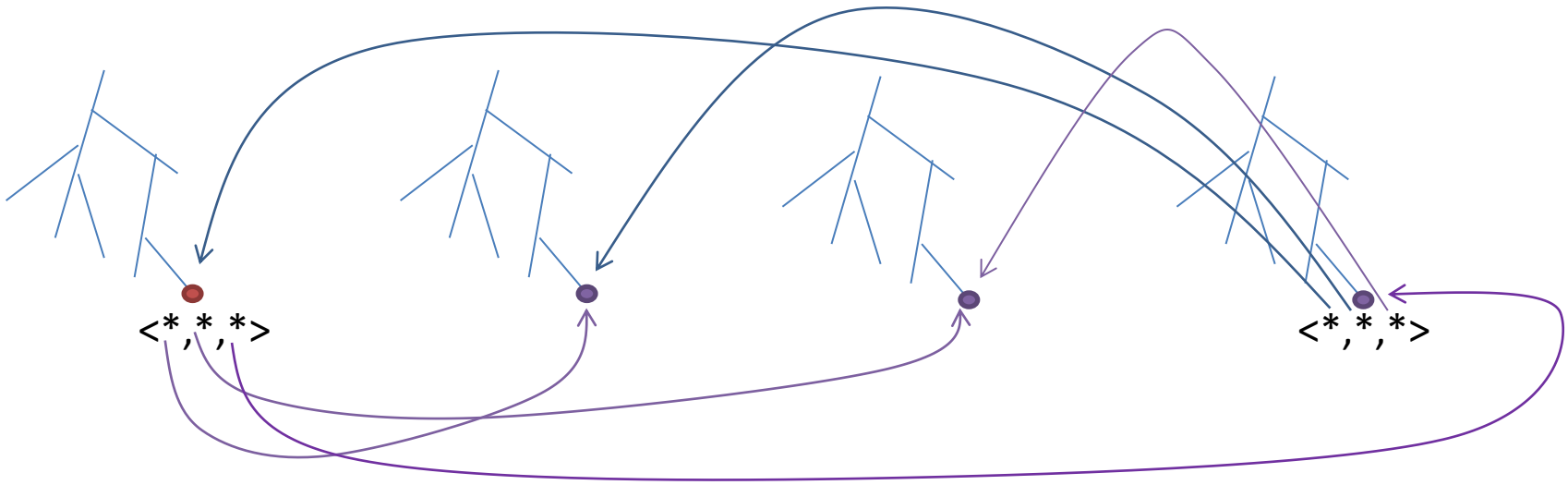


La versione di RRT bidirezionale, considera per ogni thread due liste distinte di jobs: una per ognuno dei due alberi da estendere:



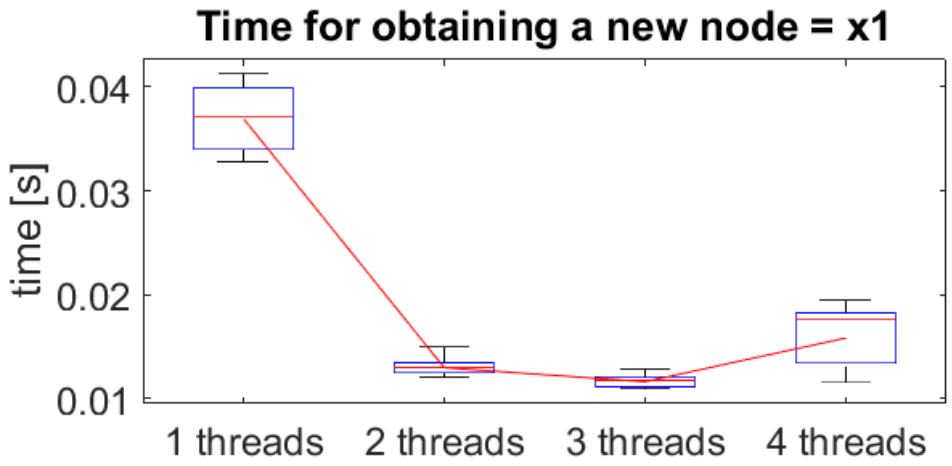
RRT*, parallel implementation

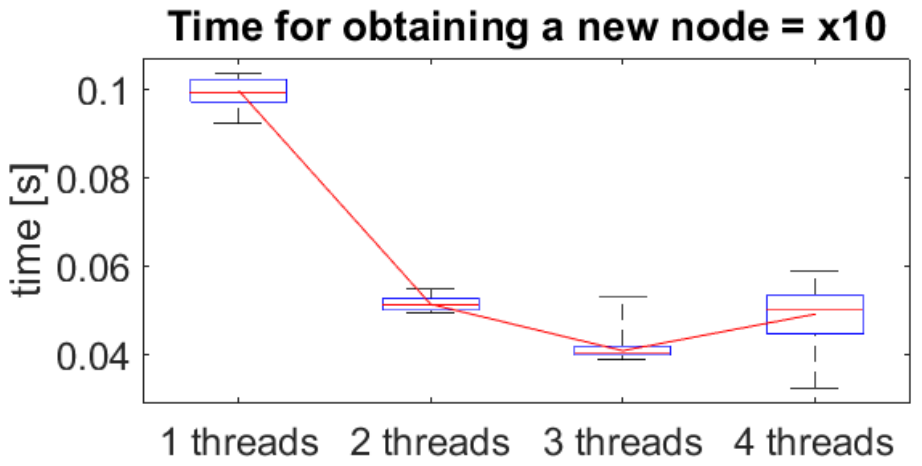
Step 2: L'implementazione RRT* richiede una piccola precisazione.
Il calcolo del costo C(q_i) associato al generico q_i richiede di scorrere la lista di predecessori, fino a giungere al nodo radice. Di conseguenza, per evitare errori, è necessario fare in modo che i nodi calcolati che vengono inseriti nelle liste di job, abbiano come padre non il nodo presente nell'albero contenuto nel thread in cui il nodo stesso è stato calcolato, ma bensì il suo corrispettivo nel thread in cui viene aggiunto come job.
Inoltre nello svolgere un certo rewird, è necessario capire qual è il clone su cui eseguirlo negli altri thread.
Queste problematiche vengono facilmente risolte, associando ad ogni nodo aggiunto (e anche alle sue copie) la lista dei suoi cloni.

