ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

Analisi del linguaggio x10 per architetture parallele: il caso di studio dell'algoritmo Gift Wrapping

Tesi in
High Performance Computing

Relatore:
Prof. Moreno Marzolla

Candidato:
Simone Romagnoli

Anno Accademico 2019 – 2020

Alla mia famiglia

Indice

Introduzione **1**

1 Introduzione al calcolo parallelo **3**
1.1 High Performance Computing 3
1.2 Valutazione delle prestazioni 3
1.2.1 Speed-up 4
1.2.2 Efficienza 5

2 Il problema dell'inviluppo convesso **9**
2.1 L'algoritmo 9
2.2 Gift Wrapping 10

3 Analisi di x10 **13**
3.1 Nascita e sviluppo del linguaggio x10 13
3.2 Componenti 14
3.2.1 Activity 15
3.2.2 Place 16
3.3 Potenzialità del linguaggio 18
3.4 Contenuto dell'elaborato 18
3.4.1 Versione a memoria condivisa 20
3.4.2 Versione a memoria distribuita 22

4 Valutazione delle prestazioni **25**
4.1 Versione a memoria condivisa 26
4.1.1 Speed-up 27
4.1.2 Strong Scaling Efficiency 29
4.1.3 Weak Scaling Efficiency 29
4.2 Versione a memoria distribuita 31
4.2.1 Speed-up 31
4.2.2 Strong Scaling Efficiency 32
4.2.3 Weak Scaling Efficiency 33

Conclusioni **35**

Ringraziamenti **37**

Bibliografia **39**

Introduzione

L'obiettivo di questa tesi consiste nello studio di x10, un linguaggio di programmazione orientato agli oggetti le cui componenti estendono linguaggi sequenziali come Java e C++ per permettere il calcolo parallelo. Per trarre delle conclusioni pratiche, questa tesi analizza l'algoritmo per il calcolo dell'inviluppo convesso, uno degli algoritmi fondamentali usati in geometria computazionale, e ne illustra una versione parallela implementata in x10.
Il concetto alla base del calcolo parallelo è quello di avere più unità di calcolo che si suddividono delle computazioni equamente, in modo da ridurre il tempo d'esecuzione di quest'ultime: l'High Performance Computing è la disciplina che studia il calcolo parallelo, e, negli anni, sono stati sviluppati diversi linguaggi e protocolli con i quali è possibile parallelizzare programmi sequenziali rendendoli estremamente più veloci.
X10 è un linguaggio basato su classi e staticamente tipizzato, e rientra fra i linguaggi di High Performance Computing: è stato progettato per ottenere calcoli ad alte prestazioni su computer che supportino circa 10^{15} operazioni al secondo e circa 10^5 thread hardware [6]. Un fattore importante che ha contribuito allo sviluppo di x10 è stato quello di voler aumentare la produttività dei programmatori; infatti, è stato concepito per avere una semantica semplice, in modo da essere facilmente compreso da persone familiari con linguaggi orientati agli oggetti.
Il lavoro di tesi è stato suddiviso nei seguenti capitoli:

- **Capitolo 1** - Introduzione al calcolo parallelo;
- **Capitolo 2** - Il problema dell'inviluppo convesso;
- **Capitolo 3** - Analisi di x10;
- **Capitolo 4** - Valutazione delle prestazioni.

Capitolo 1

Introduzione al calcolo parallelo

In questo capitolo viene brevemente introdotto il calcolo parallelo per fornire una visione più specifica del lavoro che verrà svolto.

1.1 High Performance Computing

Il calcolo parallelo nasce con l'obbiettivo di incrementare le prestazioni delle computazioni che hanno un costo elevato, ad esempio calcoli di fisica quantistica o simulazioni di eventi astronomici.
Alla base dell'High Performance Computing, vi è il concetto di concorrenza: la computazione viene divisa in flussi di dati, comunemente chiamati *thread*, che eseguono in maniera indipendente ripartendosi una porzione del carico di lavoro, operando sullo stesso insieme di dati. Tuttavia, gestire un gruppo di *thread* non è semplice: bisogna tener conto della possibilità di sovrascrittura dei dati da parte di *thread* diversi (*race condition*) oppure della necessità di sincronizzarsi per mantenere il parallelismo temporale. Di fatto, esistono una serie di costrutti che i linguaggi di HPC utilizzano per risolvere i problemi del calcolo parallelo: un esempio sono le barriere, che bloccano i *thread* finché tutti non le raggiungono in modo da sincronizzarli, oppure l'utilizzo di operazioni atomiche - che non vengono mai interrotte dall'inizio di altre istruzioni, e quindi possono essere considerate sequenziali - per evitare *race condition*.

1.2 Valutazione delle prestazioni

Nella parallelizzazione di un programma, è essenziale comprendere quanto esso possa migliorare con il progressivo aumentare dei processori a disposizione. La valutazione delle prestazioni serve a capire la scalabilità del programma,

ovvero la sua capacità di migliorare o peggiorare in funzione delle risorse. Per valutare le prestazioni si fa riferimento a due tipi di misure: lo *speed-up* e l'efficienza.

1.2.1 Speed-up

Lo scopo primario nella progettazione di un programma parallelo è quello di ottenere un miglioramento di tempo rispetto alla versione seriale: se effettivamente la versione parallela risulta più veloce, allora si può dire di aver ottenuto uno *speed-up*, ovvero un miglioramento delle prestazioni del programma, in termini di velocità.
Lo *speed-up* non è solo un concetto, ma è anche misurabile in maniera precisa; definendo `p` come il numero di processori utilizzati, T_s il tempo d'esecuzione del programma seriale, T_p`(p)` quello del programma parallelo usando `p` processi o *thread*, lo *speed-up* `S(p)` è:

$$S(p) = \frac{T_s}{T_p} \simeq \frac{T_p(1)}{T_p(p)}$$

Nel caso ideale, un programma parallelo impiega $\frac{1}{p}$ del tempo della versione seriale: ad esempio, un calcolo che impiega 60 secondi, se parallelizzato ed eseguito con 2 processori, il tempo di computazione diventa $\frac{60}{2} = 30$; nel caso i processori siano 3, il tempo calerebbe a 20 secondi, con 6 processori a 10 secondi, e così via (tempi illustrati nel grafico 1.1). Pertanto, dalla formula si ottiene che il caso ideale di *speed-up* è $S(p) = p$, ovvero uno *speed-up* lineare (mostrato in figura 1.2).

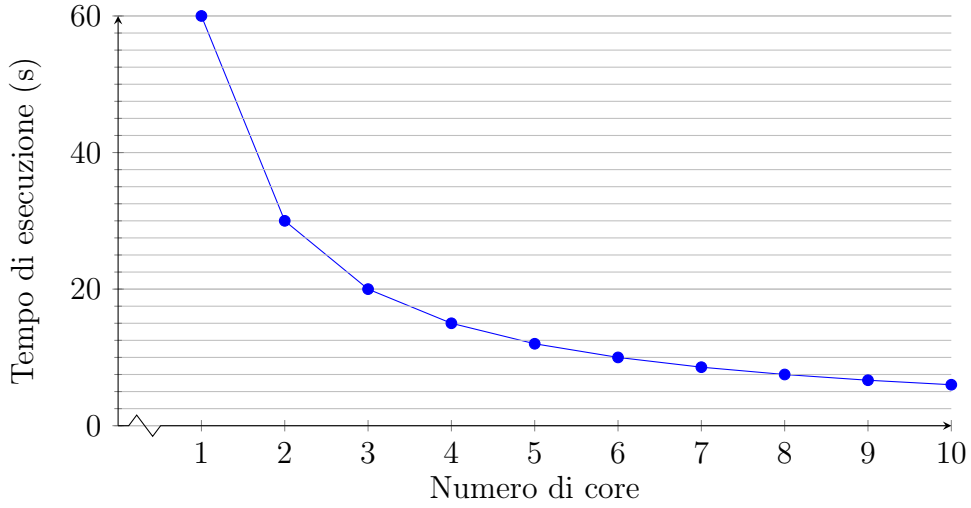

Figura 1.1: esempio di grafico dei tempi con *speed-up* lineare

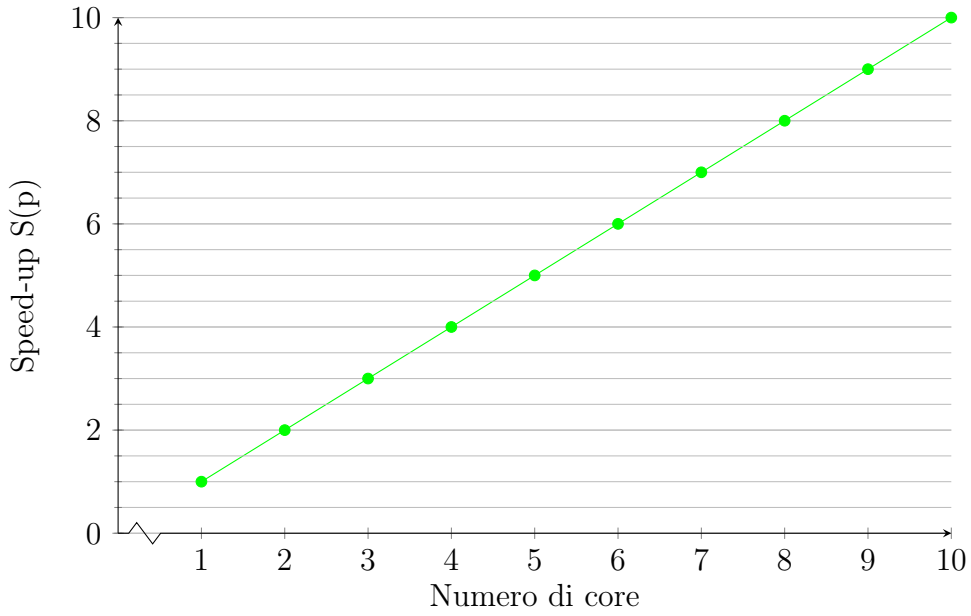

Figura 1.2: grafico dello *speed-up* lineare relativo alla figura 1.1

Nella pratica è difficile ottenere uno *speed-up* lineare in quanto spesso si riscontrano porzioni di codice non parallelizzabili, ma anche per questioni di livello più basso; quindi, realisticamente, lo *speed-up* risulta essere $S(p) \leq p$. Nel caso in cui vi sia una porzione di codice α non parallelizzabile, e supponendo che il restante $(1-\alpha)$ lo sia, si ottiene un tempo d'esecuzione parallelo rappresentato dalla seguente equazione:

$$T_p(p) = \alpha T_s + \frac{(1-\alpha)T_s}{p}$$

Non sempre α corrisponde a porzioni di codice non parallelizzabile: spesso ci sono dei ritardi dovuti a comunicazioni tra i *thread* o dipendenze dei dati insite negli algoritmi.
In rari casi, si può riscontrare uno *speed-up* superlineare, cioè $S(p) > p$; questa particolarità viene causata da fattori esterni che non riguardano il codice come l'utilizzo di tecniche di caching o l'eterogeneità dell'hardware.