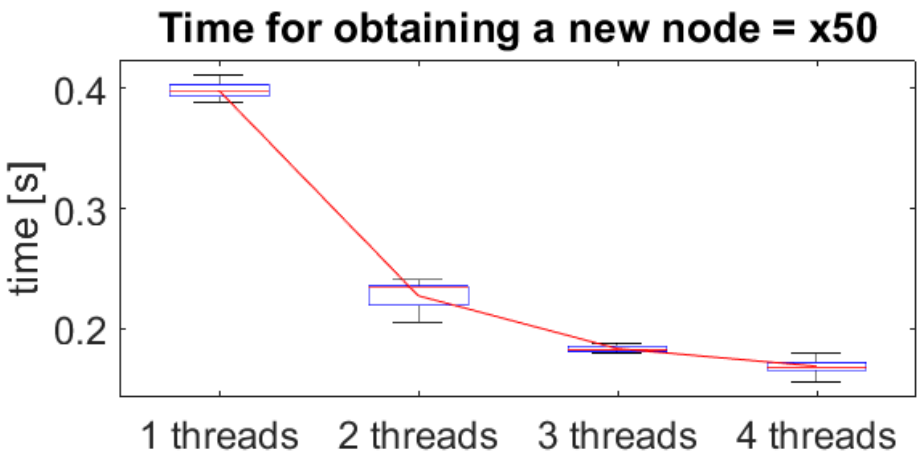


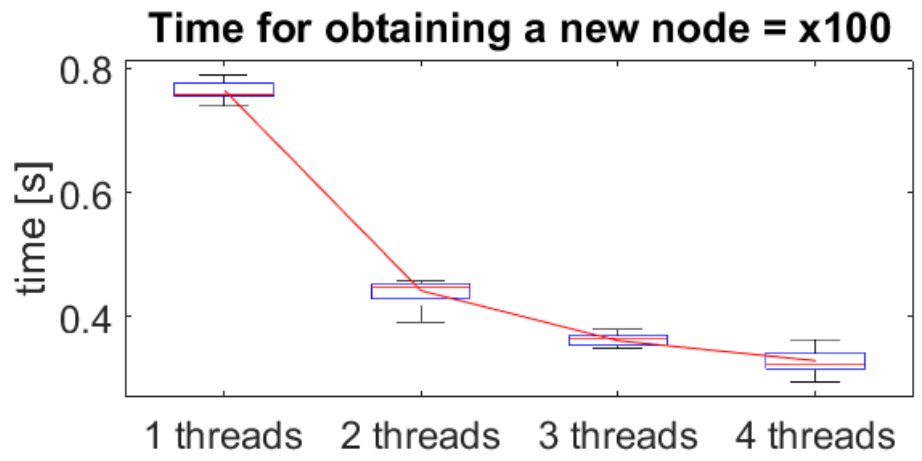
RRT-RRT*, parallel implementation

Step 2: results RRT



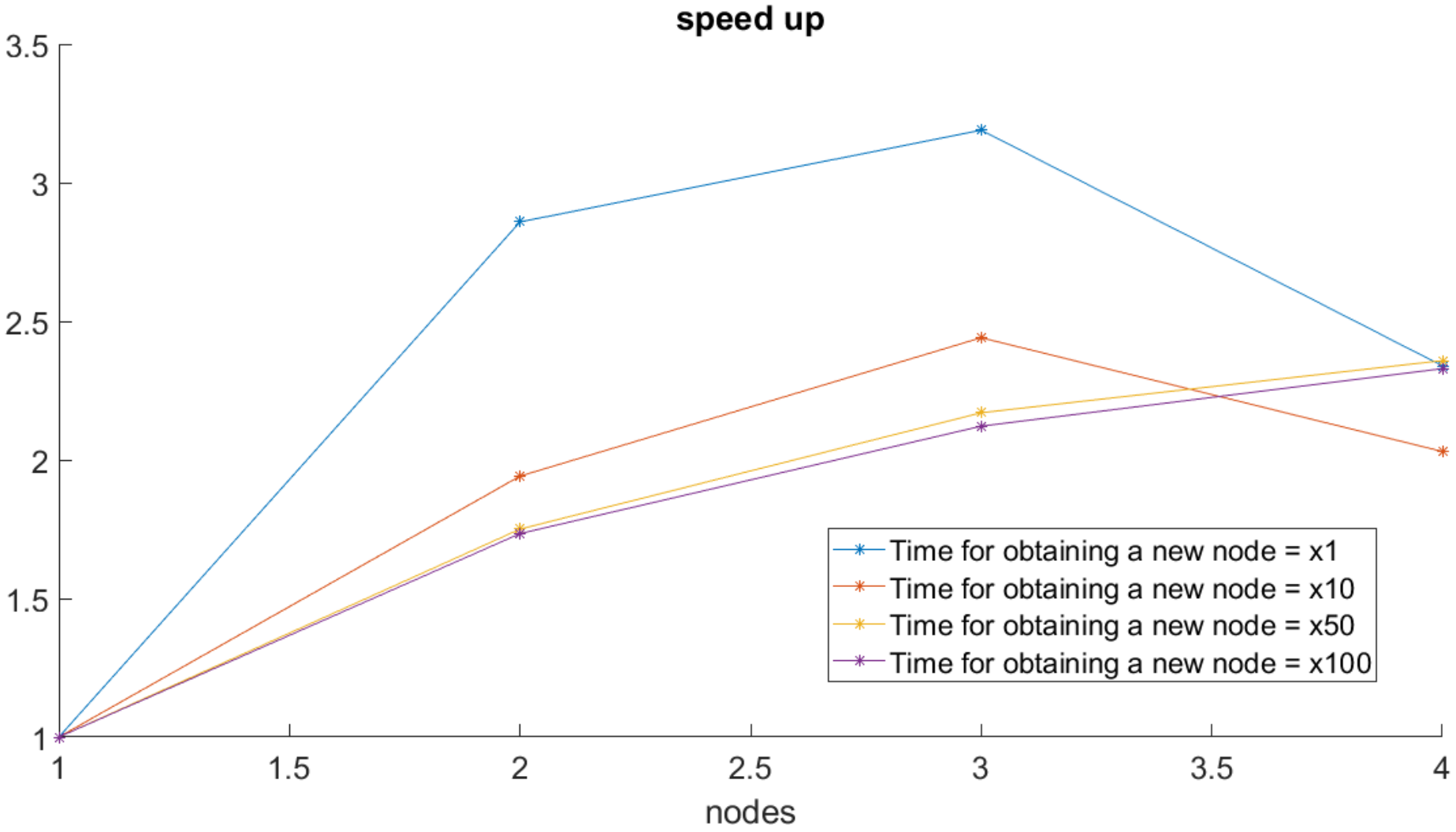






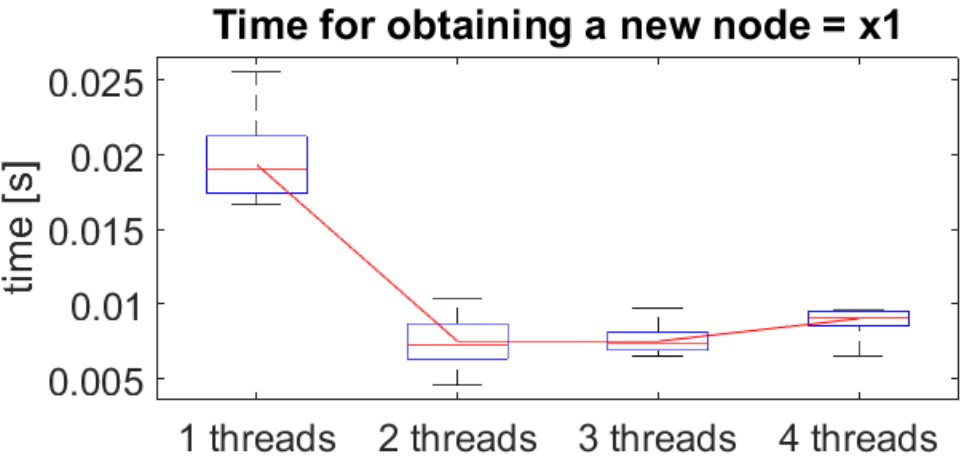
RRT-RRT*, parallel implementation

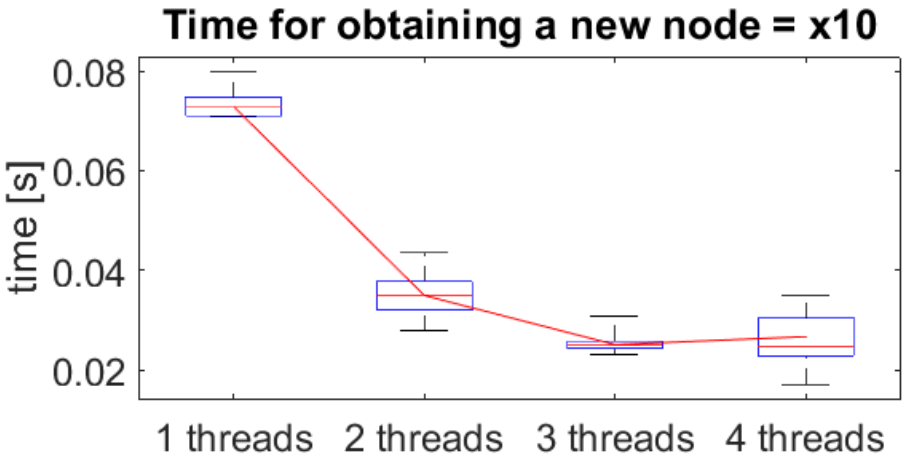
Step 2: results RRT

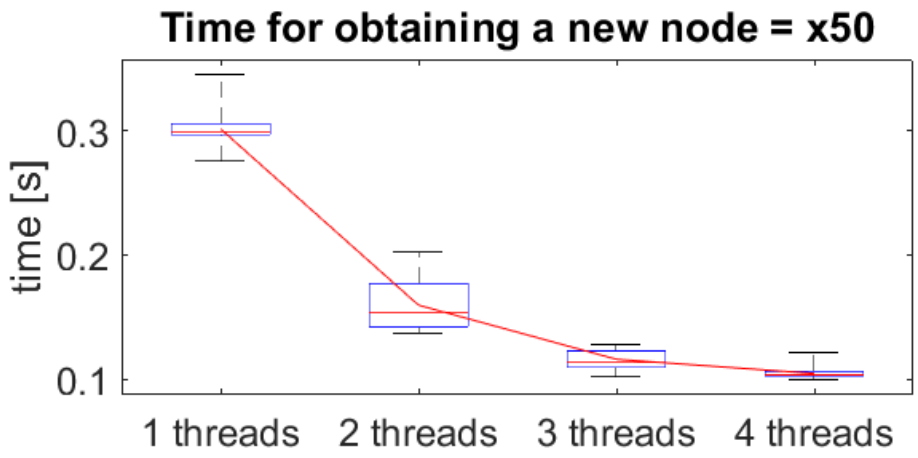


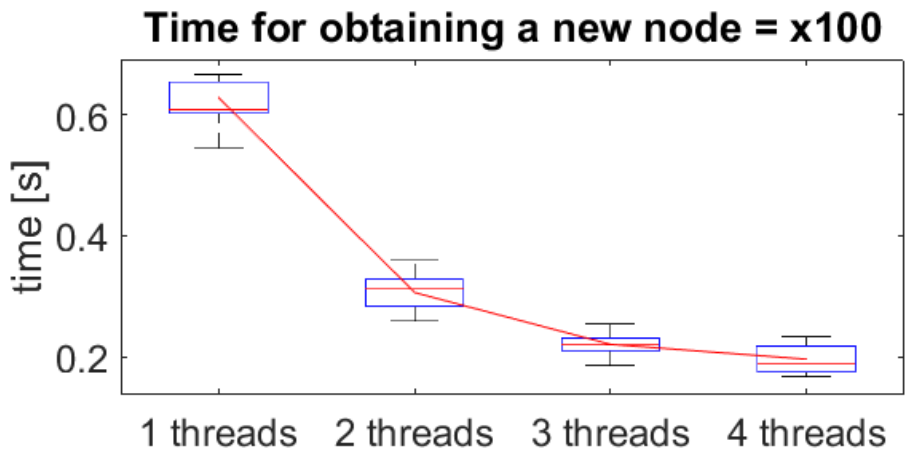
RRT-RRT*, parallel implementation

Step 2: results RRT bidirezionale



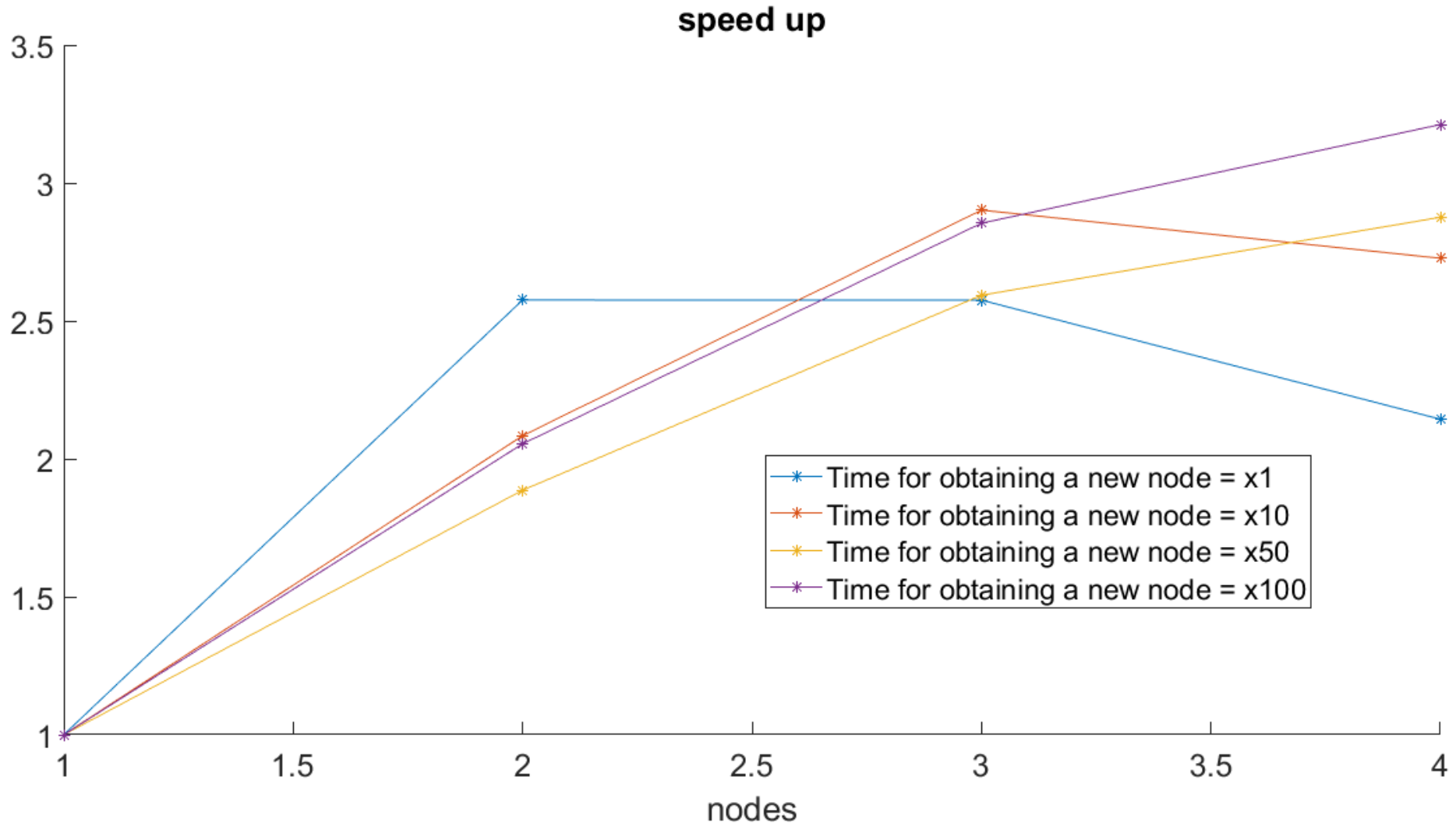






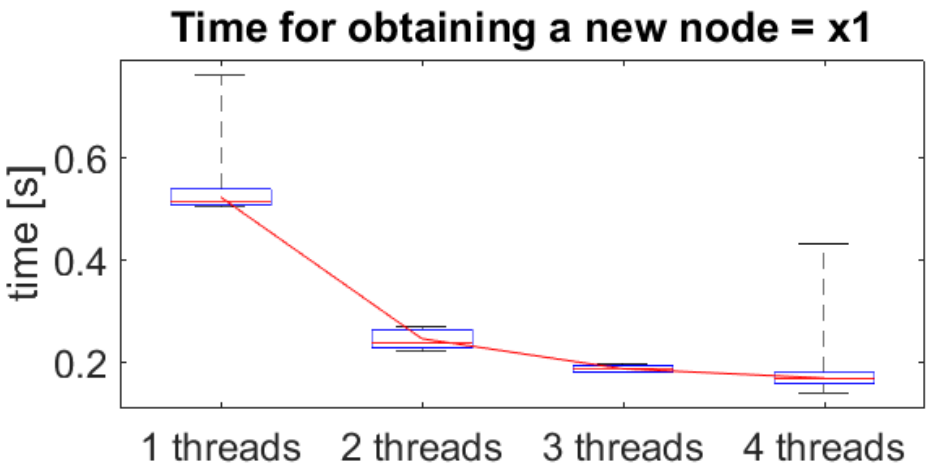
RRT-RRT*, parallel implementation

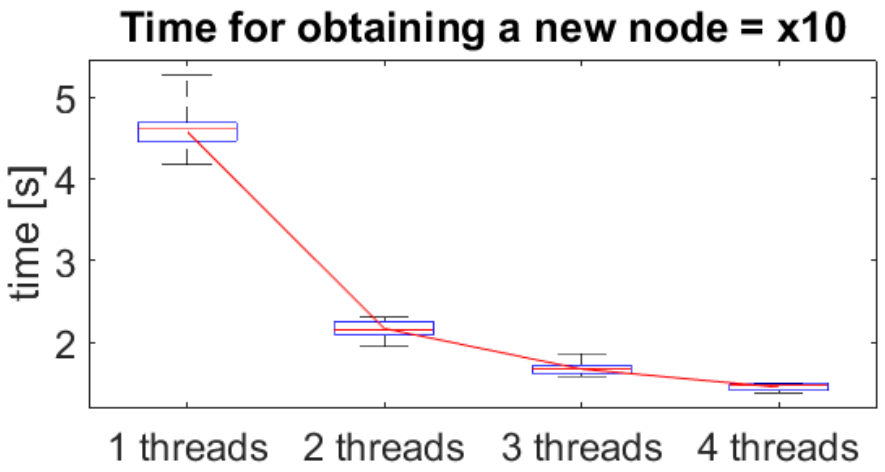
Step 2: results RRT bidirezionale

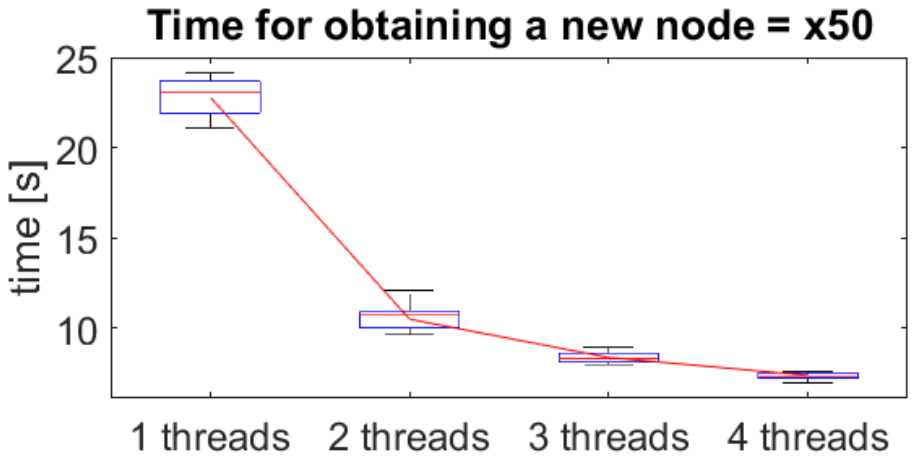


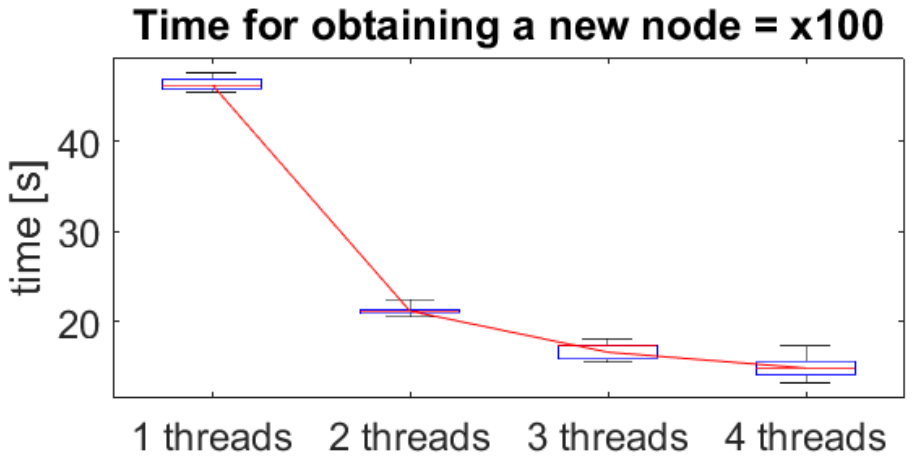
RRT-RRT*, parallel implementation

Step 2: results RRT*



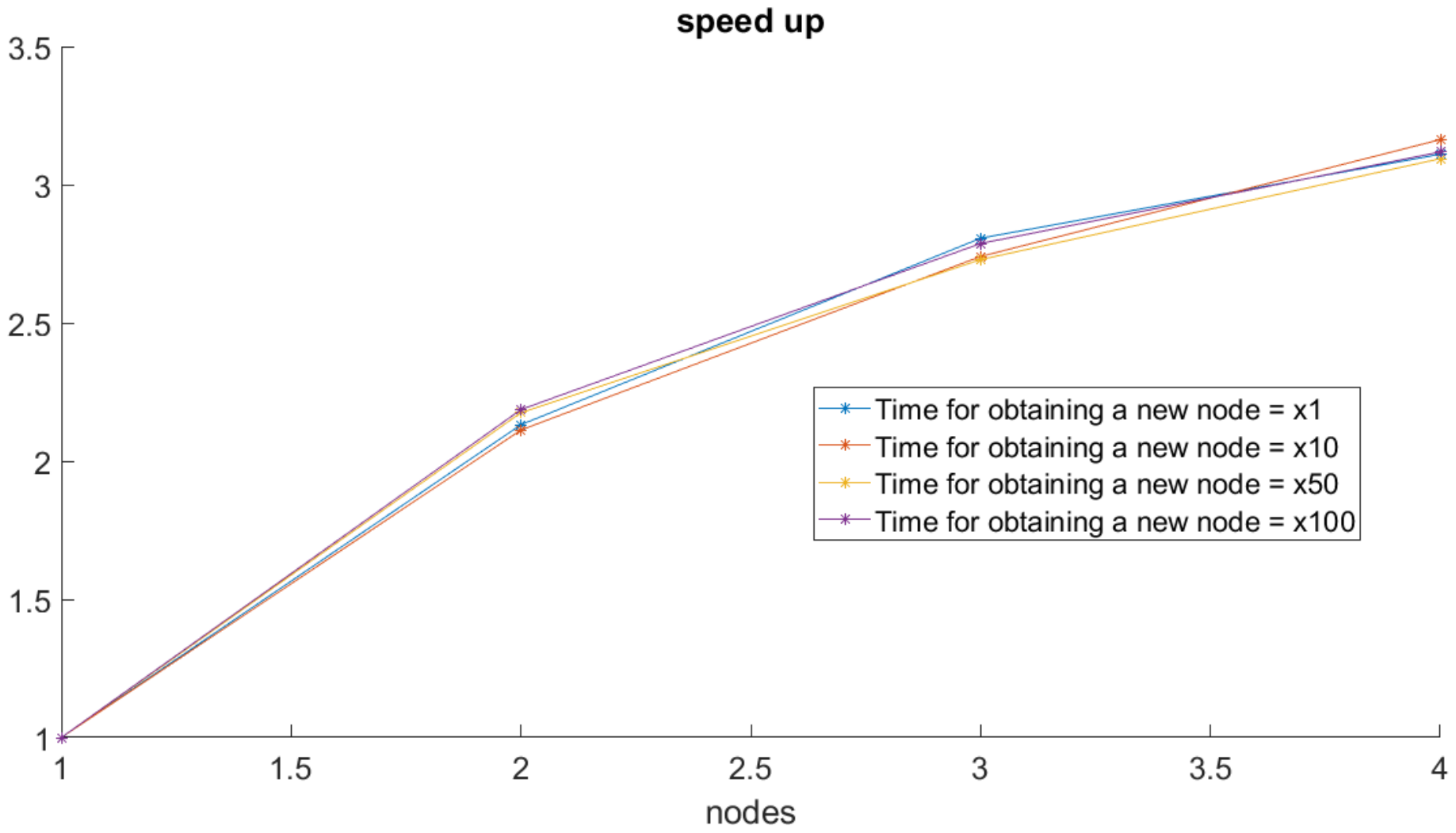




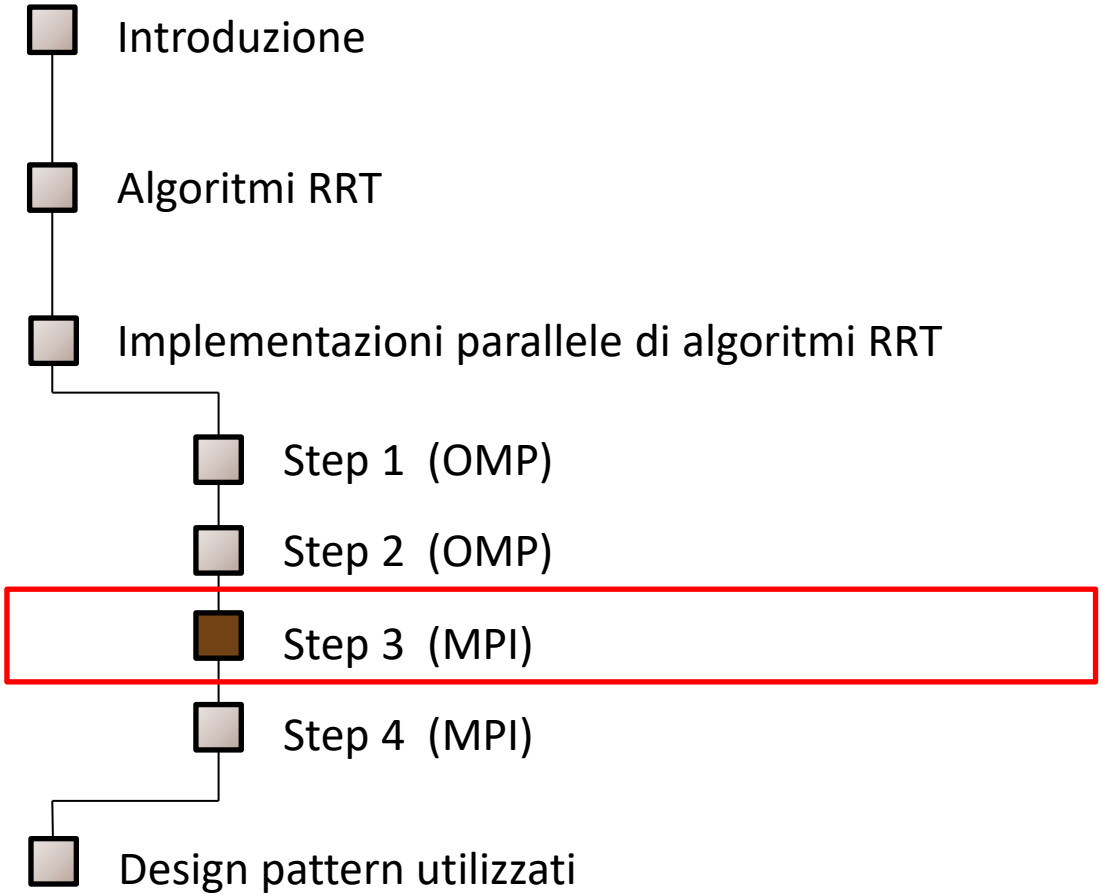


RRT-RRT*, parallel implementation

Step 2: results RRT*



Sommario


Introduzione
Algoritmi RRT
Implementazioni parallele di algoritmi RRT
Step 1 (OMP)
Step 2 (OMP)
Step 3 (MPI)
Step 4 (MPI)
Design pattern utilizzati

RRT, parallel implementation

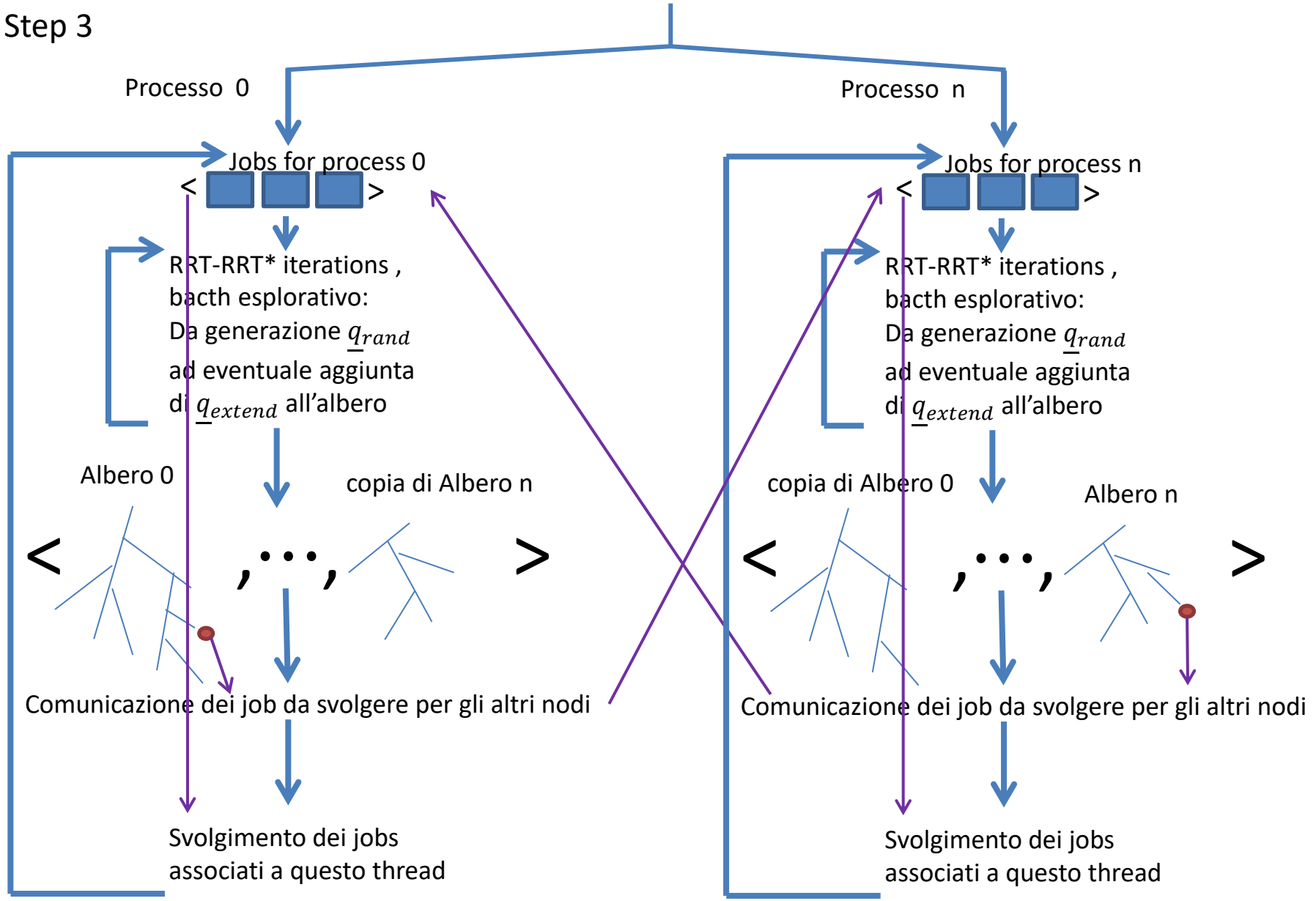
Step 3: Si può pensare di eseguire l'espansione parallela dell'albero in diversi processi invece che thread, usando openMPI. Ogni processo ha completa conoscenza dell'intero albero di ricerca, suddividendo però i nodi in varie liste separate: una di queste contiene i nodi che quel processo ha calcolato; mentre le altre liste sono copie delle liste di nodi calcolate dagli altri processi.