1.2.2 Efficienza

L'efficienza è una misura che permette di interpretare la scalabilità di un programma parallelo; è un valore spesso compreso tra 0 e 1 che indica se il programma scala con risultati positivi. Esistono due tipi di misura dell'efficienza:

la Strong Scaling Efficiency e la Weak Scaling Efficiency.
La Strong Scaling Efficiency - o scalabilità forte - rappresenta la capacità di mantenere un buon valore di *speed-up* con l'aumentare del numero di processori e tenendo fissa la dimensione totale del problema che si sta valutando. Nello specifico, si indica con $E(p)$ e si calcola come segue:

$$E(p) = \frac{S(p)}{p}$$

Lo scopo è quello di ridurre la porzione di lavoro svolto da ogni core con l'aumentare di questi, in modo da migliorare il tempo totale d'esecuzione mano a mano che si aggiungono processori. Di conseguenza, se un programma presentasse uno *speed-up* pressoché lineare, si potrebbe già prevedere una buona efficienza nella scalabilità forte. Nei grafici 1.3 e 1.4 sono mostrati rispettivamente un esempio di misurazione del tempo di un programma e la relativa Strong Scaling Efficiency.

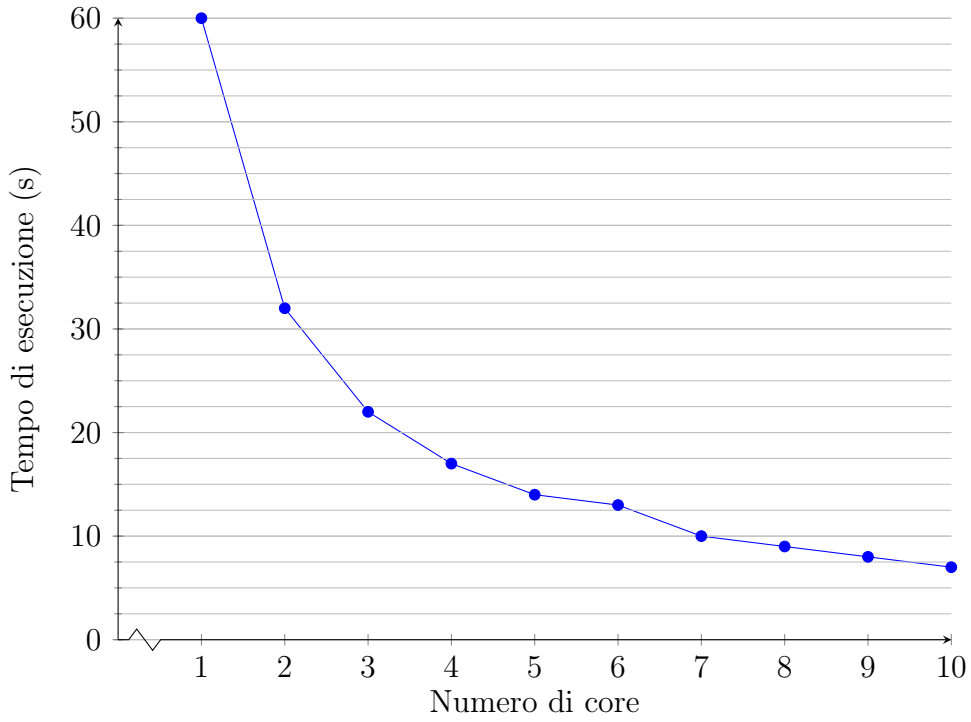

Figura 1.3: esempio di tempi per il calcolo della scalabilità forte

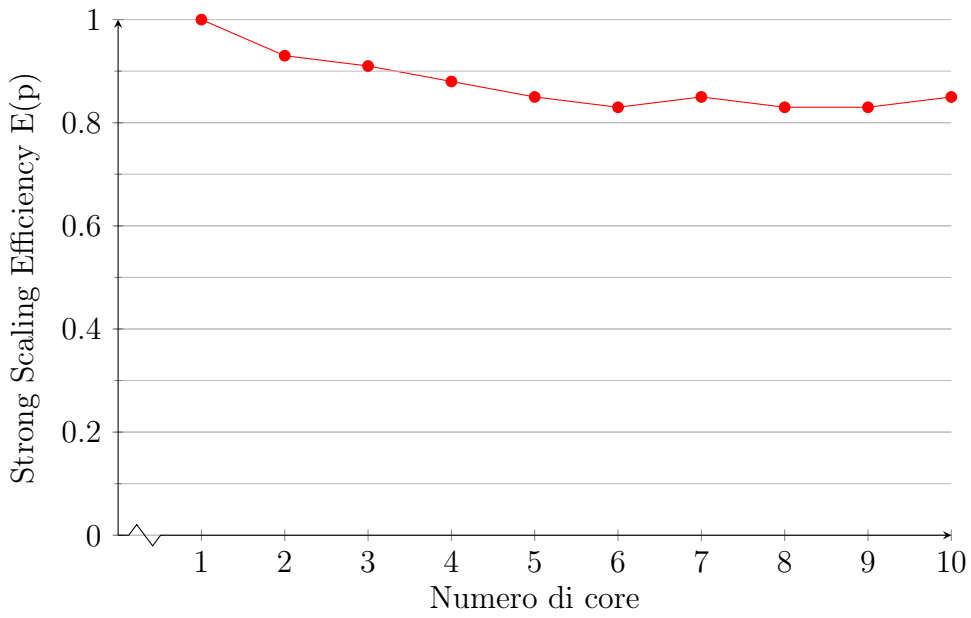

Figura 1.4: esempio di Strong Scaling Efficiency quando lo *speed-up* è quasi lineare

La Weak Scaling Efficiency - o scalabilità debole - interpreta la capacità di scala di un programma mantenendo fisso il carico di lavoro di ogni processore, aumentando di conseguenza la dimensione totale del problema con il crescere del numero `p` di processori. Si indica con $W(p)$ e si calcola come segue:

$$W(p) = \frac{T_1}{T_p}$$

dove T_1 rappresenta il tempo necessario per completare 1 unità di lavoro con 1 processore, mentre T_p è quello necessario per completare p unità di lavoro con p processori.
L'obiettivo della scalabilità debole è quello di controllare se un problema può essere risolto in dimensioni più grandi nello stesso lasso di tempo; il carico di lavoro totale del problema aumenta proporzionalmente con il crescere del numero p di processori. A differenza della scalabilità forte, quella debole non è strettamente correlata allo *speed-up*; di fatti, esso non compare nella formula poiché la Weak Scaling Efficiency mira al mantenimento dei tempi con il crescere dell'input di un problema, piuttosto che al miglioramento di essi. Nei grafici 1.5 e 1.6 sono mostrati rispettivamente un esempio di misurazione del tempo di un programma e la relativa Weak Scaling Efficiency.

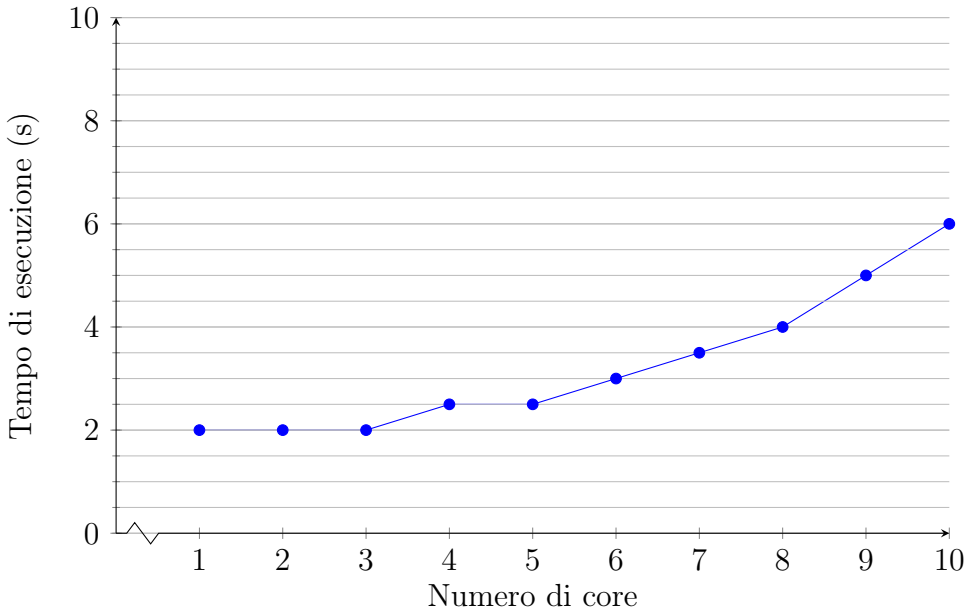

Figura 1.5: esempio di tempi per il calcolo della scalabilità debole

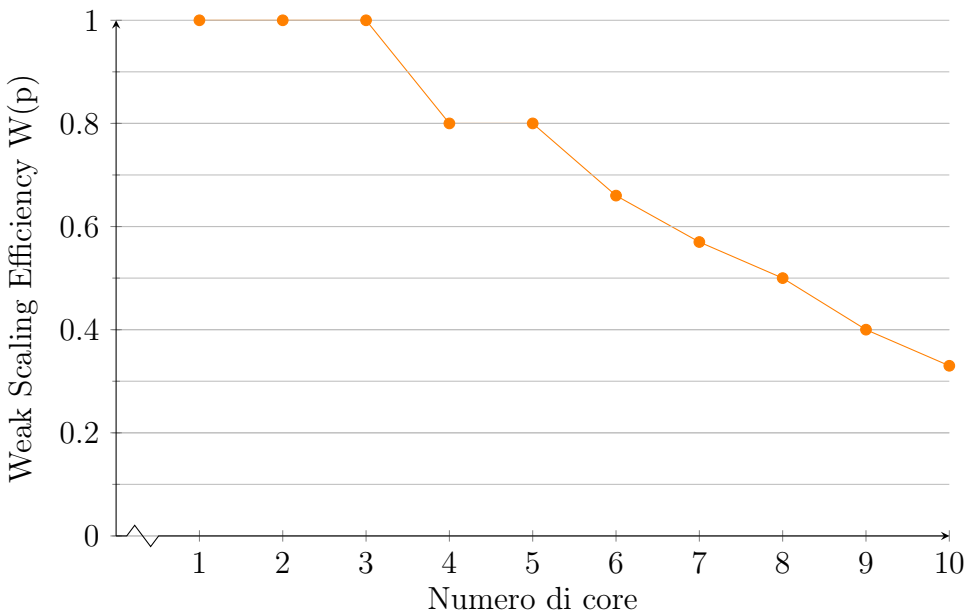

Figura 1.6: Weak Scaling Efficiency relativo ai tempi della figura 1.5

Capitolo 2

Il problema dell'inviluppo convesso

In questo capitolo viene introdotto il problema del calcolo di un inviluppo convesso. Nello specifico, si discute dei campi nei quali può essere utilizzato e si elencano alcune versioni esistenti.

2.1 L'algoritmo

In matematica, si definisce inviluppo convesso (o talvolta involucro convesso) di un sottoinsieme $\boldsymbol{I}$ di uno spazio vettoriale reale, l'intersezione di tutti gli insiemi convessi che contengono $\boldsymbol{I}$ [7].
Nello spazio vettoriale $\boldsymbol{R}^2$, l'inviluppo convesso di un insieme di punti $\boldsymbol{P}$ corrisponde al poligono convesso di area minima che contiene ogni punto di $\boldsymbol{P}$; un esempio è mostrato nella figura 2.1.

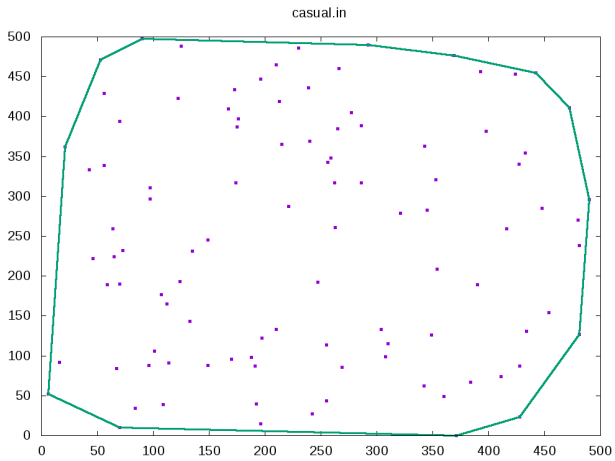

Figura 2.1: Inviluppo convesso di un insieme di 100 punti casuali

Il calcolo dell'inviluppo convesso è sfruttato in diversi ambiti: si applica a discipline come la matematica, la geometria, ma anche la ricerca operativa, la visione artificiale e molto altro; ad esempio, può essere utile nella segmentazione d'istanze di una rete neurale convoluzionale, ovvero evidenziando i bordi di un elemento in un'immagine riconosciuto da un algoritmo di machine learning. Il termine 'inviluppo convesso' (in inglese 'convex hull') è divenuto standard nel 1938, ma l'algoritmo in sè esiste da molto più tempo: l'involucro convesso di un insieme di punti nel piano appare in una lettera di Isaac Newton risalente al 1676 [7].

L'inviluppo convesso di un insieme di punti può essere trovato in diversi modi: infatti esistono diversi algoritmi conosciuti che risolvono tale problema; di seguito, ne vengono elencati e descritti alcuni. L'inviluppo convesso composto da h punti di un insieme di n punti può essere trovato con:

- **Gift Wrapping** (o anche **Jarvis march**) - uno degli algoritmi più semplici anche se meno efficiente, costruisce l'involucro convesso in maniera incrementale con complessità computazionale di $0(nh)$ [4]; è l'algoritmo che è stato implementato in x10 all'interno del progetto di questa tesi, e di conseguenza verrà approfondito successivamente in questo capitolo.

- **Graham scan** - è più sofisticato del Gift Wrapping, ma anche più efficiente: di fatti, ha una complessità computazionale di $O(n \log n)$; tale algoritmo richiede di ordinare i punti di input e, come conseguenza, in caso siano già ordinati, la complessità si riduce a $O(n)$.

- **QuickHull** - ha un approccio di tipo *divide et impera* e ha una complessità computazionale di $O(n \log n)$ che può degenerare a $O(n^2)$ nel caso pessimo.

- **Algoritmo di Chan** - ideato nel 1996, è il più recente degli algoritmi che risolvono il problema dell'inviluppo convesso: combina il Gift Wrapping con l'esecuzione del Graham scan su piccoli sottoinsiemi dell'input; ciò implica un miglioramento della complessità al tempo $O(n \log h)$

Per questa tesi è stato scelto il Gift Wrapping come algoritmo da parallelizzare in quanto si vuole evidenziare l'efficacia del calcolo parallelo su procedimenti relativamente complessi.

2.2 Gift Wrapping

Come anticipato, Gift Wrapping trova gli h vertici dell'inviluppo convesso di un insieme di n punti in maniera incrementale, ovvero esegue h iterazioni in ognuna delle quali determina uno dei punti del risultato finale. Siccome per

ogni iterazione viene letto ognuno degli n punti dell'insieme, la complessità dell'intero procedimento è a $O(nh)$. Nel caso in cui l'insieme di input sia già un involucro convesso, ogni vertice appartiene anche al risultato finale dell'algoritmo: tale situazione costituisce il caso pessimo per il Gift Wrapping, con la quale si nota un cambiamento della complessità che peggiora a $O(n^2)$. L'inviluppo convesso viene costruito partendo da uno dei punti estremi dell'insieme di punti iniziale, ovvero uno dei vertici con coordinata minima o massima a scelta tra ascisse e ordinate, per poi procedere in senso orario o antiorario; per convenienza, in questa implementazione, consideriamo come punto di partenza il vertice con ascissa minore, cioè quello più a sinistra di tutti, il quale appartiene sicuramente all'inviluppo convesso, e, come senso di procedimento, quello orario. Sia `p` l'array monodimensionale contenente l'insieme di punti di input: ad ogni iterazione dell'algoritmo, partendo dall'ultimo vertice inserito `p[cur]`, viene aggiunto `p[next]` il vertice successivo dell'inviluppo convesso; ciò avviene facendo in modo che la tripletta `p[cur] - p[next] - p[j]` curvi verso destra per ogni `j`: in figura 2.2 sono illustrati i tipi di triplette di punti possibili.

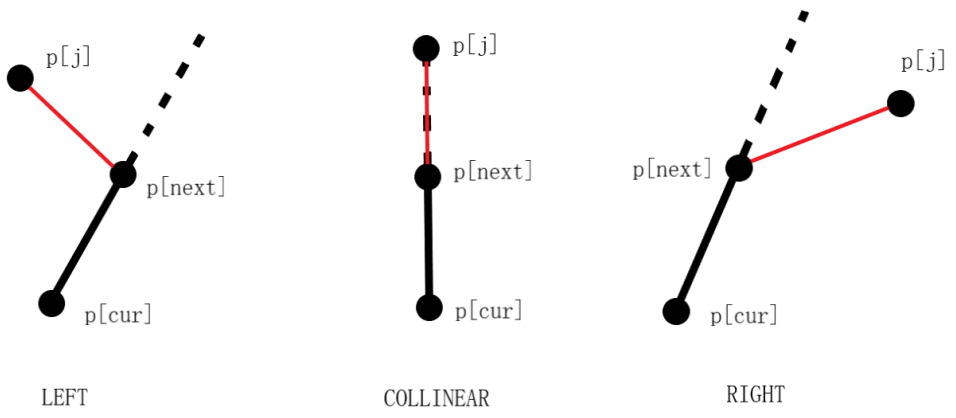

Figura 2.2: Triplette di punti: curva a sinistra, collineare e destra.

Per determinare l'indice `next` vengono controllati tutti i punti `p[j]` appartenenti all'insieme di input; nello specifico, si parte dal punto successivo a quello corrente, dopo di che si iterano tutti i punti `j`: se il vertice `j` in questione crea una curva a sinistra nella spezzata `p[cur] - p[next] - p[j]`, allora diventa il nuovo candidato a punto successivo.

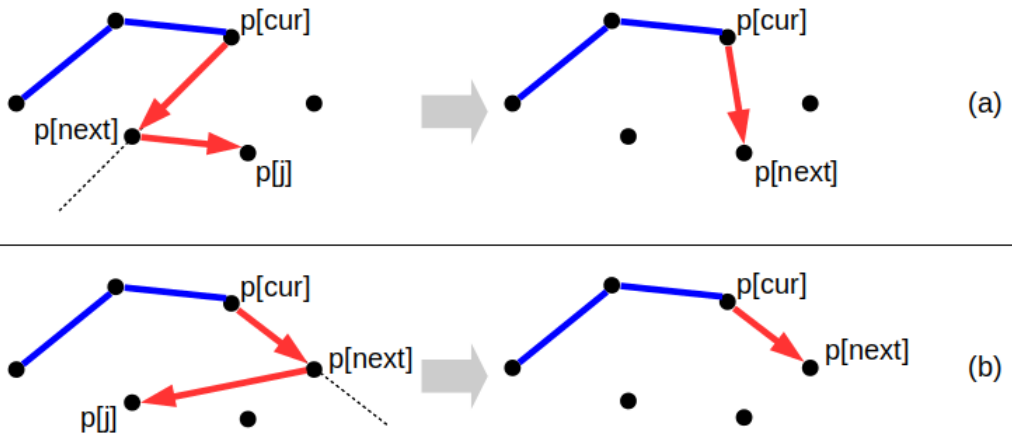

Figura 2.3: Determinazione del punto successivo del poligono convesso [5].

Analizzando la figura 2.3: al passo **(a)** la tripletta di punti considerata crea una curva a sinistra, di conseguenza il punto `next` viene sostituito da `j`; invece, al passo **(b)**, la spezzata curva verso destra, quindi il vertice `next` non viene cambiato.
Iterando in questa maniera si riesce a costruire il poligono convesso che contiene tutti i punti in input. L'algoritmo termina quando il punto `next` trovato alla fine di un ciclo corrisponde a quello iniziale, ovvero quello più a sinistra. L'intero procedimento può essere descritto con il seguente pseudocodice [5].

Algorithm: Gift Wrapping

Data: $P[]$ is the set of points
Result: $H[]$ is the set of corners of the convex hull of $P[]$

$cur = leftmost =$ index of leftmost point in $P[0 \dots n-1]$
do
  append $P[cur]$ to $H[]$
  $next = (cur + 1) \ \% \ n$
  for $j = 0$ *to* $n - 1$ **do**
    if $P[cur]$–$P[next]$–$P[j]$ *turns left* **then**
      $next = j$
    end
  end
  $cur = next$
while $cur \neq leftmost$

return $H[]$

Capitolo 3

Analisi di x10

In questa parte verrà analizzato il linguaggio x10: in primo luogo, verranno forniti dati storici sulla nascita e sullo sviluppo di x10 definendone le componenti e le potenzialità; in secondo luogo, verrà esposto l'elaborato mostrandone il codice e spiegandone le caratteristiche.