I vari processi eseguono parallelamente dei batch d'iterazioni dell'algoritmo RRT-RRT*, per successivamente comunicarli agli altri. La comunicazione avviene in maniera collettiva: i nodi a turno inviano agli altri, i jobs da eseguire.
Terminata la ricezione dei jobs, i vari processi provvedono parallelamente a svolgerli. Al termine di queste operazioni ha inizio un nuovo batch di esplorazioni e così via.

La suddivisione in più liste, garantisce che la posizione dei nodi all'interno delle stesse e nelle loro copie (ospitate negli altri processi) sia la stessa. In questa maniera, per comunicare il padre a cui deve essere collegato un nodo che viene mandato ad un altro processo, o tra quali nodi è stato eseguito un rewird, basta mandare un identificativo, cioè una lista di numeri interi che definiscono in quale albero si trova un certo nodo e qual è la sua posizione nello stesso.

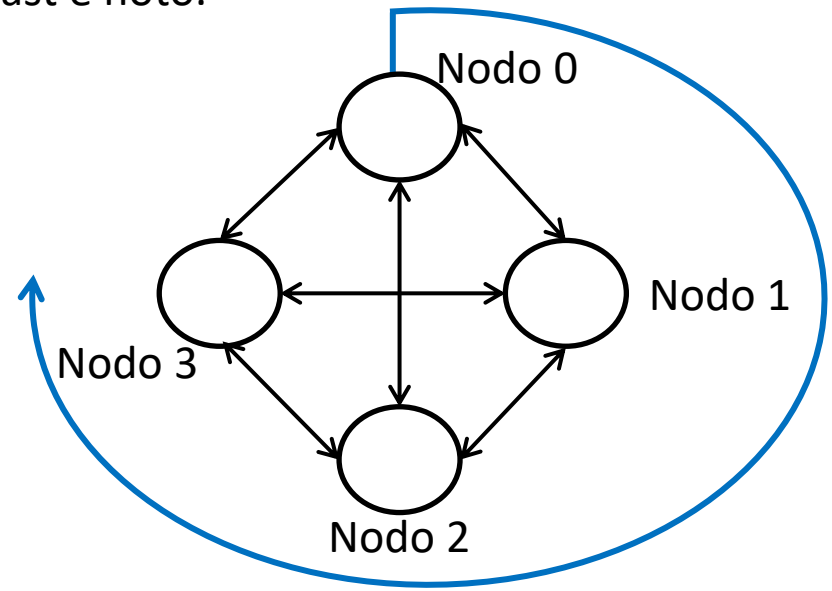
Non appena una soluzione viene trovata in uno dei processi, un segnale di arresto viene dispacciato a tutti gli altri, che terminano l'esplorazione.

RRT, parallel implementation



RRT, parallel implementation

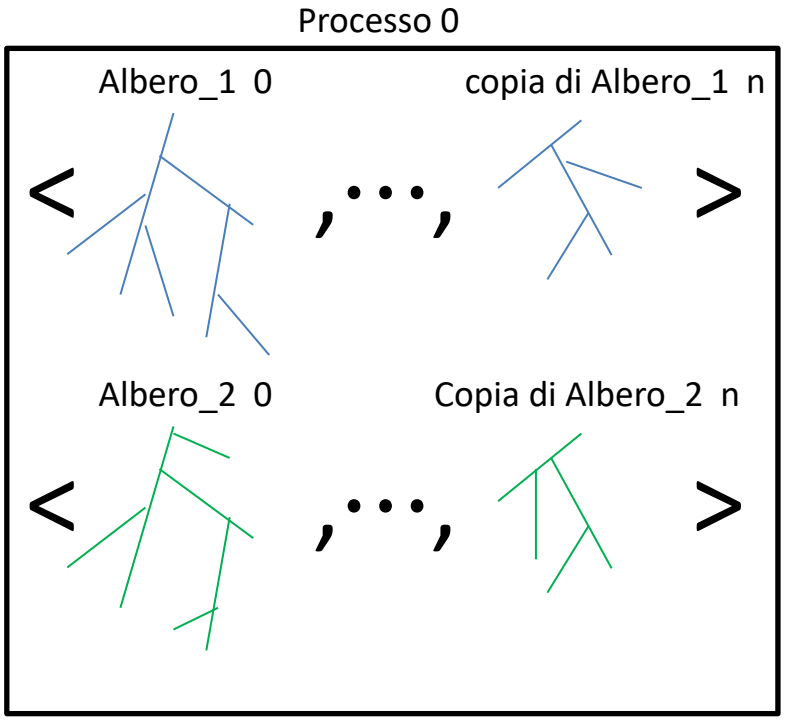
La comunicazione dei jobs, avviene attraverso chiamate a MPI_Bcast: un token viene fatto circolare tra i vari nodi che a turno comunicano i risultati. L'ordine con cui il token gira è lo stesso in tutti i processi, per cui ad ogni iterazione il root da considerare per la chiamata alla funzione MPI_Bcast è noto.

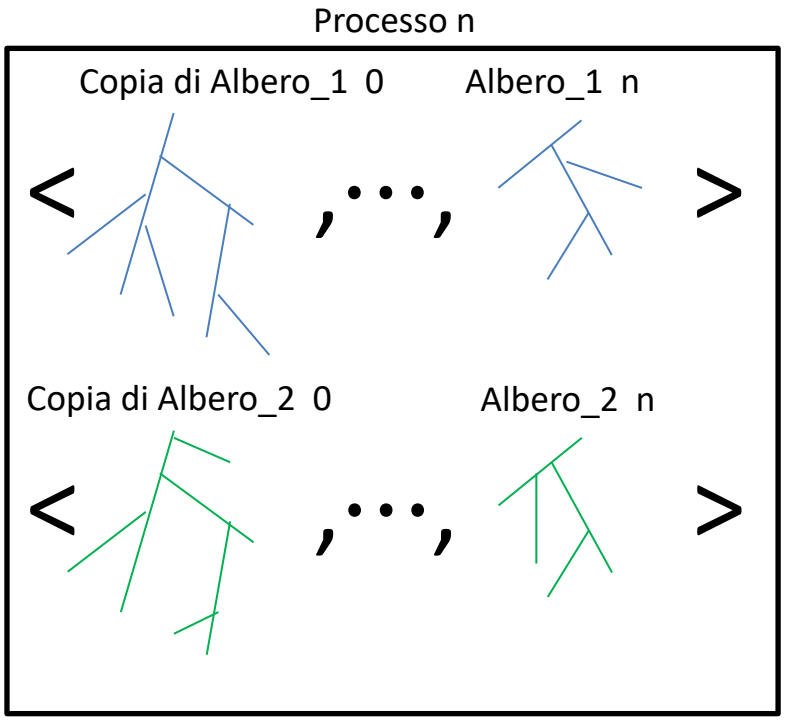


Il carico di lavoro risulta essere bilanciato, in quanto ad ogni iterazione ogni processo è impegnato nel mandare o ricevere jobs, evitando che ve ne siano alcuni in uno stato di attesa.

RRT, parallel implementation

Step 3: Nell'implementazione di RRT bidirezionale, ogni processo ospita due alberi, e le copie dei due alberi ospitati negli altri processi

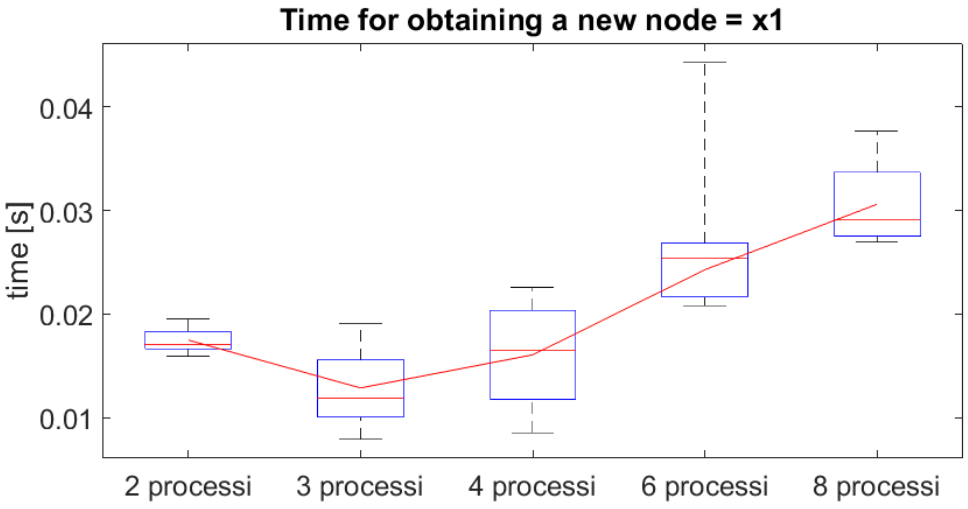


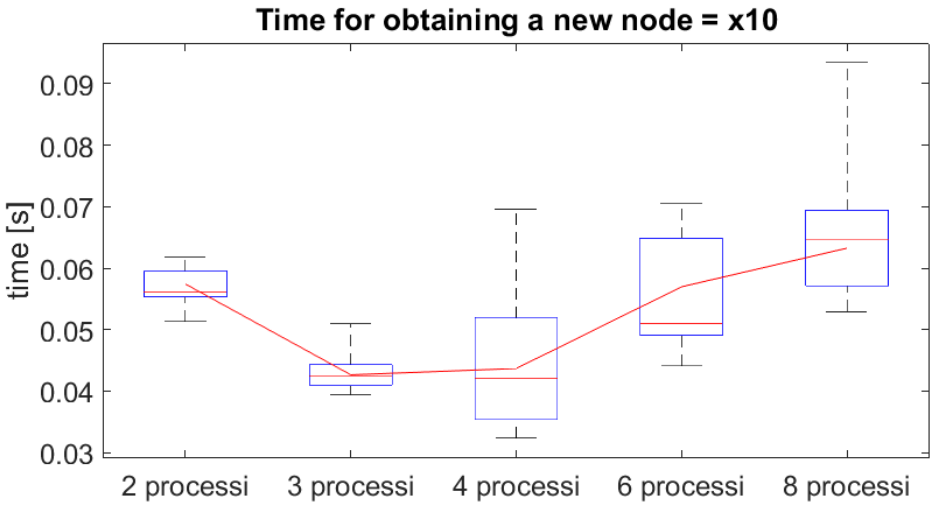


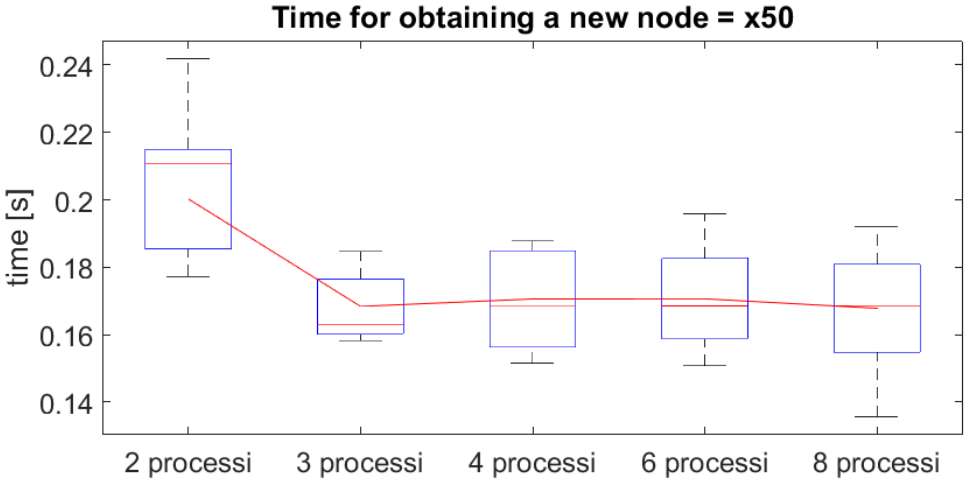
In questo caso l'identificativo associato ad ogni nodo non è più una coppia di numeri, ma bensì una terna: {# processo che lo ha generato , posizione nell'albero a cui appartiene , appartenenza all'albero che origina da $\underline{q_o}$ o a quello che origina da $\underline{q_f}$ (0/1) }.