3.1 Nascita e sviluppo del linguaggio x10

La nascita dell'High Performance Computing ha portato molti vantaggi, tra i quali il principale è quello dell'incremento di velocità d'esecuzione dei programmi; tuttavia, non sono mancate delle problematiche: chi lavora in tale ambito deve sapere come dividere il lavoro disponibile in blocchi che possano essere eseguiti simultaneamente, senza incorrere in situazioni che blocchino i computer, come il *deadlock*[1]. Un tentativo di soluzione del problema è stato il passaggio al modello a memoria frammentata, in cui più processori si dividono i dati in memorie separate e interagiscono tra di loro grazie ad un sistema di scambio di messaggi: uno dei protocolli più utilizzati che implementa il *message-passing* è MPI (*Message Passing Interface*). Tale modello ha portato ad una perdita di produttività dei programmatori: il formato *message-passing* è integrato su un linguaggio ospite e il programmatore deve gestire l'interazione tra i processori così come lo scambio dei dati; inoltre, grandi strutture dati, come array distribuiti o grafi, devono essere pensate come frammentate in diversi nodi, quando concettualmente sono intese per essere unitarie [6].
La nascita del modello PGAS (*Partitioned Global Address Space*) è stata la soluzione naturale per le problematiche di produttività: il programmatore è portato a pensare ad una computazione eseguita per ciascun processore, ma

[1]Deadlock, indica una situazione in cui due o più processi o azioni si bloccano a vicenda, aspettando che uno esegua una certa azione che serve all'altro e viceversa.

con un'unica area di memoria condivisa comune; in più, i processori vengono sincronizzati automaticamente da un insieme di barriere globali. Il linguaggio x10 appartiene a questa classe e la estende grazie alle sue componenti asincrone rientrando quindi nel modello APGAS (*Asynchronous PGAS*).
Il progetto High Productivity Computing Systems portato avanti dal *DARPA*[2] aveva come scopo quello di sviluppare linguaggi che riescano a conseguire alte performance e produttività: tre linguaggi che si fecero strada all'interno del progetto furono x10, Fortress e Chapel; x10, prodotto da IBM, è quello che ha avuto più successo [1].

3.2 Componenti

X10 è un linguaggio staticamente tipizzato e orientato a oggetti; similmente a Java o altri linguaggi, x10 è un linguaggio basato su classi e interfacce, che consente anche ereditarietà e polimorfismo. Nel listato 3.1 viene fornito un semplice esempio di classe con campi, costruttori e metodi.

```
public class DoublePoint {

      var x:double;
      var y:double;

      def this(p:DoublePoint) { x = p.getX(); y = p.getY(); }
      def this(px:double, py:double) { x = px; y = py; }
      def this() { x = 0; y = 0; }

      public def getX() { return x; }

      public def getY() { return y; }

      public def setX(v:double) { this.x = v; }

      public def setY(v:double) { this.y = v; }

}
```

Listato 3.1: Esempio di classe in x10 che rappresenta un punto nel piano con coordinate di tipo Double.

[2]Defense Advanced Research Projects Agency, agenzia degli USA responsabile per lo sviluppo di tecnologie per utilizzo nel campo militare.

Nonostante estenda dei linguaggi sequenziali, il punto forte di x10 sono le sue componenti parallele. In termini di design del linguaggio parallelo, l'implementazione di componenti di concorrenza e distribuzione è stata fondamentale.

3.2.1 Activity

I thread di x10 vengono chiamati *Activity*: un qualsiasi programma x10 parte con l'invocazione del metodo `main`, il quale viene lanciato con una *Activity* iniziale, detta *root Activity*. Le computazioni x10 possono avere una o più *Activity* concorrenti che eseguono allo stesso momento; il comando per la loro creazione è `async`. Utilizzando `async {statements();}` viene lanciata una *Activity* che esegue il contenuto delle parentesi graffe in maniera indipendente, mentre l'*Activity* che ha lanciato il comando salta il blocco asincrono e riprende la sua esecuzione dalle istruzioni successive. Il corpo di una *Activity* è soggetto ad una restrizione per cui deve essere interpretabile come un metodo di tipo `void`: non deve avere un valore di ritorno e non può modificare delle variabili di tipo `var` . Nei programmi x10, le *Activity* non sono ordinate in alcun modo: non hanno un numero identificativo come i thread OpenMP o i processi MPI, tuttavia vengono aggiunte all'insieme di *Activity* generate.
Ogni *Activity*, ad eccezione di quella *root*, viene generata dinamicamente da un'altra; di conseguenza, nei programmi x10, esse vanno a formare un albero con radice la `main` *Activity*; per questo, le computazioni x10 vengono dette '*rooted*'. Ogni *Activity* può essere nei seguenti stati: *running*, *blocked* o *terminated*. Una *Activity* termina quando non ha più istruzioni da eseguire. x10 distingue tra terminazione locale e globale: una *Activity* termina localmente quando finisce l'esecuzione della sua ultima istruzione; invece, termina globalmente quando, oltre ad essere terminata localmente, tutte le *Activity* che può aver generato terminano globalmente [6]. I concetti di terminazione locale e globale sono importanti per il programmatore, in quanto deve essere consapevole del numero di *Activity* in esecuzione e del loro stato per poterle gestire e sincronizzare. Prendendo in esempio il seguente codice:

```
async {statement1();}
async {statement2();}
```

Supponendo di essere all'interno del metodo `main`, la *root Activity* genera due *child Activity* e, subito dopo, termina localmente. Le due *Activity* figlie eseguono le istruzioni all'interno del rispettivo blocco asincrono e possono richiedere più tempo per finire la loro computazione; quando sia `statement1();` sia `statement2();` terminano localmente, la *root Activity* termina globalmente. Per queste caratteristiche, x10 non permette la creazione di thread 'demoni', ovvero flussi di dati che sopravvivono anche dopo la terminazione della *root*

Activity.
La gestione della concorrenza è fondamentale, e x10 fornisce gli strumenti per riuscire a sincronizzare le *Activity* ed evitare situazioni di *race condition*.
I costrutti principali per la sincronizzazione in x10 sono l'istruzione `finish` e la classe *Clock*. L'istruzione `finish {statements();}` permette di creare un blocco all'interno del quale la terminazione globale viene convertita in locale: l'*Activity* che esegue il contenuto del blocco, una volta terminati gli `statements()`, attende che tutte le *Activity* figlie generate all'interno del blocco terminino. Invece, la classe *Clock* permette di generare delle istanze che possono essere associate alle *Activity*, ad eccezione di quella *root*, registrandole tramite l'istruzione `clocked`:

```
val c:Clock = Clock.make();
async clocked(c){s1();}
async clocked(c){s2();}
```

In questa maniera, i blocchi `s1()` e `s2()` eseguono le loro computazioni in parallelo, ma sono in grado di sincronizzarsi tramite l'istanza `c` di *Clock*; ad esempio, con il comando `c.advance()` viene creata una barriera per tutte le *Activity* che sono registrate all'istanza `c`.
Per evitare situazioni di *race condition*, x10 mette a disposizione un comando con il quale si garantisce l'atomicità delle istruzioni, il costrutto `atomic`. Con `atomic {statements();}` viene creato un blocco che viene eseguito sequenzialmente come se fosse un'unica istruzione atomica, nei confronti di altri blocchi `atomic` eseguiti dalle altre *Activity*. Per via di queste caratteristiche i blocchi `atomic` sono soggetti ad una serie di restrizioni; il corpo di un'istruzione `atomic` non può generare *Activity*, non può utilizzare direttive bloccanti come `finish` o `Clock.advance()`, e non può utilizzare l'espressione `at` per cambiare di *Place*.

3.2.2 Place

Per implementare la distribuzione dei dati, x10 utilizza delle partizioni fisiche di memoria che vengono chiamate *Place*: un *Place* può essere considerato come un deposito per dati che introduce il concetto di 'località'. Le *Activity* che eseguono all'interno di un *Place* possono accedere ai dati locali con facilità, mentre sono previsti dei costi di comunicazione per l'accesso a dati remoti, cioè a dati localizzati in altri *Place*. La classe `x10.lang.Place` fornisce una serie di metodi utili per gestirne le istanze: ad esempio, il metodo `Place.places()` ritorna l'insieme di *Place* in esecuzione, come istanza di `PlaceGroup` (un array di `Place`), per poterli iterare.
Ogni programma viene lanciato con un numero n di *Place* scelto a priori, settando la variabile d'ambiente `X10_NPLACES` oppure con l'opzione *-np n* da riga di comando. Essi sono numerati a partire da 0 e il rispettivo numero

identificativo è contenuto nel campo `place.id`; il metodo `Place.numPlaces()` ritorna il numero totale di *Place* in esecuzione. Ogni programma inizia con il metodo `main` che esegue all'interno di `Place.FIRST_PLACE`, il quale è il primo della lista `Place.places()`. La variabile `here` è utile per fare riferimento al *Place* corrente: essa punta costantemente al *Place* nel quale sta avvenendo la computazione, similmente a come `this` fa riferimento all'istanza di una classe. Quando un oggetto viene creato la sua istanza viene localizzata nel *Place* corrente e non può cambiare località; tuttavia, le variabili possono essere copiate da un *Place* a un altro tramite il costrutto `at`.

`at (`*Expression*`) {statements();}`

In questo modo si cambia il *Place* corrente a quello indicato in *Expression*, che deve essere di tipo *Place*; tutte le variabili a cui si fa riferimento in `statements()` vengono copiate in altre con lo stesso nome all'interno del *Place* indicato in *Expression*. Come ogni operazione distribuita, ha un costo elevato: richiede lo scambio di almeno due messaggi tra i *Place* e l'eventuale copia di un insieme di dati di dimensioni variabili. Il costrutto `at(p) {Stmts();}` non è asincrono: non crea una nuova *Activity*, ma deve essere visto come se trasportasse la *Activity* corrente in `p` e vi eseguisse `Stmts()`, per poi riportarla al *Place* di partenza.
La combinazione di `at` e `async` è possibile ed è la base per la programmazione distribuita in x10: sia `at(p) async {Stmts();}` sia `async at(p) {Stmts();}` sono sintatticamente corretti; in generale, è preferibile utilizzare la prima forma in quanto evita di creare una *Activity* solo per valutare `p`. Siccome nella programmazione distribuita è necessaria la presenza di dati persistenti in ogni partizione di memoria, il costrutto `at` non è d'aiuto visto che copia i valori di variabili per poi cancellarli; per questo problema è utile la classe generica `GlobalRef[T]`: essa salva in `here` un riferimento a delle variabili di tipo `T` che permane nella durata di vita del programma. Le variabili di tipo `GlobalRef` non vengono considerate dal costrutto `at` e, infatti, vi si può accedere solamente dal *Place* nel quale vengono create.
Il modo in cui x10 gestisce gli `async` è associandoli ad un gruppo di *thread worker* all'interno di ogni *Place*: il numero di *thread worker* presenti in ciascun *Place* è dato dalla variabile d'ambiente `X10_NTHREADS`. Per avere delle buone prestazioni, sarebbe corretto impostare un *thread worker* per ogni processore disponibile e dividere quest'ultimi equamente: per esempio, supponendo di voler usare un programma con 4 *Place* su una macchina con 12 core, bisognerebbe impostare la variabile `X10_NTHREADS` a 3 (4 *Place* $\times$ 3 *thread worker* per *Place* = 12 core attivi) [2].

3.3 Potenzialità del linguaggio

Grazie a questa struttura, x10 è un linguaggio che permette di parallelizzare codice permettendo al programmatore di costruirsi un'architettura a memoria condivisa oppure a memoria distribuita.

- **Memoria condivisa** - è possibile mantenendo tutti i dati all'interno di un unico Place e generando molteplici Activity; in questo modo, possono essere implementati diversi protocolli di parallelizzazione. Si può scegliere a priori un numero di *Activity* e generarle dinamicamente, oppure tale numero può essere scelto a run-time in base alle dipendenze algoritmiche. La combinazione di `async` con la classe `Clock` permette di implementare facilmente il protocollo master-slave, nel quale le *Activity* generate si coordinano con la principale.