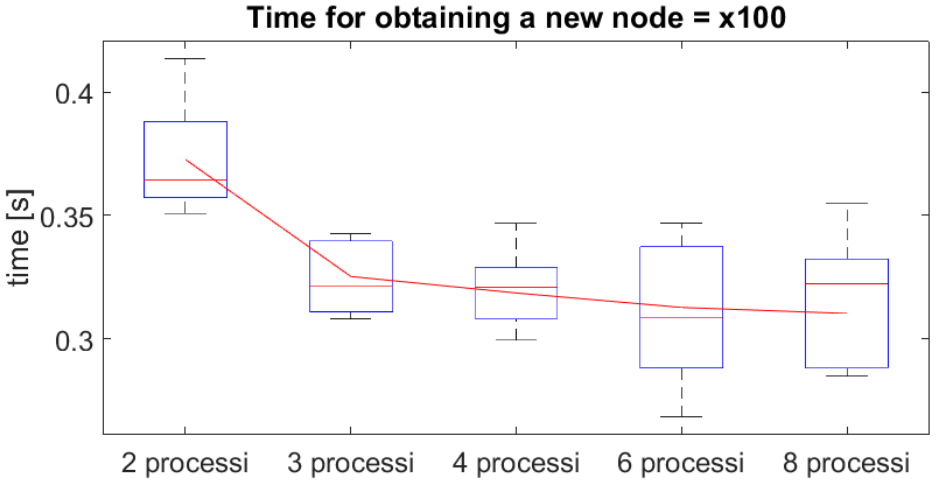
RRT-RRT*, parallel implementation

Step 3: results RRT



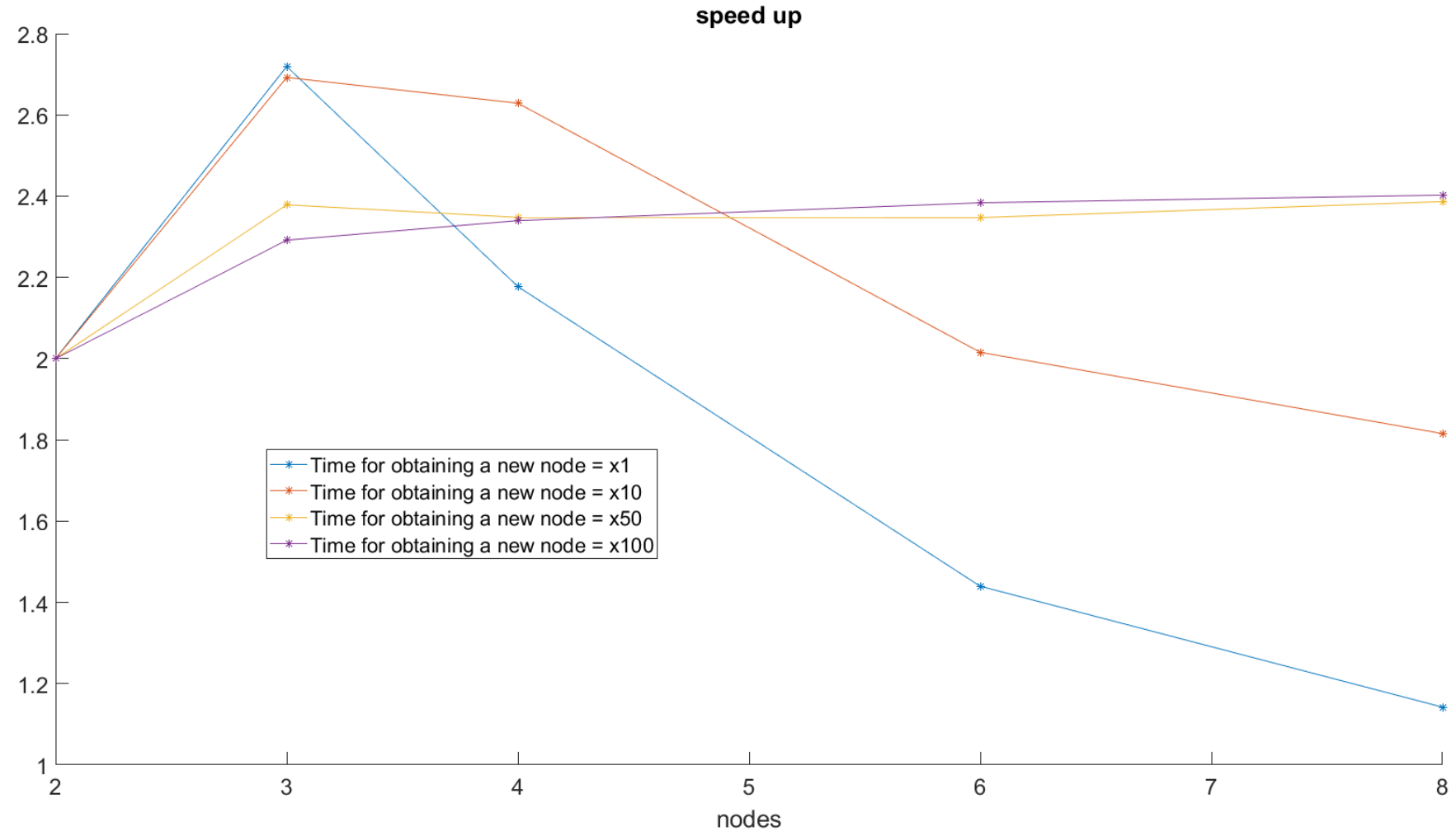






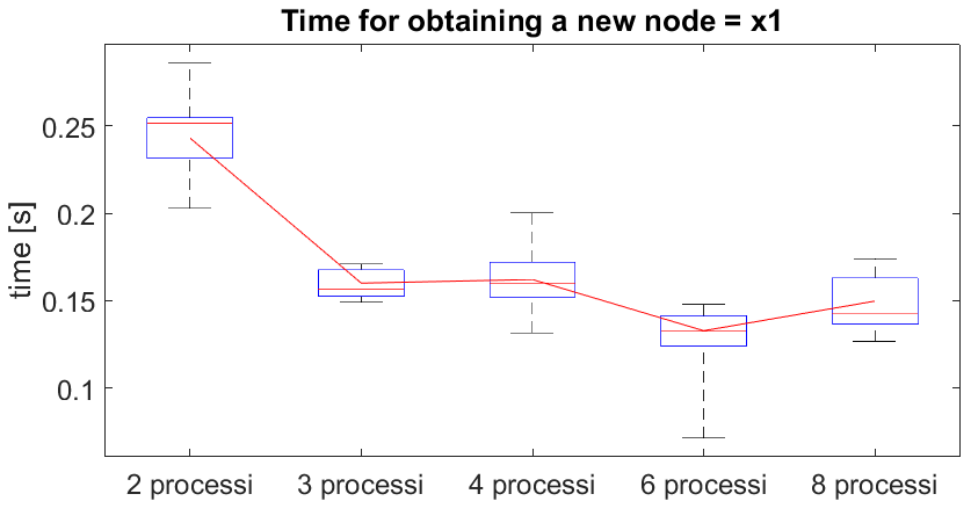
RRT-RRT*, parallel implementation

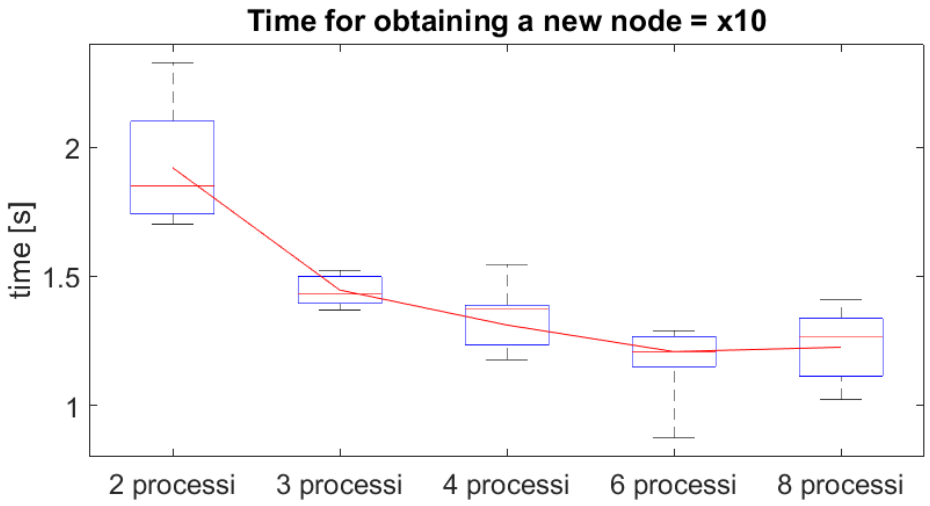
Step 3: results RRT

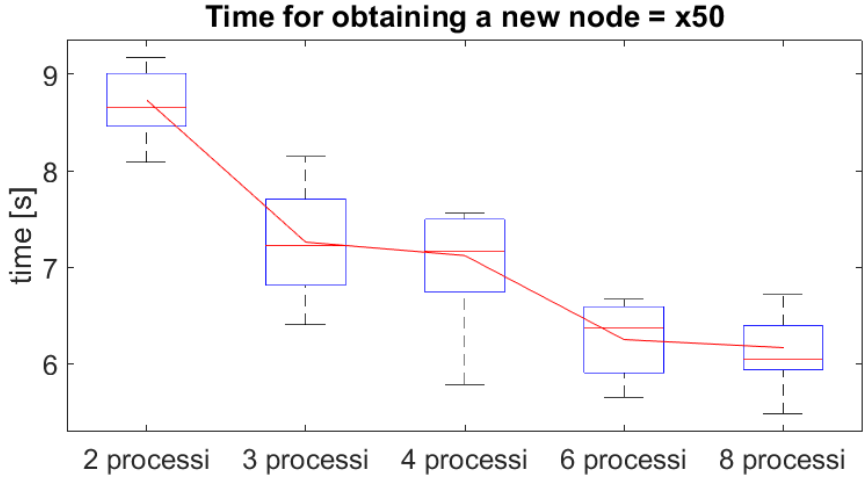


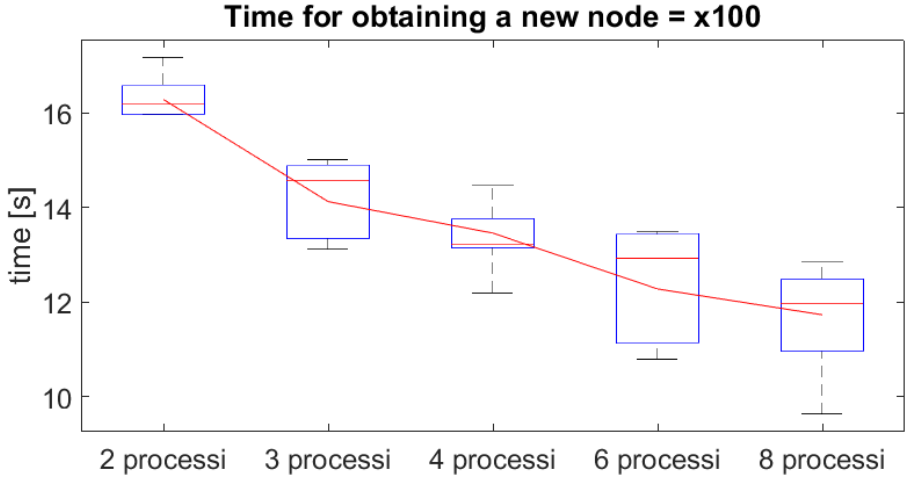
RRT-RRT*, parallel implementation

Step 3: results RRT*



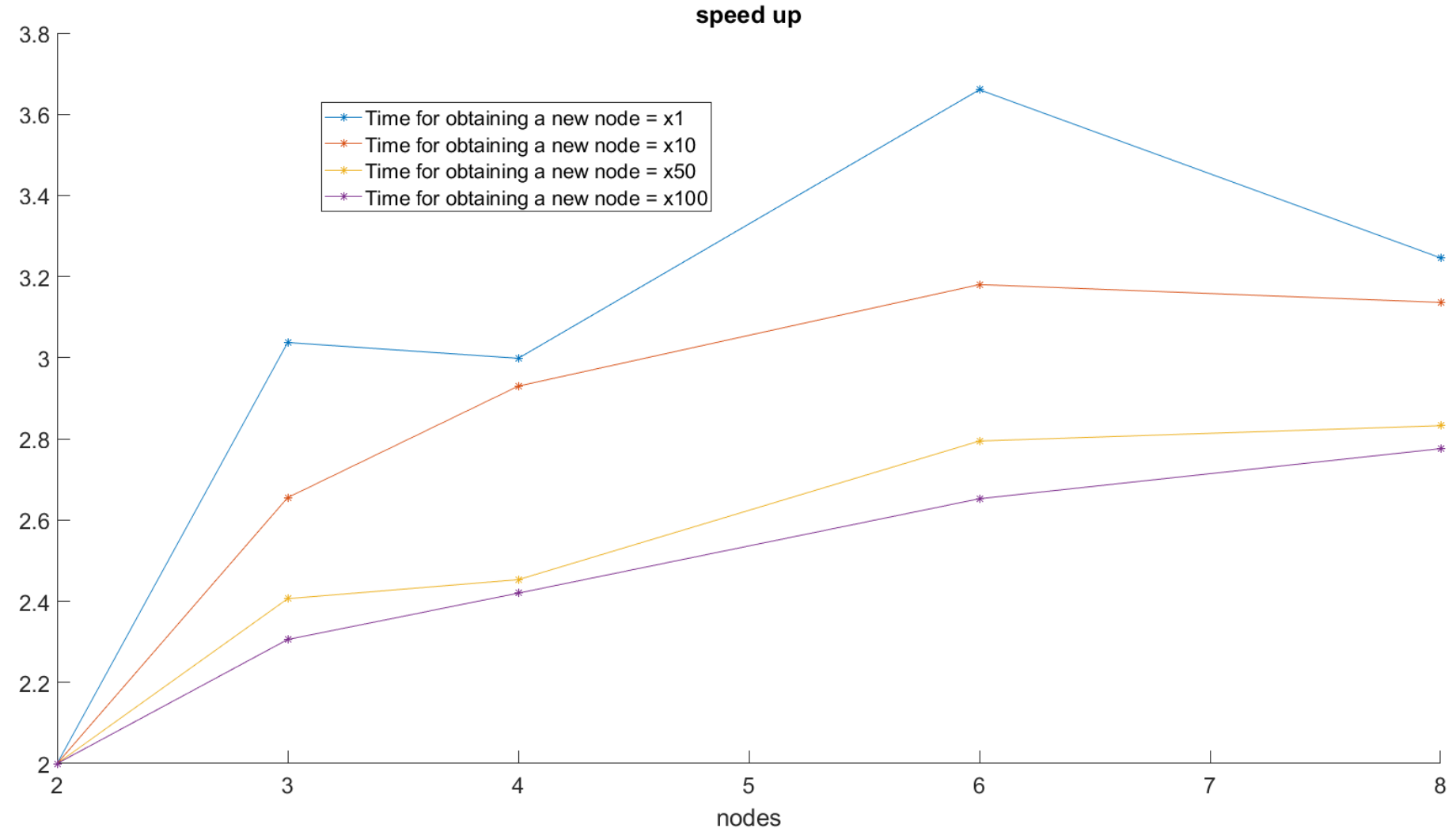




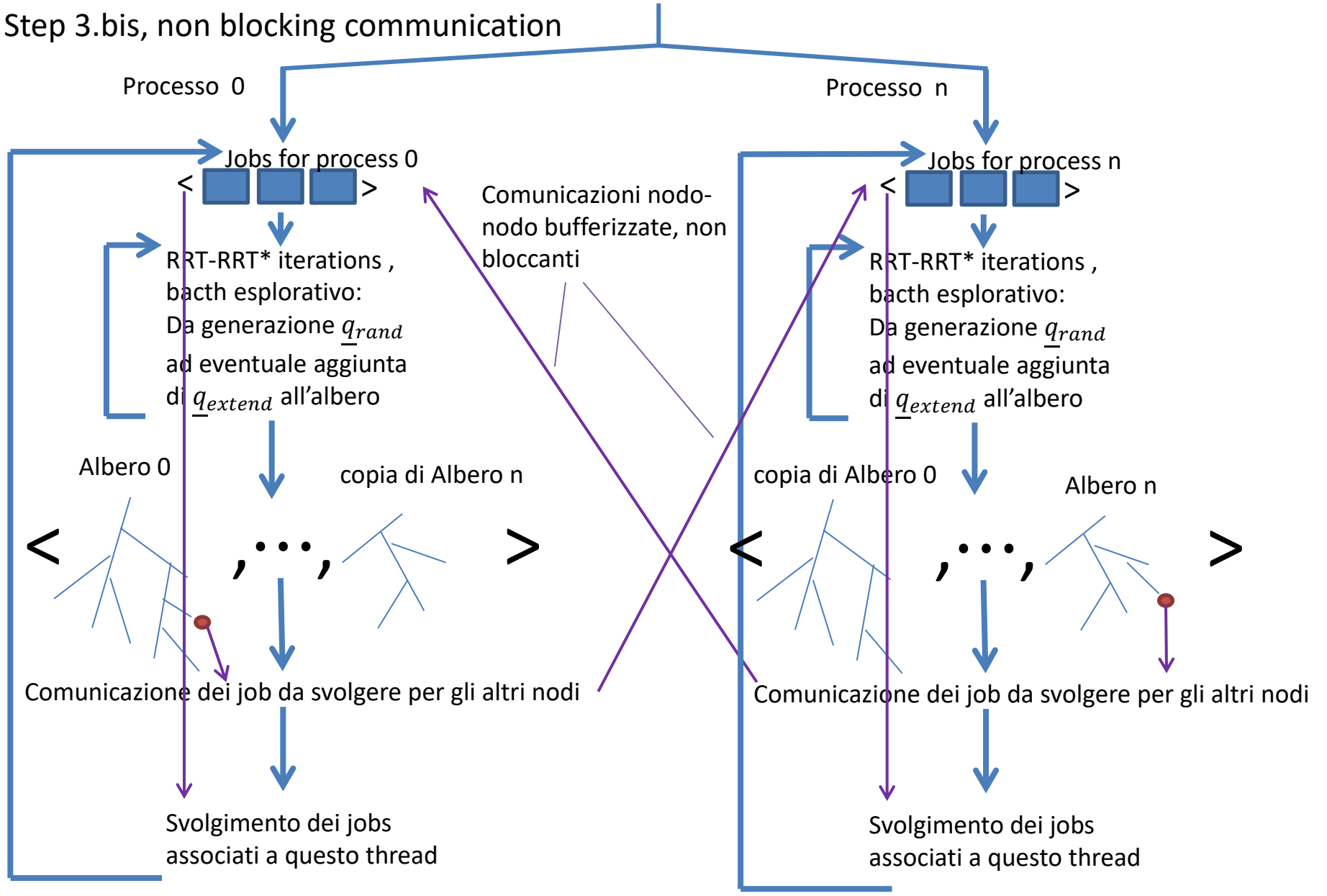


RRT-RRT*, parallel implementation

Step 3: results RRT*

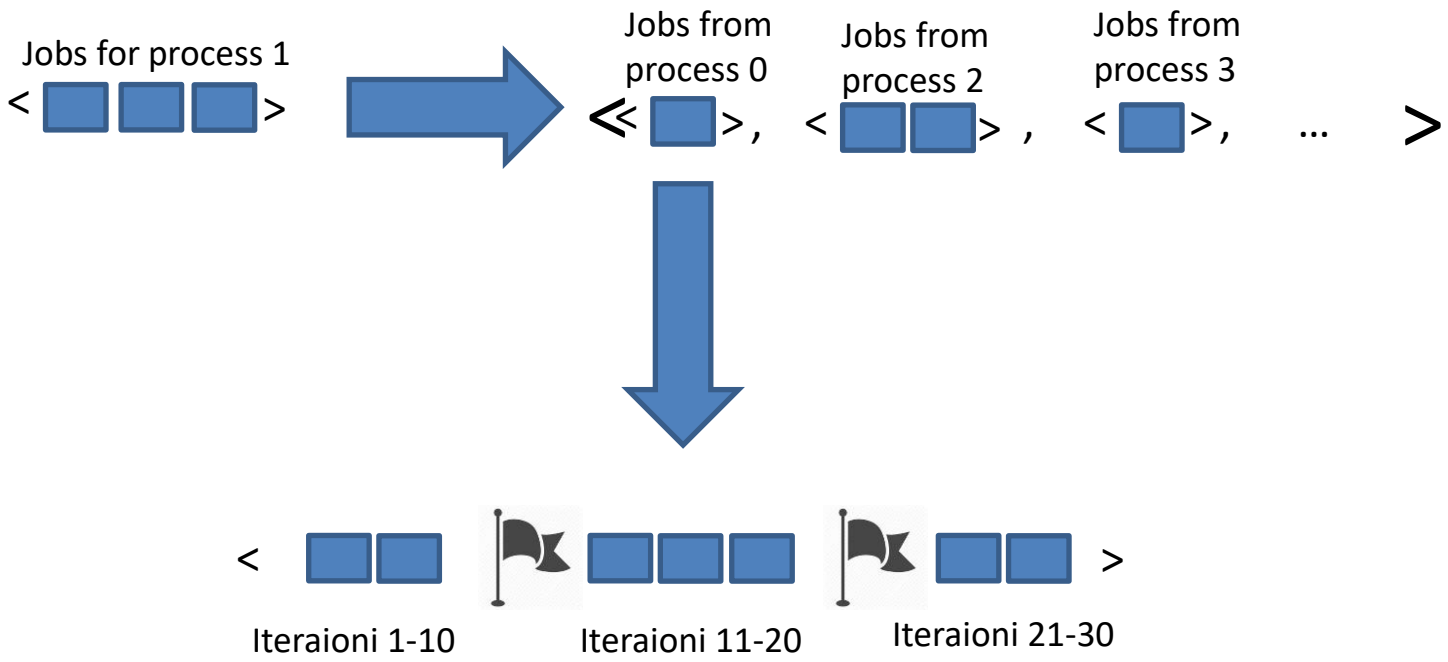


RRT, parallel implementation

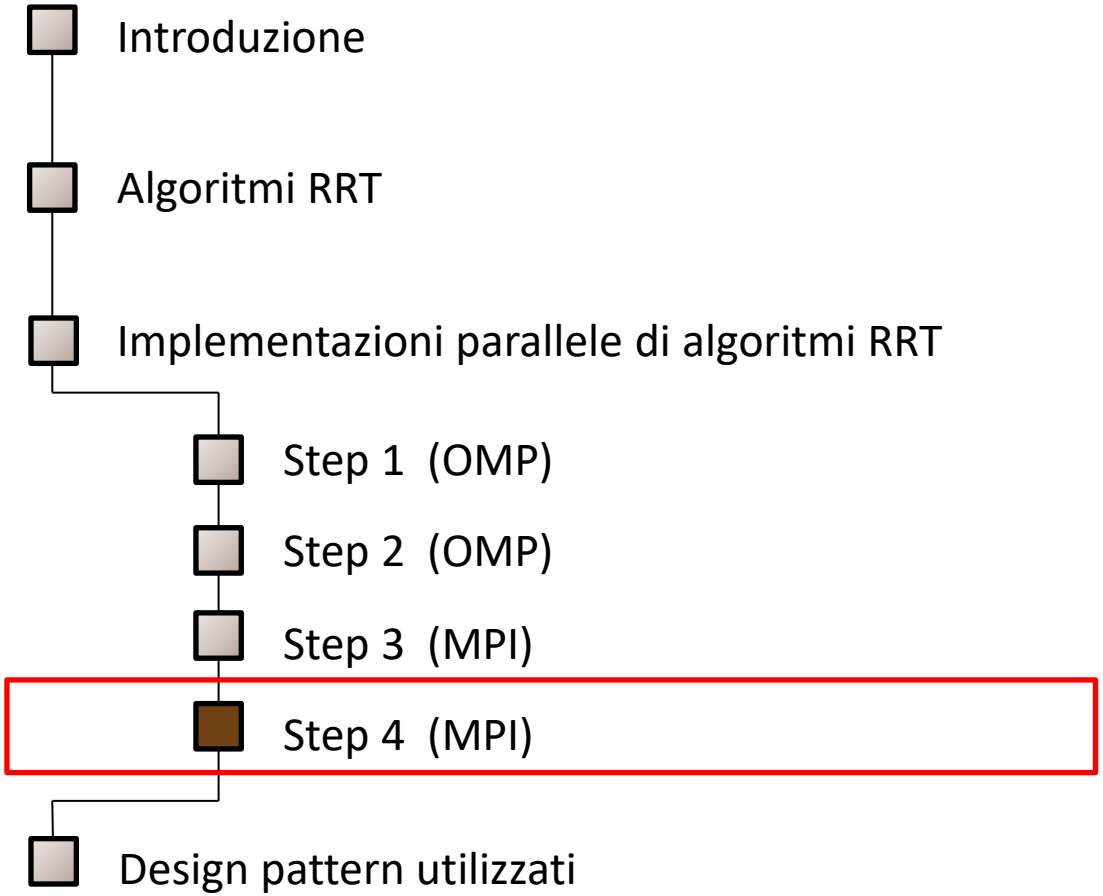


RRT-RRT*, parallel implementation

Step 3.bis: La lista di job è in realtà un accostamento di liste, una per ogni altro processo. Inoltre ogni lista associata ad un certo processo, contiene i jobs, separati però da milestone, in modo tale che sia possibile processare tutti i job provenienti dagli altri processi e relativi ad un certo specifico batch di iterazioni in maniera separata



Sommario


Introduzione
Algoritmi RRT
Implementazioni parallele di algoritmi RRT
Step 1 (OMP)
Step 2 (OMP)
Step 3 (MPI)
Step 4 (MPI)
Design pattern utilizzati

RRT, parallel implementation

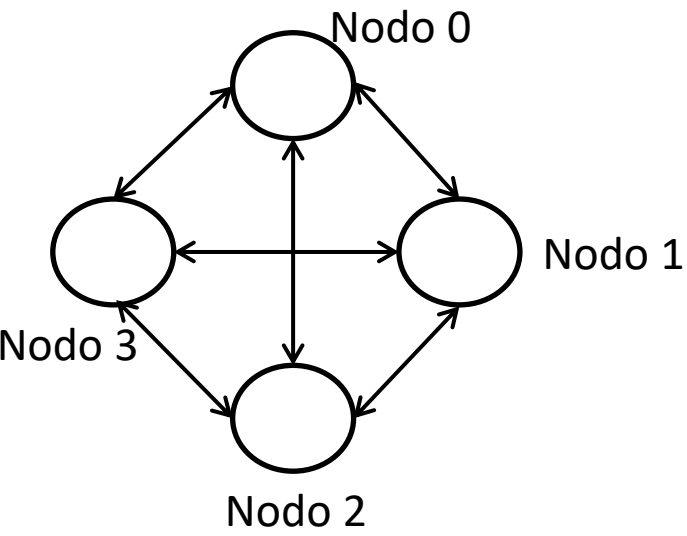
Step 4: L'espansione dell'albero può essere svolta parallelamente da nodi slaves, coordinati da un unico nodo master.
E' il nodo master, l'unico ad avere completa conoscenza dell'albero di ricerca, e di conseguenza è lui che si incarica di generare nuove pose randomiche e di trovare i realtivi nodi vicini. Tali vicini vengono poi dispacciati agli slaves (ognuno ne riceve uno diverso), che cominciano un'esplorazione locale per un certo numero di iterazioni a partire da queste nuove radici.