- **Memoria distribuita** - è possibile distribuendo i dati tra i *Place* messi a disposizione e generando una *Activity* per ciascuno di questi. Come ogni programma a memoria distribuita, tale modello introduce sicuramente dei ritardi per via della comunicazione tra *Place*, ma può avere risultati positivi nel caso di grandi moli di dati o di algoritmi che non necessitano di costante sincronizzazione.

Attualmente, x10 non permette la creazione di *thread* demoni, ovvero *thread* che sopravvivono anche dopo il termine della *root Activity*; tuttavia, rendere l'*Activity* iniziale un demone potrebbe essere un'estensione futura del linguaggio per permettere computazioni che non terminano, come ad esempio quelle di un web server [6].
Infine, un altro punto di forza di x10 è *X10RT*, ovvero una libreria utilizzata dai programmi a memoria distribuita per la comunicazione tra *Place*: al momento, vi sono diverse implementazioni di *X10RT*, tra le quali quella standard è `STANDALONE`, che supporta l'utilizzo di molteplici *Place* su un singolo host; al contrario, l'implementazione `SOCKETS` utilizza il protocollo TCP per consentire molteplici *Place* su host diversi [3].

3.4 Contenuto dell'elaborato

In questa sezione viene illustrato il codice sviluppato ed esposte eventuali considerazioni pensate in fase di progettazione.
Nello specifico, è stato sviluppato un metodo per velocizzare anche la versione seriale del Gift Wrapping: tale algoritmo può eseguire un numero di iterazioni maggiore del necessario poiché aggiunge all'inviluppo convesso anche punti allineati tra di loro; i punti collineari possono essere non considerati, in modo da

ridurre il numero h di vertici dell'inviluppo convesso, e, di conseguenza, anche la complessità $O(nh)$. Questa ottimizzazione è possibile prendendo il punto più lontano in caso di vertici collineari durante la ricerca del punto successivo nell'inviluppo: in questo modo, se parte del poligono convesso è composta da punti allineati, l'algoritmo evita automaticamente quelli intermedi. Nel listato 3.2 viene fornita un'implementazione dell'algoritmo Gift Wrapping in x10 che contiene il miglioramento appena descritto.

```
public static def serialConvexHull(pset:PointSet):PointSet {

     val leftmost:long = leftmost(pset);
     var hull:PointSet = new PointSet(pset.length());
     var cur:long = leftmost;
     var next:long;

     do {
          hull.p(hull.n) = pset.p(cur);
          hull.n++;
          next = (cur + 1) % pset.n;

          for(j in 0..(pset.n-1)) {
             val d:Direction = turn(pset.p(cur), pset.p(next),
                 pset.p(j));
             if ( d.isCollinear() && distanceBetween(pset.p(cur),
                 pset.p(j)) > distanceBetween(pset.p(cur),
                 pset.p(next)) ) {
                next = j;
             }
             if ( d.isLeft() ) {
                next = j;
             }
          }
          cur = next;

     } while(leftmost!=cur);

     return new PointSet(hull);
}
```

Listato 3.2: Implementazione seriale dell'algoritmo Gift Wrapping in x10.

Sia per la versione seriale che per quelle parallele, sono stati utilizzati i seguenti metodi:

- `leftmost(pset:PointSet)` - determina l'indice del punto più a sinistra

nell'insieme `pset`;

- `turn(p0:DoublePoint, p1:DoublePoint, p2:DoublePoint)` - determina se la tripletta ordinata di punti `p0 - p1 - p2` crea una curva verso destra, verso sinistra o dritta;

- `distanceBetween(p0:DoublePoint, p1:DoublePoint)` - calcola la distanza nel piano tra i punti `p0` e `p1`.

Un problema nella parallelizzazione dell'algoritmo Gift Wrapping è la presenza di una dipendenza in ogni sua iterazione: per poter determinare il punto successivo del poligono convesso, Gift Wrapping ha bisogno di conoscere l'ultimo vertice trovato; di conseguenza, non è possibile parallelizzare in maniera semplice il ciclo *do-while* in quanto ogni iterazione dipende dalla precedente. Un modo per parallelizzare tale ciclo consiste nel determinare in anticipo dei punti che appartengono sicuramente all'inviluppo convesso e lanciare una *Activity* per ognuno di questi: purtroppo gli unici vertici con tale caratteristica sono i quattro vertici 'cardinali', cioè quello più in alto, quello più in basso, quello più a destra e quello più a sinistra; tuttavia, dividendo la computazione in questo modo le prestazioni sarebbero limitate in quanto lo *speed-up* rimarrebbe minore o uguale a 4. Sia la versione a memoria condivisa, sia quella a memoria distribuita, sono state implementate parallelizzando ogni iterazione, o meglio il ciclo *for* dell'algoritmo.

3.4.1 Versione a memoria condivisa

Per la compilazione è stato utilizzato il back end Java poiché rende l'esecuzione più veloce rispetto al compilatore C++, il quale ha altri vantaggi. La strategia di parallelizzazione della versione a memoria condivisa consiste nella creazione di `p` *Activity* ad ogni iterazione, ognuna delle quali opera su una partizione dell'insieme di `n` punti: all'interno di quest'ultima, ogni *Activity* trova il punto successivo all'ultimo aggiunto nel poligono convesso; successivamente, le `p` *Activity* terminano per permettere alla *root* di determinare il migliore fra i punti trovati. Il partizionamento dell'insieme di vertici viene fatto a grana grossa: ogni partizione è composta da circa $\frac{n}{p}$ punti; siccome si sta lavorando a memoria condivisa, l'istanza `pset` di `PointSet` non viene divisa, bensì ogni *Activity* opera su un determinato insieme di indici. Nella figura 3.1 viene dato un semplice esempio di partizionamento regolare a grana grossa con 4 core.

Figura 3.1: Partizionamento regolare di un vettore monodimensionale su 4 processori.

Nel caso in cui n non sia un multiplo di p, l'ultima *Activity* si prende carico degli indici rimanenti. Il listato 3.3 mostra come ogni *Activity* trova un candidato per il punto successivo dell'inviluppo convesso e lo aggiunge al rispettivo indice dell'array `candidates`.

```
finish for(id in 0..(activities-1)) {
   async {

      val actualSize:long = id != activities-1 ? localSize :
          localSize + resto;
      var localNext:long = localSize * id;

      for(j in (localSize * id)..(localSize * id + actualSize - 1)) {

           val d:Direction = turn(pset.p(cur), pset.p(localNext),
              pset.p(j));
           if ( d.isCollinear() && distanceBetween(pset.p(cur),
              pset.p(j)) > distanceBetween(pset.p(cur),
              pset.p(localNext)) ) {
            localNext = j;
         }
         if ( d.isLeft() ) {
            localNext = j;
         }
      }
      candidates.raw()(id) = localNext;
   }
}
```

Listato 3.3: Parte dell'implementazione parallela a memoria condivisa dell'algoritmo Gift Wrapping in x10.

Una volta uscite dal blocco asincrono, le *Activity* si sincronizzano grazie alla direttiva `finish`. In seguito, la *root Activity* itera l'array `candidates` per determinare quale dei vertici trovati sia effettivamente quello successivo del poligono convesso.
Un aspetto negativo dell'implementazione consiste nella generazione di `p` *Activity* ad ogni iterazione del ciclo `do while`: l'istruzione `async` ha un costo, anche

se minimo, che può introdurre dei ritardi nel programma in caso di chiamata ripetuta numerosamente. In x10, la sincronizzazione dei *thread* può avvenire con l'utilizzo della classe `Clock`; tuttavia, nell'implementazione parallela del Gift Wrapping risulta sconveniente poiché bisognerebbe sincronizzare le *Activity* due volte per ciclo: una dopo aver trovato i vertici candidati a successivi, e un'altra dopo che la *root Activity* abbia individuato l'effettivo prossimo vertice, in modo da bloccare l'inizio della iterazione seguente.

3.4.2 Versione a memoria distribuita

Per la compilazione è stato utilizzato il back end di C++ in quanto permette l'ottimizzazione delle comunicazioni remote tra *Place* con l'opzione `-OPTIMIZE_COMMUNICATIONS` da riga di comando: in tale modo, i tempi peggiorano con lo stesso dominio rispetto alla compilazione con back end Java; tuttavia, si ottengono risultati più efficienti nella valutazione delle prestazioni.
La strategia di parallelizzazione della versione a memoria distribuita è la stessa utilizzata con architettura a memoria condivisa: ad ogni iterazione vengono generate `p` *Activity* che operano su una partizione dell'insieme totale di punti, con la differenza che essi vengono distribuiti tra `p` *Place*.
In x10, esistono diversi modi per accedere alla memoria locale di un *Place* e per distribuire dei dati in remoto: uno di questi consiste nell'utilizzo dei *DistArray*, cioè array distribuiti. Nel listato 3.4 si mostra come viene effettuata la distribuzione dell'insieme di punti iniziale.

```
val blk:Dist = Dist.makeBlock(Region.make(0, pset.n-1));
val dist:DistArray[DoublePoint] = DistArray.make[DoublePoint](blk,
    ([i]:Point) => new DoublePoint(pset.p(i)) );
```

Listato 3.4: Creazione ed inizializzazione di un array distribuito

L'array distribuito è salvato nella variabile `dist` e il partizionamento dei dati viene effettuato a grana grossa e regolare grazie alla funzione `Dist.makeBlock`: la classe `Dist` mette a disposizione una serie di metodi che permettono di gestire la distribuzione dei dati arbitrariamente, e tale metodo rappresenta il caso migliore per il Gift Wrapping.
Il listato 3.5 mostra come ogni *Activity* agisce localmente in un *Place* diverso per trovare un candidato per il punto successivo dell'inviluppo convesso, per poi copiarne le coordinate in un array remoto mantenuto nel *Place* principale. Grazie al campo `home` della classe `GlobalRef` non è necessario tener traccia del *Place* nel quale è salvata l'istanza `candidates`: esso punta direttamente al *Place* remoto nel quale è stato creato l'array.

```
finish for (p in Place.places()) {
   async at(p) {

      val local = dist.getLocalPortion().raw();
      var localNext:long = 0;

      for(j in 0..(local.size-1)) {
         val d:Direction = turn(lastPoint, local(localNext), local(j));
         if ( d.isCollinear() && distanceBetween(lastPoint, local(j))
             > distanceBetween(lastPoint, local(localNext)) ) {
            localNext = j;
         }
         if ( d.isLeft() ) {
            localNext = j;
         }
      }

      val nextPoint = localNext;
      val index:long = here.id;

      at (candidates.home) {
         candidates.getLocalOrCopy().raw()(index) = local(nextPoint);
      }
   }
}
```

Listato 3.5: Parte dell'implementazione parallela a memoria distribuita dell'algoritmo Gift Wrapping in x10.

La versione a memoria distribuita non è tanto diversa da quella a memoria condivisa; invece, implementando l'algoritmo Gift Wrapping in OpenMP e MPI, è necessario cambiare la strategia di parallelizzazione in quanto le architetture sottostanti costituiscono dei limiti: x10 permette di ottenere, più o meno, la stessa implementazione di un algoritmo sia a memoria condivisa, sia a memoria distribuita, cambiandone esclusivamente la struttura.

Capitolo 4

Valutazione delle prestazioni

In questo capitolo viene analizzate le prestazioni delle due implementazioni parallele dell'algoritmo Gift Wrapping: nello specifico, vengono evidenziati lo *speed-up* e l'efficienza misurati. Le misurazioni sono state effettuate su una macchina con sistema operativo Linux che dispone di 12 core logici; perciò, sono stati presi i tempi dei programmi fino ad un massimo di 15 *Activity*, per studiare anche il caso in cui il numero di *thread* supera quello dei processori. Sia per la versione a memoria condivisa, sia per quella a memoria distribuita, l'input utilizzato rappresenta il caso pessimo per il Gift Wrapping, ovvero un insieme di punti che forma già un poligono convesso; in particolare, è stato utilizzato un numero di vertici diverso, a seconda della misurazione, il quale forma una circonferenza nel piano: è riportato un esempio nella figura 4.1.

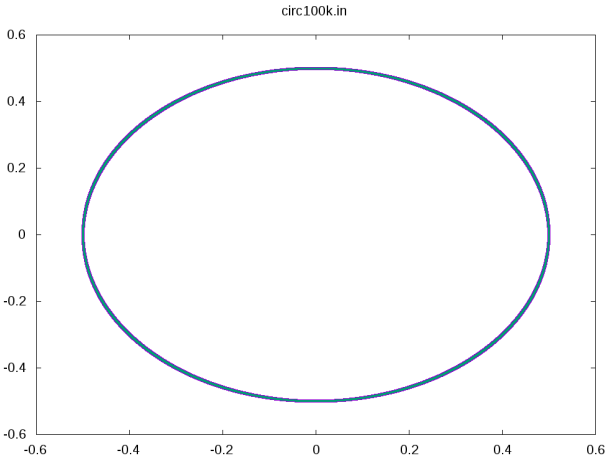

Figura 4.1: Esempio di input del caso pessimo per il Gift Wrapping

Inoltre, la misurazione dei tempi è stata automatizzata grazie all'utilizzo di

script bash: per ogni numero di *thread*, sono state osservate più esecuzioni e ne si è calcolata la media per ottenere un risultato affidabile.

4.1 Versione a memoria condivisa

Per l'implementazione a memoria condivisa sono stati utilizzati degli *script bash* che eseguono il programma con un solo *Place* ed un numero crescente di *Activity*. I risultati ottenuti vengono considerati corretti, in quanto tutti gli inviluppi convessi calcolati presentano lo stesso numero di punti e la stessa area interna. Nel listato 4.1 viene illustrato un esempio dell'output ottenuto, che mostra i risultati ottenuti dall'algoritmo per ogni esecuzione; in particolare, si mostra:

- il numero di *Activity* in esecuzione;
- il nome del file di input utilizzato;
- il numero di vertici appartenenti all'inviluppo convesso;
- l'area dell'inviluppo convesso;
- i tempi d'esecuzione in millisecondi e secondi.

```
Activities = 1
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 112685
Elapsed time (s): 112
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 122143
Elapsed time (s): 122
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 113309
Elapsed time (s): 113
```

```
Activities = 2
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 70845
Elapsed time (s): 70
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 71290
Elapsed time (s): 71
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 67621
Elapsed time (s): 67
```

Listato 4.1: Output del programma a memoria condivisa.

4.1.1 Speed-up

Per calcolare lo *speed-up*, è necessario avere i tempi d'esecuzione del programma con un numero crescente di *thread* e stesso dominio: il grafico 4.2 mostra tale misurazione, la quale è stata effettuata con un numero di *Activity* da 1 a 15 e con un input di 100.000 punti.

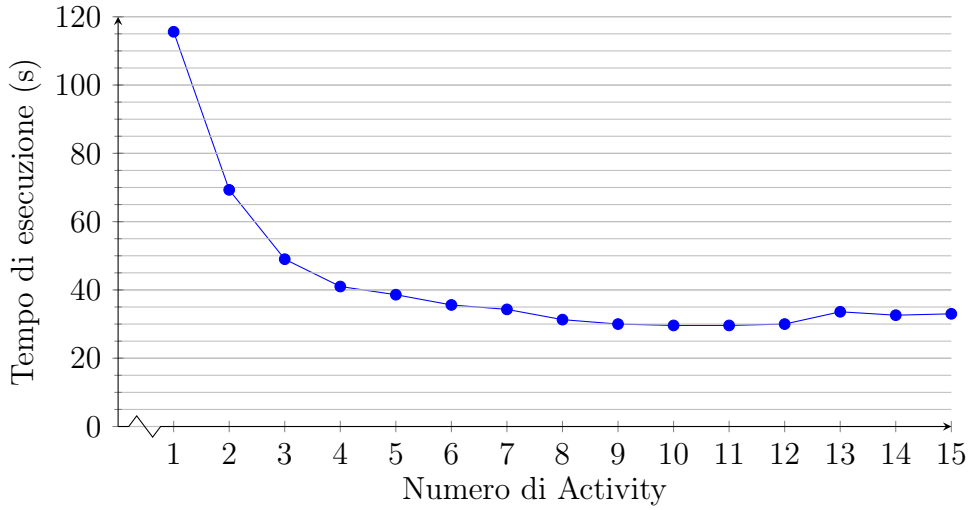

Figura 4.2: Grafico dei tempi dell'implementazione a memoria condivisa

Nel grafico 4.2 si nota un progressivo miglioramento dei tempi che si interrompe con un numero di processori p superiore a 12: questa particolarità è dovuta al superamento del numero di core logici messi a disposizione dalla macchina su cui sono state eseguite le misurazioni; poiché, in questo caso, il massimo di processori logici è 12, con un numero di *thread* maggiore o uguale a 13, quelli 'in più' dovranno attendere che i core abbiano terminato le loro computazioni prima di poter iniziare le proprie.
Con questi tempi viene calcolato lo *speed-up* mostrato nel grafico 4.3.

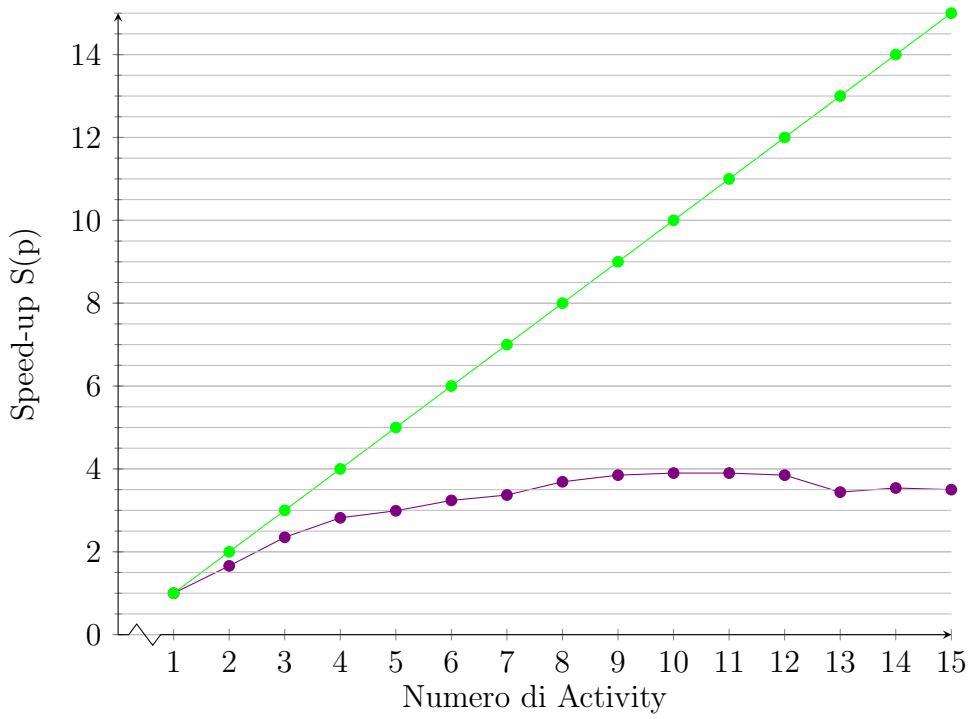

Figura 4.3: *Speed-up* dell'implementazione a memoria condivisa (viola) sui tempi in figura 4.2 a confronto con uno *speed-up* lineare (verde)

Come si può notare, lo *speed-up* ottenuto non è lineare: probabilmente tale fenomeno è dovuto alla sincronizzazione delle *Activity*; sebbene le *Activity* impieghino circa lo stesso tempo per terminare un ciclo, la maggior parte del tempo viene spesa dalle *Activity* più leggermente più veloci alla fine del blocco `finish` in attesa delle altre più lente, e l'attesa aumenta con il crescere delle *Activity*.
Un metodo per migliorare questo ritardo potrebbe essere quello di cambiare il tipo di partizione effettuata: con un partizionamento a grana grossa il tempo totale di esecuzione di una computazione diventa quello del *thread* più lento;

invece, facendo una partizione a grana fine, cioè dividendo l'input in parti più piccole, si potrebbe ridurre il tempo richiesto per la sincronizzazione.

4.1.2 Strong Scaling Efficiency

Osservando lo *speed-up* in figura 4.3, si può già prevedere che la scalabilità forte non sia ottimale. Nel grafico 4.4 viene mostrata la Strong Scaling Efficiency calcolata sullo *speed-up* precedentemente analizzato.

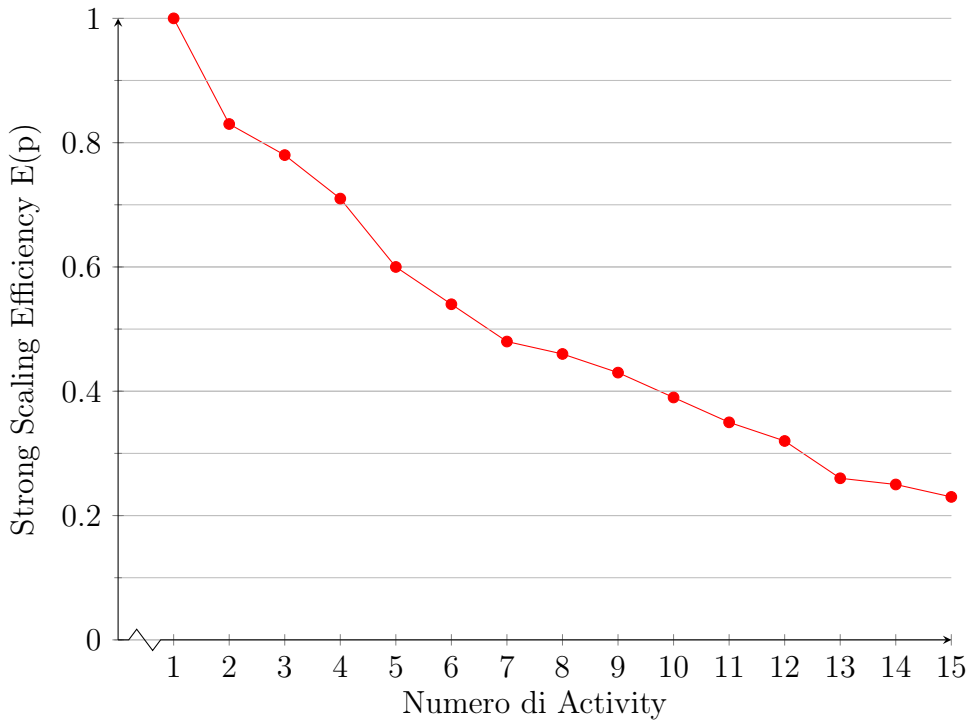

Figura 4.4: Strong Scaling Efficiency dell'implementazione a memoria condivisa

L'efficienza decresce quasi linearmente, mostrando come l'aumentare delle *Activity* rallenti la loro sincronizzazione: i tempi sono migliori rispetto alla versione seriale dell'algoritmo, ma le prestazioni ottenute non soddisfano lo scopo principale della scalabilità forte, cioè ridurre il carico assegnato ad ogni processore con l'aumentare di questi.