Terminata l'espansione, gli slaves comunicano al master i nodi trovati. Il master provvede quindi ad aggiungere tali nodi all'albero di ricerca.

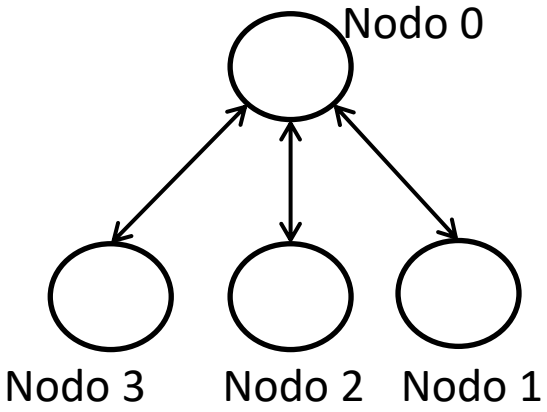
Quando una soluzione viene trovata, il nodo master, manda un comando di arresto ai vari slaves.

RRT, parallel implementation

Step 3



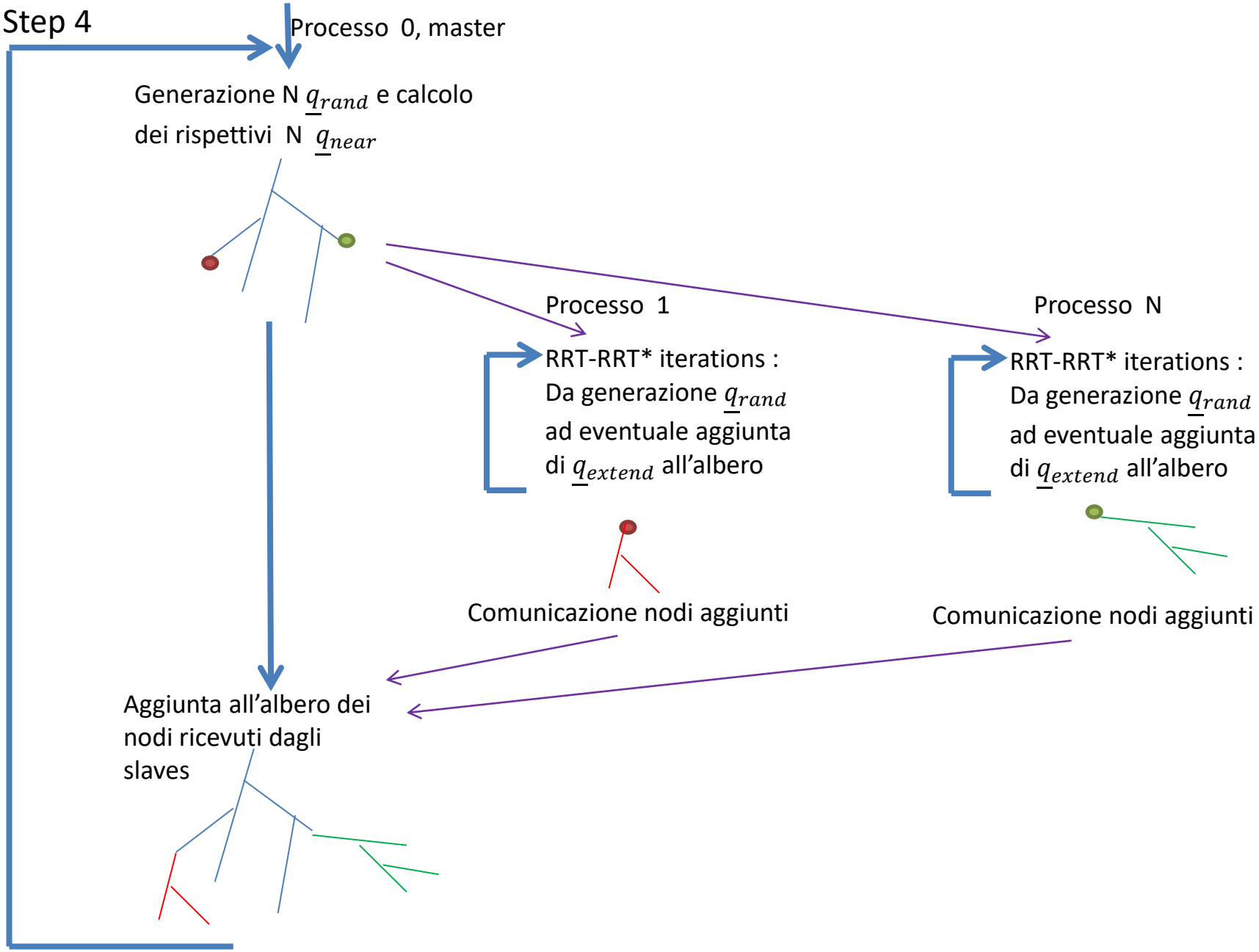
Step 4



Il tempo speso per comunicare i risultati calcolati ad altri processi è decisamente inferiore nel caso dello Step 4.

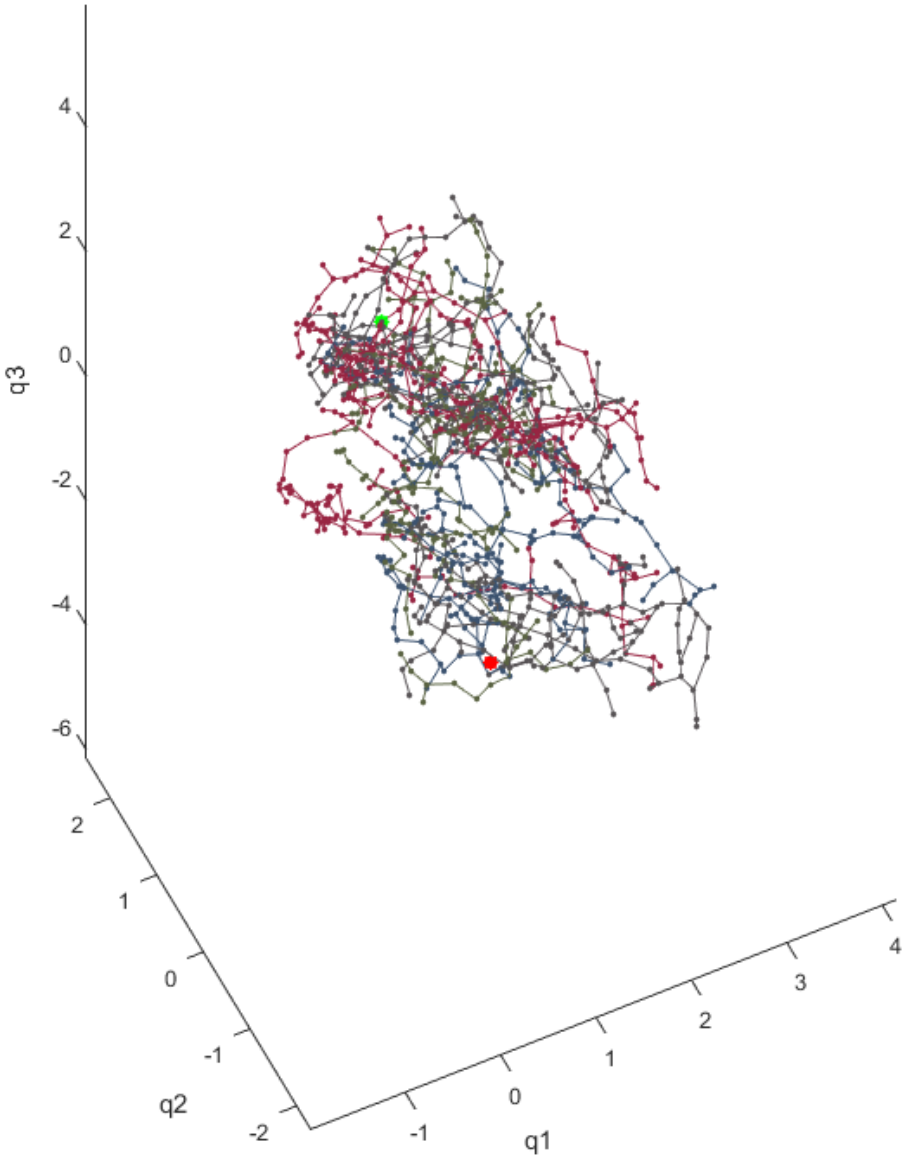
Tuttavia l'esplorazione che svolgono gli slaves, non è quella canonica prevista dall'RRT, in quanto avviene localmente. Se il numero di iterazioni associate all'esplorazione da parte degli slaves è non troppo grande, non si manifestano problemi nella pratica.

RRT, parallel implementation



RRT, parallel implementation

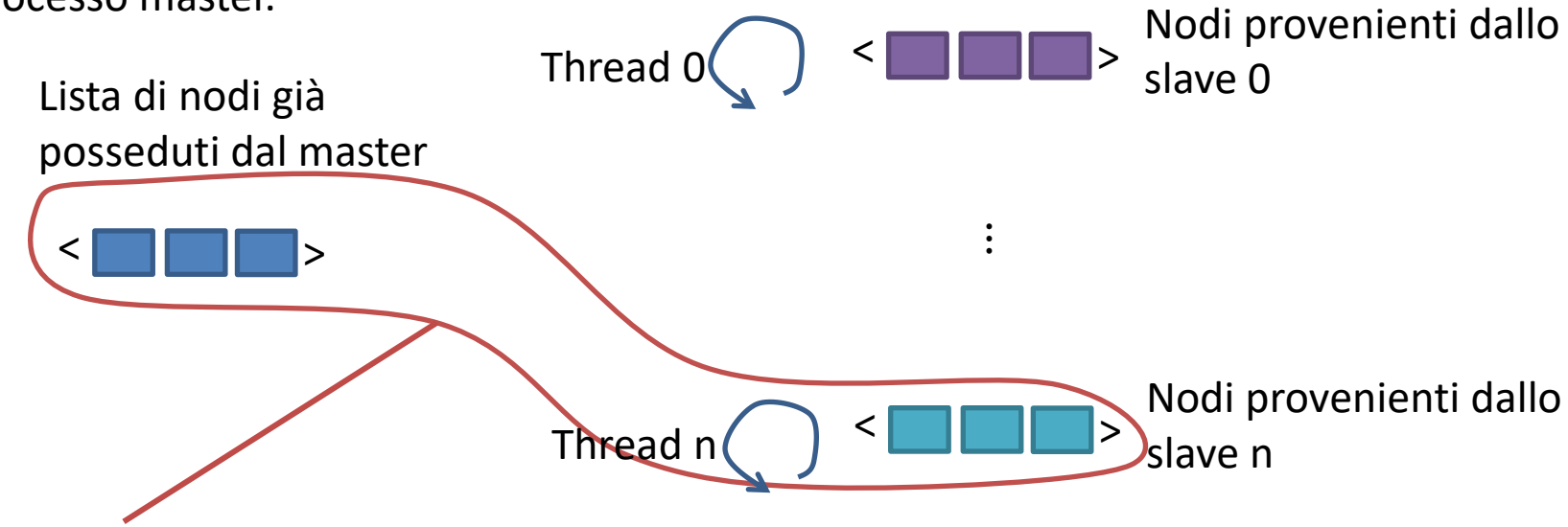
Step 4: Esempio di albero ottenuto. I vari colori differenziano i nodi ottenuti dai vari slaves.



RRT, parallel implementation

Step 4: La versione RRT bidirezionale non è implementabile, mentre la versione RRT* può esserlo nella seguente maniera.

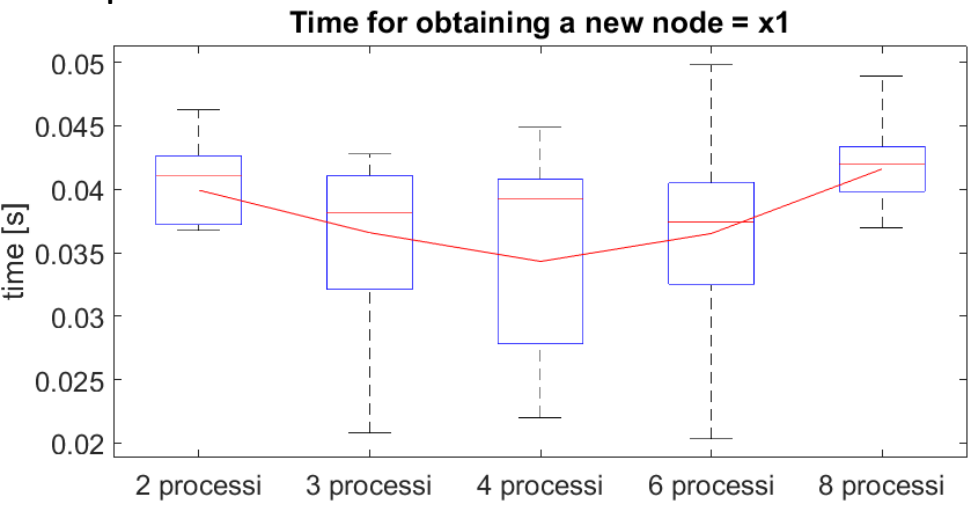
Gli slaves espandono la nuova radice ricevuta attraverso le normali iterazioni dell'algoritmo RRT e comunicano i vari nodi trovati al master. E' il master che all'atto della ricezione svolge i rewird sui nodi che riceve. Tale operazione può essere parallelizzata da una sezione parallela di threads, usando OpenMP: i nodi ricevuti vengono smistati in liste temporanee, una per ogni slave/thread. Per svolgere i rewird (e quindi calcolare i vari set $Near$) si considerano i nodi già posseduti dal master, e quelli in una specifica lista di 'smistamento'. In questa maniera sui thread non devono essere applicate regioni critiche. Al termine della ricezione, tutti nodi vengono prelevati dalle liste di smistamento per essere aggiunti all'albero del processo master.

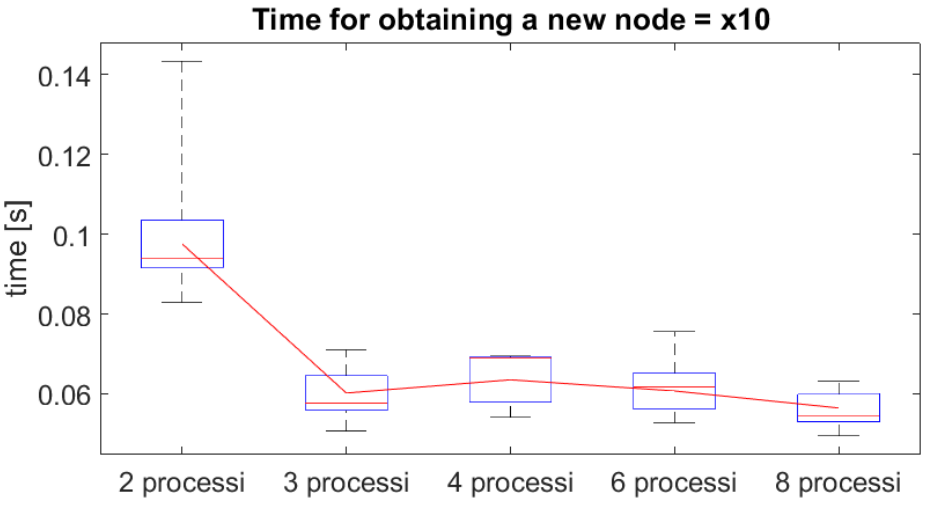


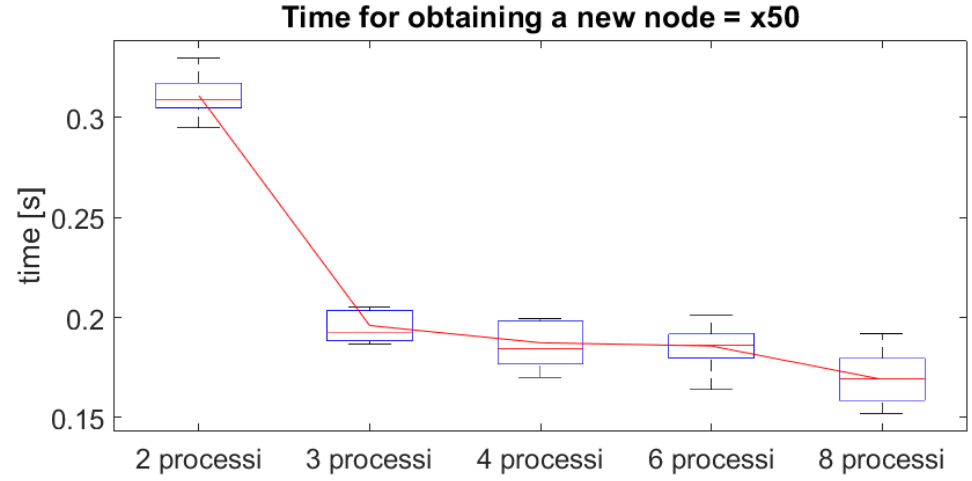
Lista di nodi considerata per la determinazione del $Near$ set del prossimo nodo che verrà aggiunto alla lista di smistamento n