4.1.3 Weak Scaling Efficiency

Per calcolare la scalabilità debole è necessario effettuare un'altra valutazione dei tempi: poiché è necessario proporzionare il dominio del problema al numero

di processori, è stata creata una serie di file di input che rappresentano il caso pessimo per Gift Wrapping, ma con un numero crescente di punti. In particolare, con 1 processore l'input è costituito da 10.000 punti e con p processori è costituito da $\sqrt{p} \times 10.000$ punti; questo perchè, dato che si sta considerando il caso pessimo con complessità $O(n^2)$, il lavoro di ogni *Activity* è $\frac{n^2}{p}$, e per mantenerlo costante bisogna utilizzare $\sqrt{p} \times 10.000$ punti ($\frac{(\sqrt{p} \times 10.000)^2}{p} = 10.000^2$). La misurazione è mostrata nel grafico 4.5.

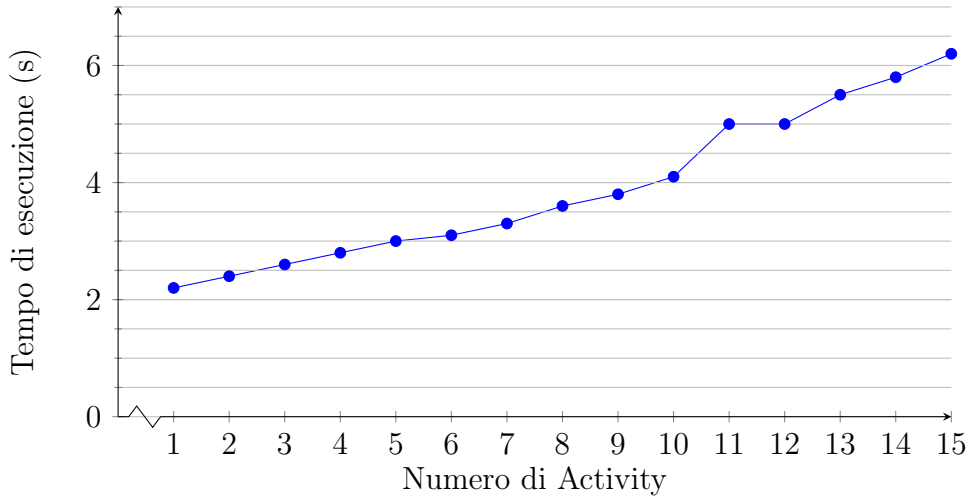

Figura 4.5: Grafico dei tempi dell'implementazione a memoria condivisa per la scalabilità debole

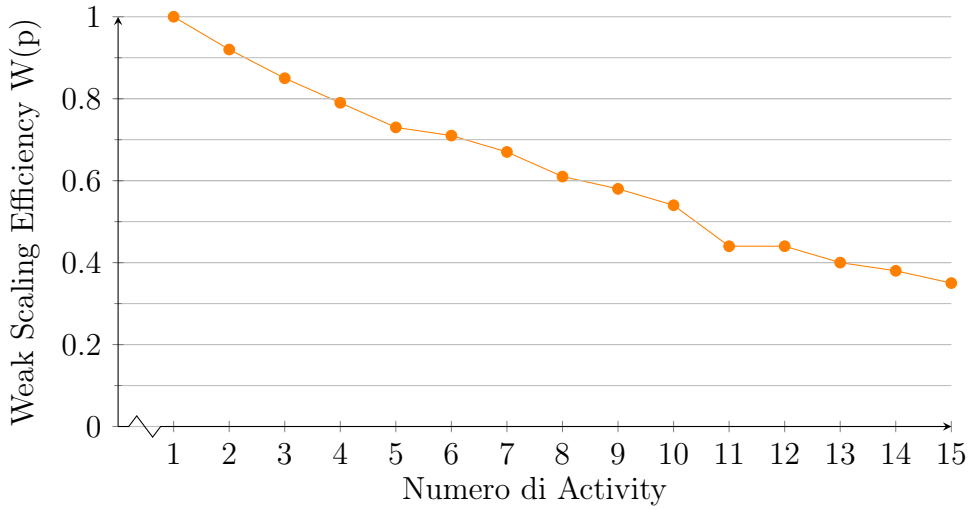

Figura 4.6: Weak Scaling Efficiency dell'implementazione a memoria condivisa

Dal grafico 4.5 si nota che non viene rispettato perfettamente l'obiettivo della scalabilità debole, ovvero eseguire programmi con dominio più grande nello stesso tempo. La misurazione del tempo risulta crescere quasi linearmente, ma non ha una pendenza troppo ripida: la diretta conseguenza è un leggero calo nel calcolo dell'efficienza, la quale viene mostrata nel grafico 4.6.

4.2 Versione a memoria distribuita

Per l'implementazione a memoria distribuita sono stati utilizzati degli *script bash* che eseguono il programma con un numero crescente di *Place*: il programma crea dinamicamente una *Activity* per ogni *Place*, come mostrato nel listato 3.5, di conseguenza non è necessario gestirne il numero da riga di comando. Il tipo di output generato dagli *script* per questa implementazione è il medesimo di quella a memoria condivisa, illustrato nel listato 4.1.

4.2.1 Speed-up

Per la misurazione dei tempi, illustrata nel grafico 4.7, è stato utilizzato un input di 5.000 punti e un numero crescente di *Place* da 1 a 15.

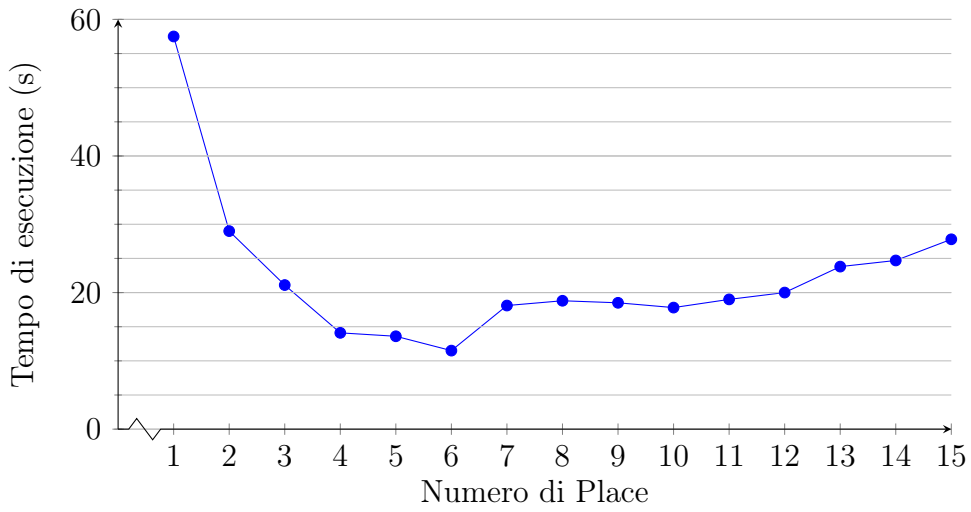

Figura 4.7: Grafico dei tempi dell'implementazione a memoria distribuita

I tempi migliorano fino ad un numero di *Place* uguale 6, per poi peggiorare leggermente: probabilmente, questo fenomeno è dato dall'aumentare dei costi di comunicazione dovuti al crescere dei *Place*; i tempi rimangono migliori della versione sequenziale, ma vengono introdotti dei ritardi mano a mano che si

aggiungono *Place* e *Activity* da sincronizzare.
Con questi tempi viene calcolato lo *speed-up* mostrato nel grafico 4.8.

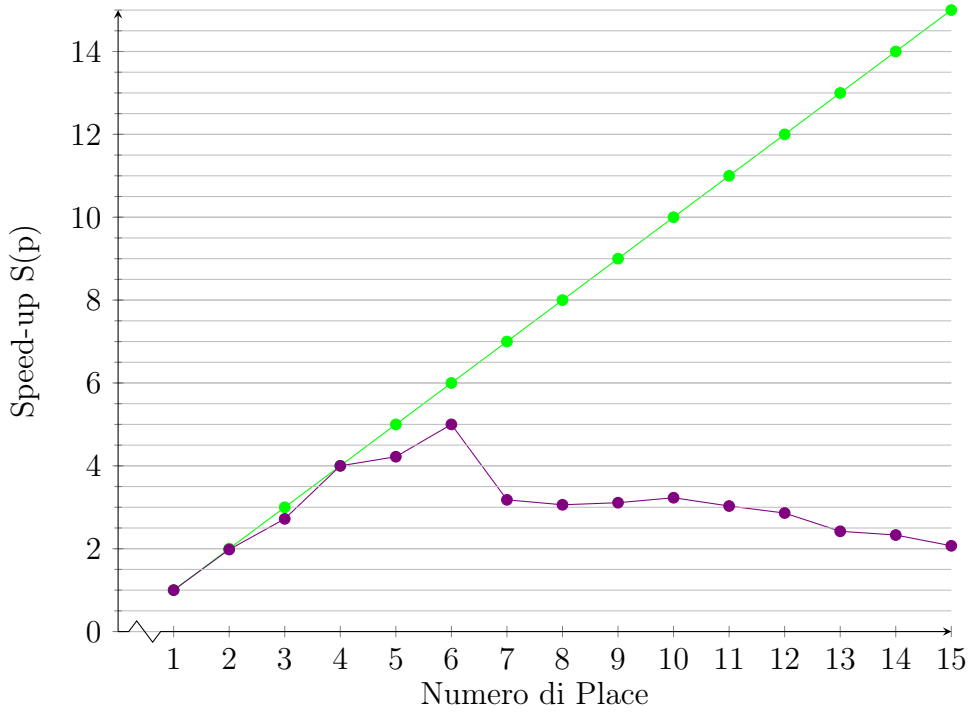

Figura 4.8: *Speed-up* dell'implementazione a memoria distribuita (viola) sui tempi in figura 4.7 a confronto con uno *speed-up* lineare (verde)

Lo *speed-up* ottenuto è quasi lineare per un numero di *Place* p fino a 4 e mantiene un buon livello fino a 6 *Place*. Come si può intuire dal grafico dei tempi, lo *speed-up* decresce con 7 o più *Place*.