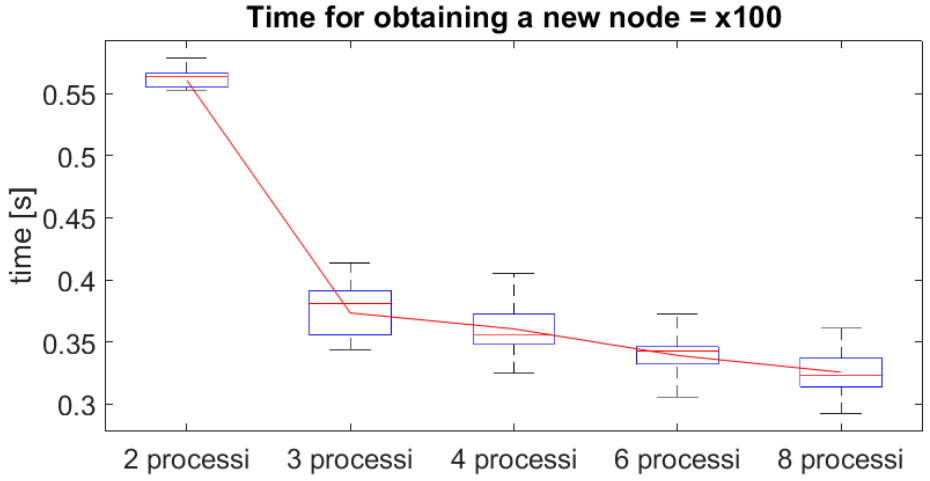
RRT-RRT*, parallel implementation

Step 4: results RRT



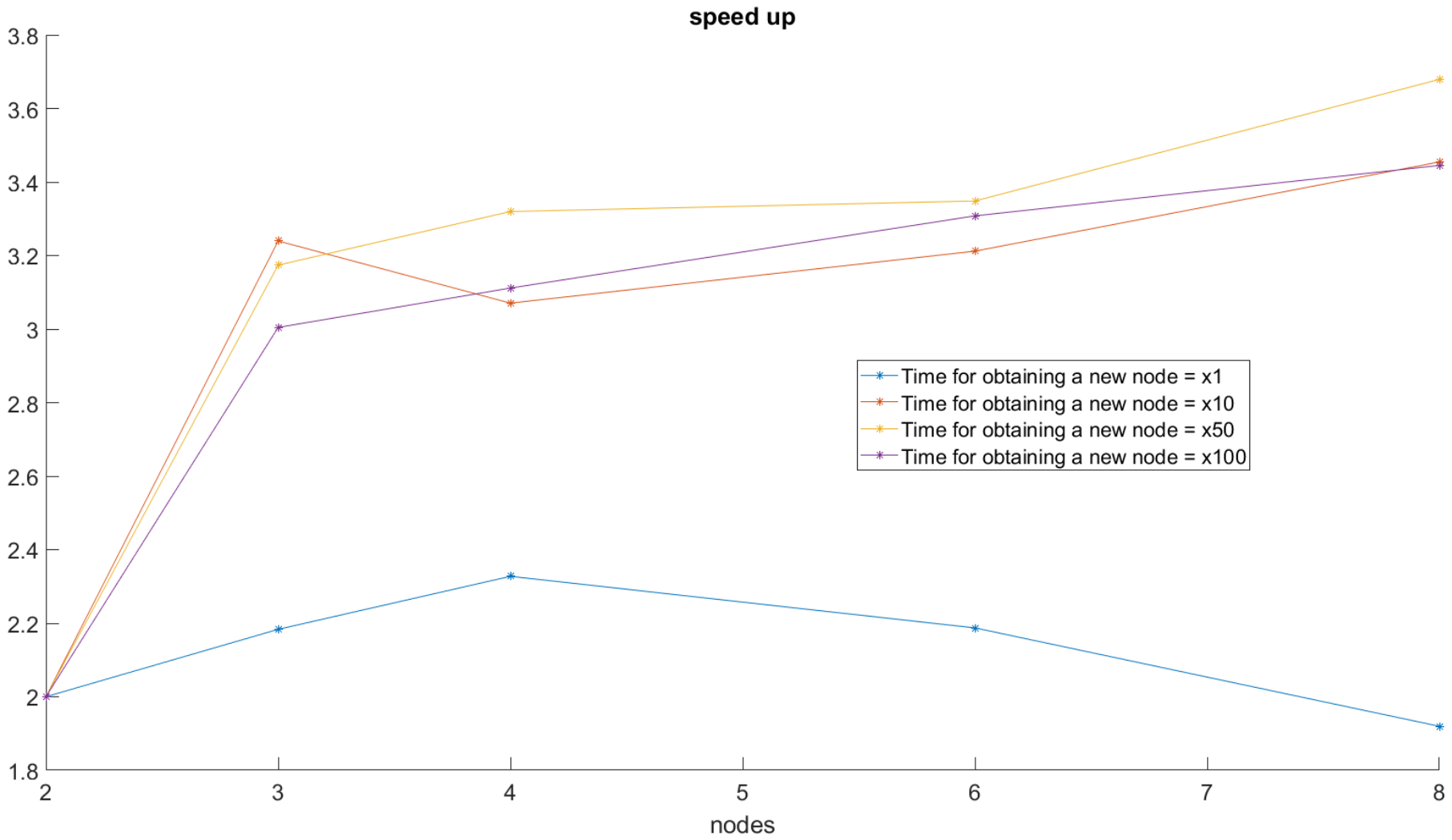






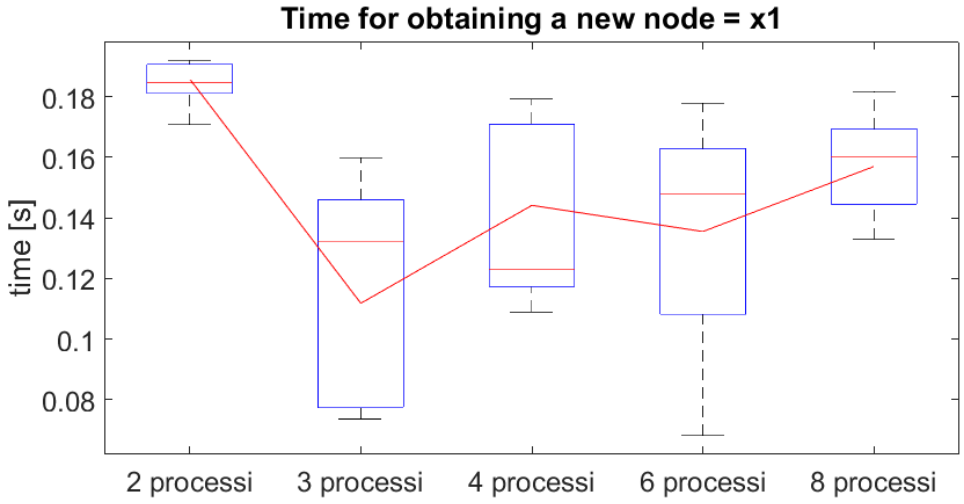
RRT-RRT*, parallel implementation

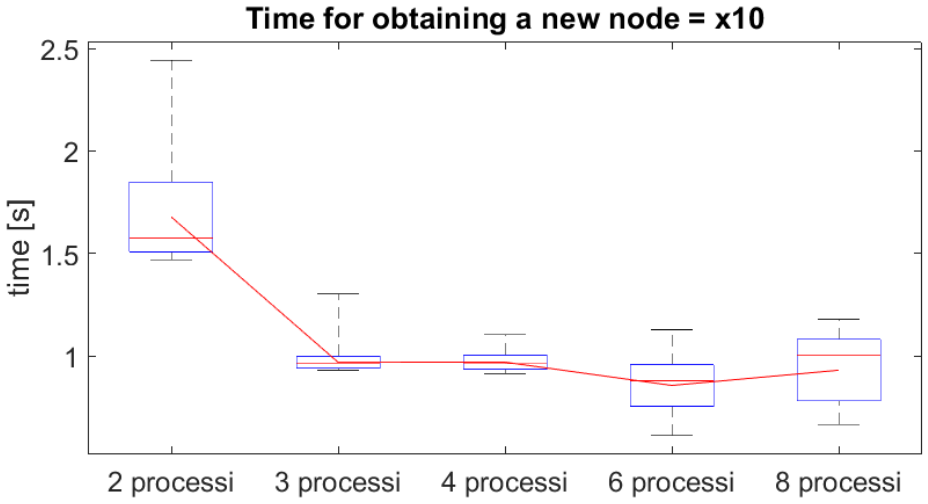
Step 4: results RRT

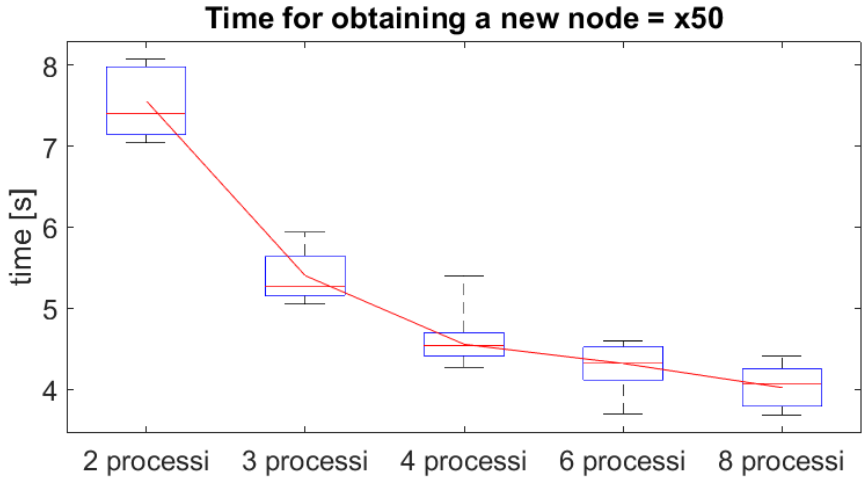


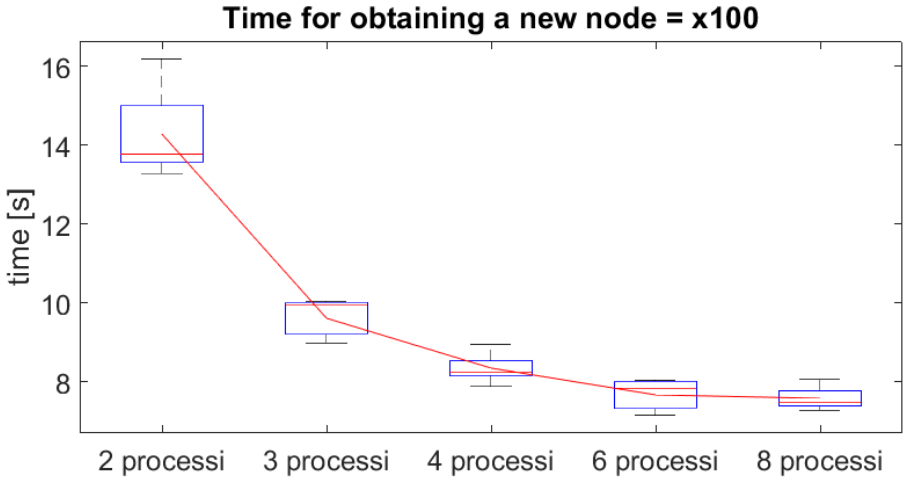
RRT-RRT*, parallel implementation

Step 4: results RRT*



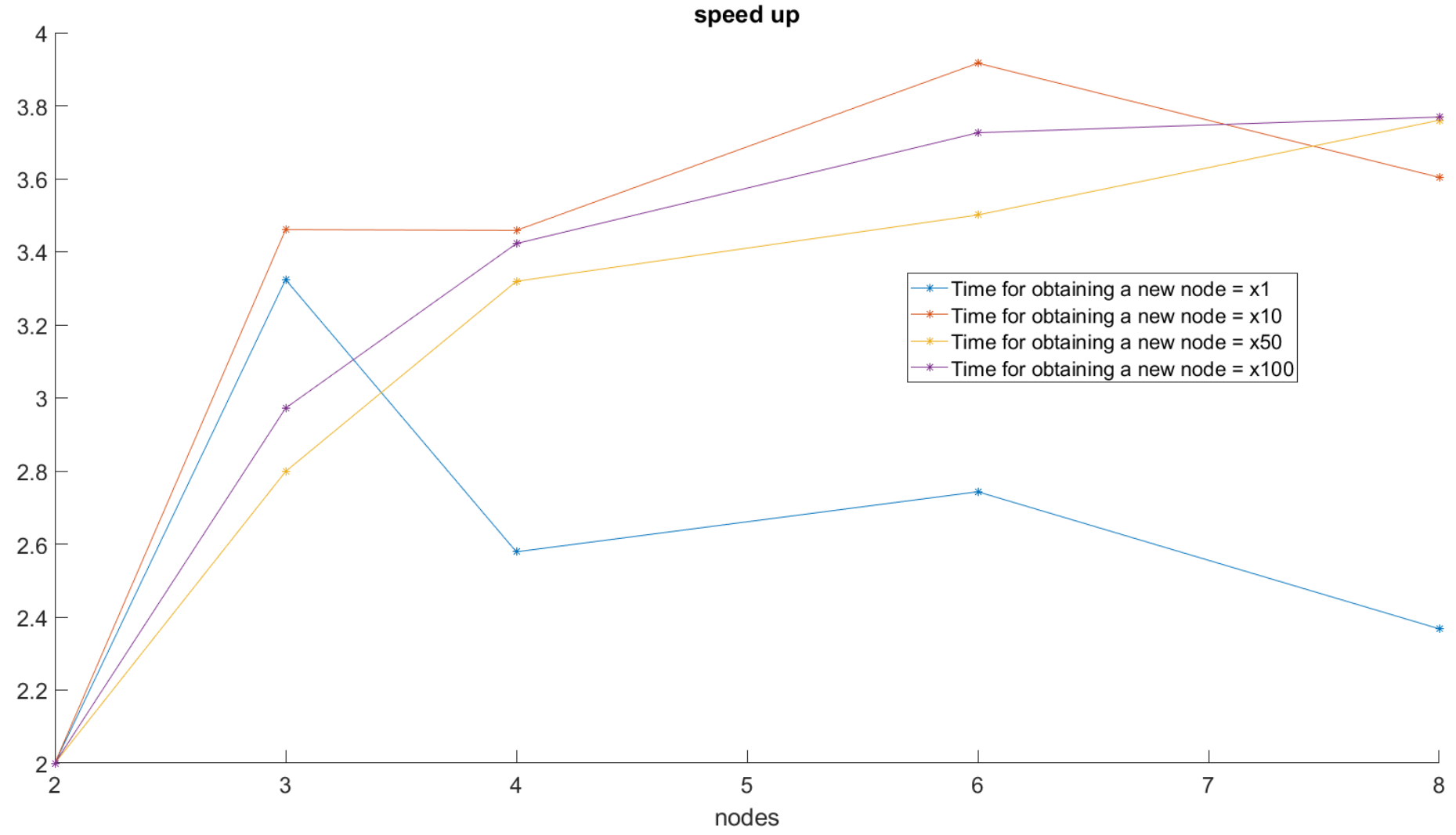




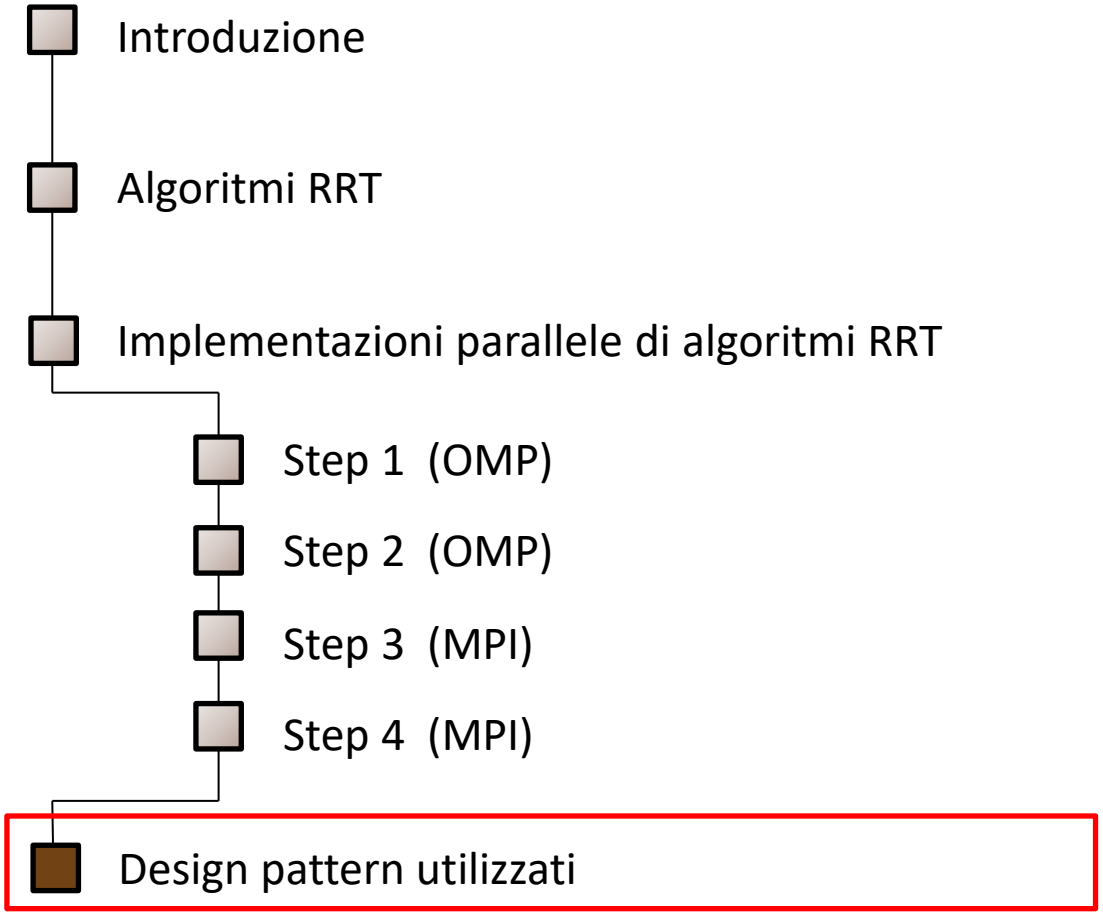


RRT-RRT*, parallel implementation

Step 4: results RRT*

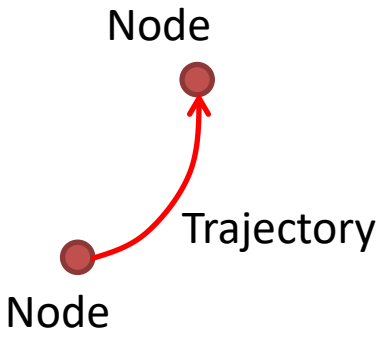


Sommario


Introduzione
Algoritmi RRT
Implementazioni parallele di algoritmi RRT
Step 1 (OMP)
Step 2 (OMP)
Step 3 (MPI)
Step 4 (MPI)
Design pattern utilizzati

Generalità dell'implementazione

Le traiettorie ottimali che connettono gli stati vengono calcolate tramite una fabbrica di nodi



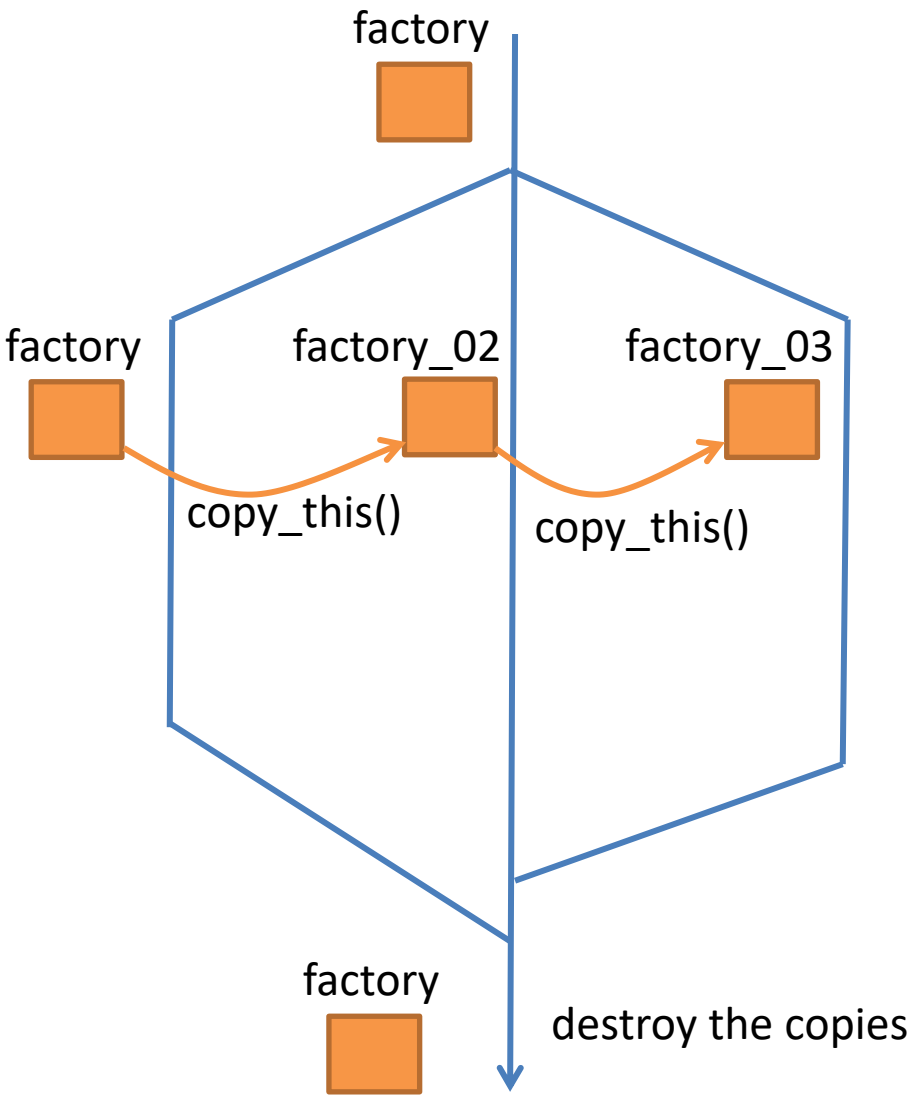
Trajectory_Factory
Trajectory_Factory* copy_this() Trajectory * Get_optimal_traj(Node* start,Node* target)

Node
float* mState

Trajectory
float Get_cost() Node* Get_next_on_traj()

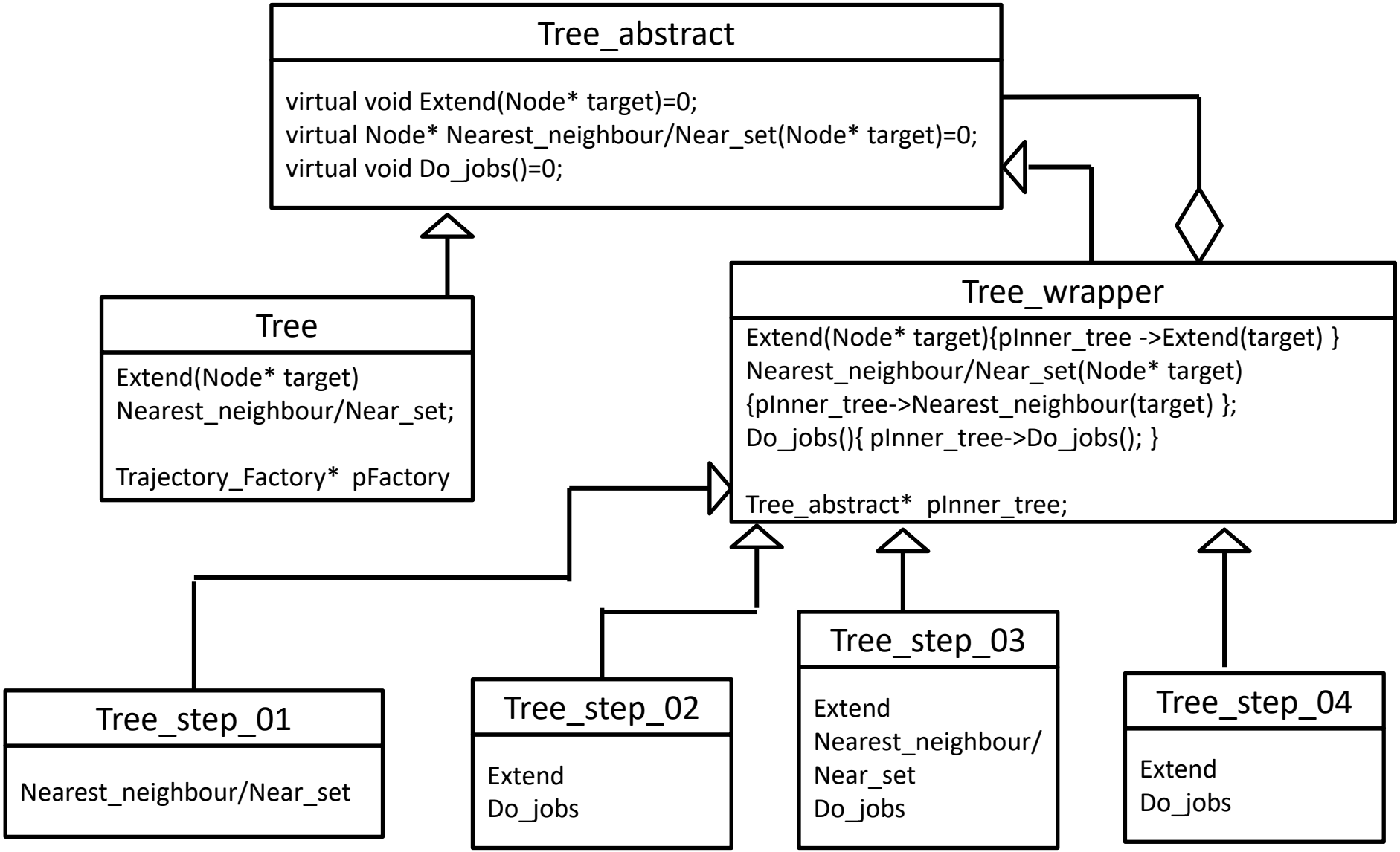
Generalità dell'implementazione

Nel caso di implentazioni multi threading, le fabbriche di traiettorie vengono copiate nei thread, evitando di dover proteggere in sezioni critiche l'accesso ad un'unica fabbrica

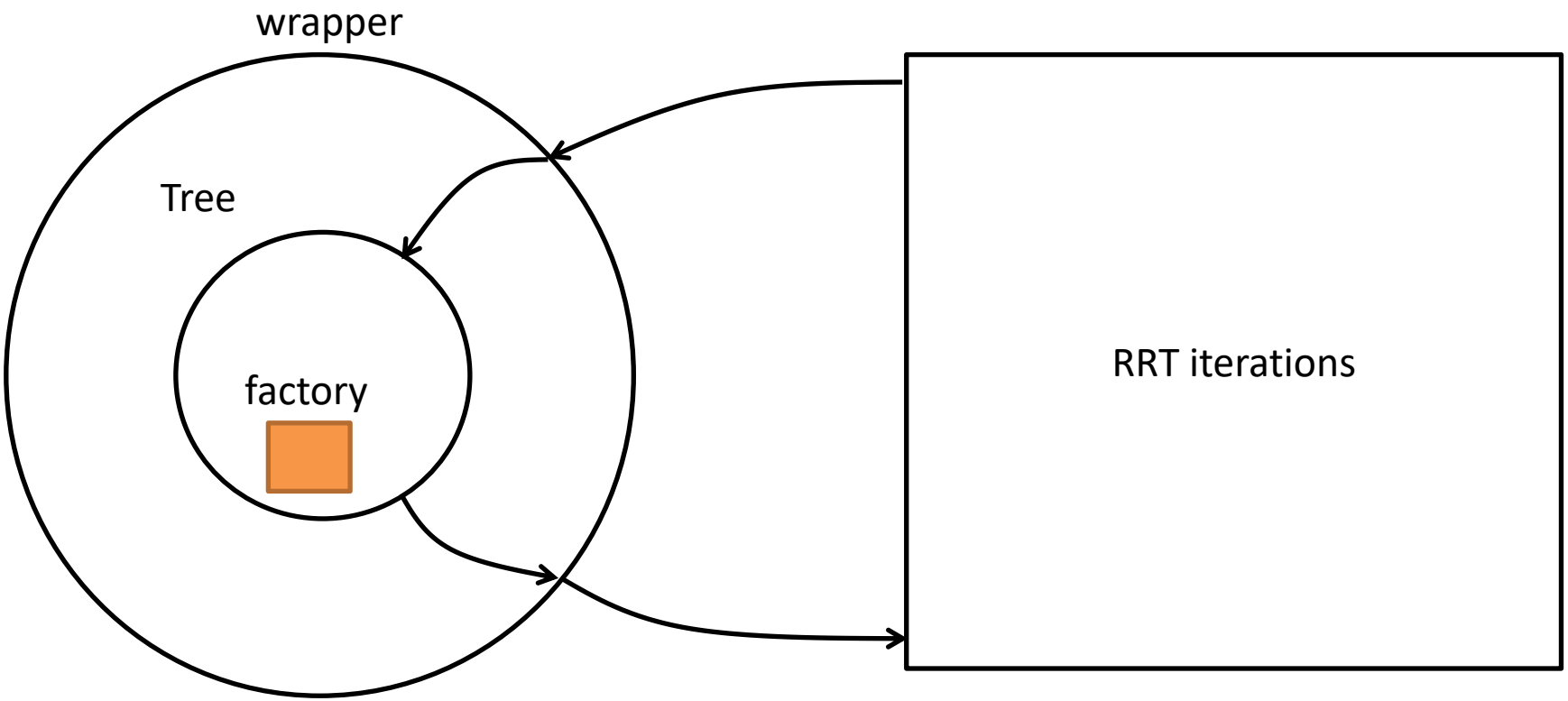


Generalità dell'implementazione

I vari step di estensione dell'albero (o degli) descritti vengono gestiti nelle diverse implementazioni attraverso l'utilizzo di wrapper



Generalità dell'implementazione



Generalità dell'implementazione