4.2.2 Strong Scaling Efficiency

Osservando lo *speed-up* in figura 4.8, si può già prevedere che la scalabilità forte è ottima per un numero di *Place* inferiore a 7. Nel grafico 4.9 viene mostrata la Strong Scaling Efficiency calcolata sullo *speed-up* precedentemente analizzato.

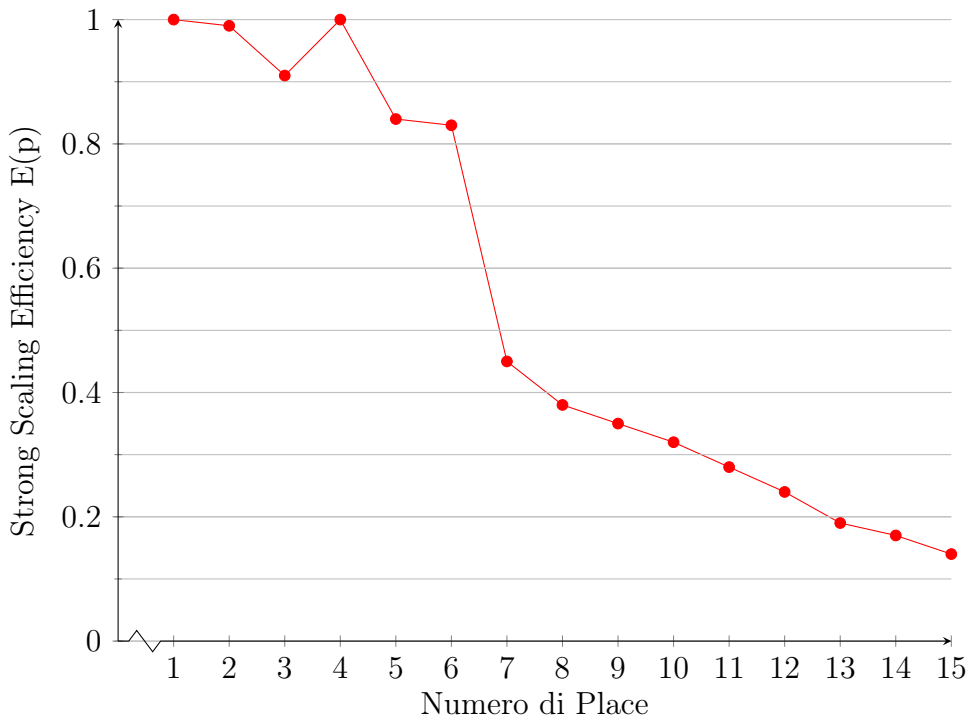

Figura 4.9: Strong Scaling Efficiency dell'implementazione a memoria distribuita

Il grafico della scalabilità forte suggerisce in maniera chiara che il ritardo introdotto dalla sincronizzazione è minimo, ma permette di soddisfare l'obiettivo della Strong Scaling Efficiency solo per un numero limitato di *Place*. Un metodo per ottenere un ulteriore miglioramento utilizzando la distribuzione in x10, potrebbe consistere nell'utilizzo di pochi *Place*, ma con molteplici *Activity* che eseguono localmente all'interno di ognuno; ad esempio, volendo sfruttare 12 core logici a pieno, si potrebbero distribuire dati tra 4 *Place* in ciascuno dei quali operano 3 *Activity*.

4.2.3 Weak Scaling Efficiency

Per calcolare la scalabilità debole è necessario effettuare un'altra valutazione dei tempi, illustrata nel grafico 4.10: la misurazione è stata fatta con un dominio crescente, come per la versione a memoria condivisa; in particolare, con 1 processore l'input è costituito da 1.000 punti e con p processori è costituito da $\sqrt{p} \times 1.000$ punti.

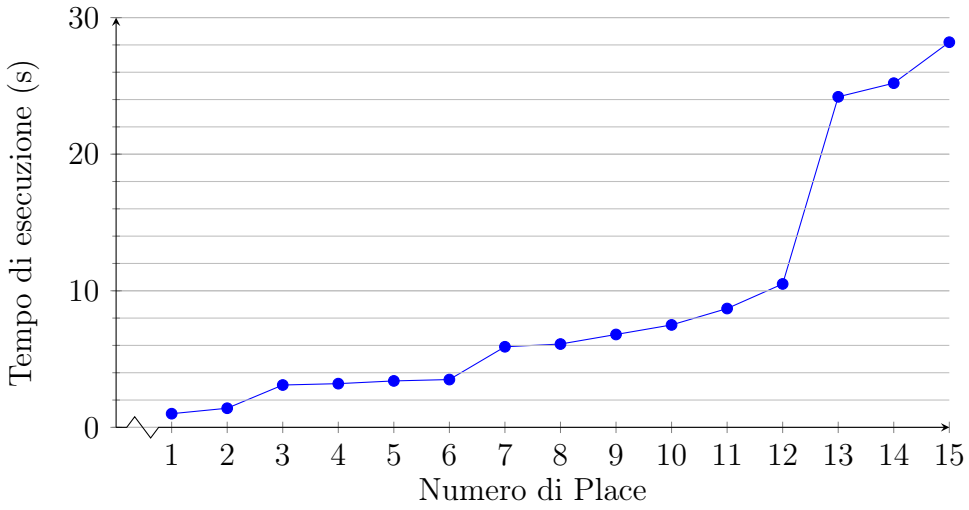

Figura 4.10: Grafico dei tempi dell'implementazione a memoria distribuita per la scalabilità debole

I tempi della scalabilità debole crescono di poco per un numero di *Place* minore o uguale a 13: l'obiettivo della Weak Scaling Efficiency non viene raggiunto neanche in questa implementazione. Per un numero di *Place* superiore a 12, i tempi salgono a picco: motivo per cui l'efficienza mostrata nel grafico 4.11 presenta un netto peggioramento sopra a tale cifra.

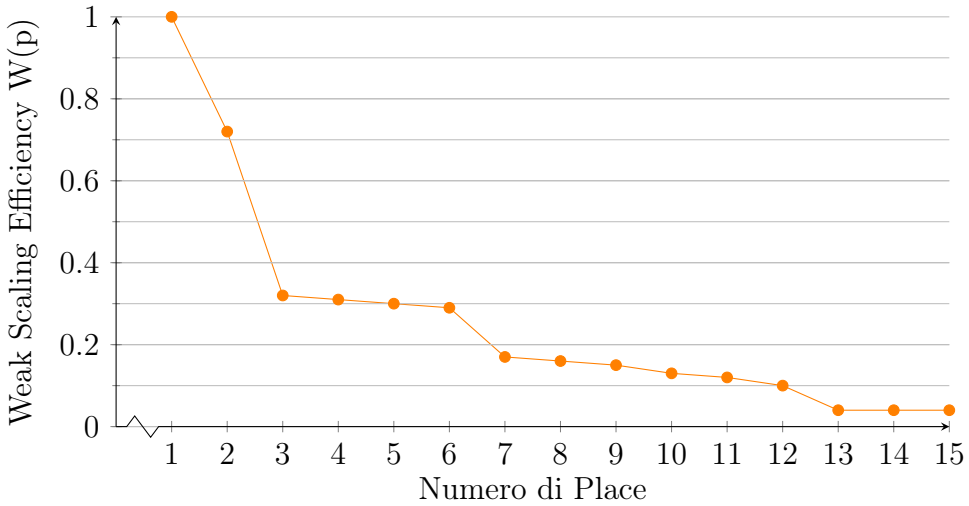

Figura 4.11: Weak Scaling Efficiency dell'implementazione a memoria distribuita

Conclusioni

In questa tesi è stato analizzato il linguaggio x10 evidenziandone le potenzialità e i contributi che può dare all'High Performance Computing grazie alle sue componenti particolari per le architetture parallele; il codice sorgente dell'elaborato è reperibile nel repository pubblico `https://bitbucket.org/SRomagnol21/parallelconvexhull/` . Questo lavoro ha preso in esempio un problema e lo ha valutato utilizzando relativamente poche risorse, ovvero 12 processori; pensare di poter applicare una miglioria come la parallelizzazione a problemi di grandi dimensioni in ambiti come l'astrofisica o il machine learning, rende tale disciplina ancora più affascinante di quanto non lo sia già.
X10 è un linguaggio ancora in via di sviluppo; ciò nonostante, presenta molte componenti utili per la parallelizzazione che permettono ai programmatori di operare in maniera produttiva, suddividendo il carico di lavoro in modo veloce e naturale. Inoltre, essendo orientato agli oggetti, può essere la base di partenza per sviluppi futuri di linguaggi o protocolli moderni che consentano di fare ulteriori passi avanti nell'ambito dell'informatica.
É importante ricordare che la scelta dell'architettura sottostante al programma sia un interrogativo fondamentale da porsi prima della progettazione di un elaborato, in quanto memoria condivisa e distribuita apportano vantaggi e svantaggi diversi a seconda del problema che si vuole affrontare: con x10 tale problematica assume un peso molto meno significativo, dato che supporta entrambe le strutture e rende semplice al programmatore l'azione di cambiare dall'una all'altra.

Ringraziamenti

Ringrazio il relatore di questo lavoro, il Prof. Moreno Marzolla, che ha acceso il mio personale interesse nei confronti di questa splendida disciplina e mi ha seguito nella svolgimento dell'elaborato, partendo dal progetto, fino alla stesura della tesi.
Ringrazio i miei genitori, che mi hanno sempre spronato a dare il massimo e mi sono sempre stati vicini durante il mio percorso di studi.
Ringrazio amici e colleghi che hanno contribuito alla stesura di questa tesi e che hanno reso quest'esperienza unica supportandomi durante l'intero percorso.
Ringrazio in particolare Chiara Zammarchi che mi ha aiutato durante i miei studi e che mi ha sostenuto con idee e consigli durante la scrittura di questa tesi.

Bibliografia

[1] IBM. *The X10 Parallel Programming Language*. 2019. URL: http://x10-lang.org/.

[2] IBM. *X10 Performance Tuning*. 2019. URL: http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html.

[3] IBM. *X10RT Implementations*. 2019. URL: http://x10-lang.org/documentation/practical-x10-programming/x10rt-implementations.html.

[4] R. A. Jarvis. «On the identification of the convex hull of a finite set of points in the plane». In: *Information Processing Letters* 2 (1973), pp. 18–21.

[5] Moreno Marzolla. «Progetto di High Performance Computing 2019-2020». In: (2019). URL: https://www.moreno.marzolla.name/teaching/HPC/convex-hull.

[6] Vijay Saraswat et al. *X10 Language Specification. Version 2.6.2*. 2019. URL: http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf.

[7] Wikipedia. *Convex hull*. 2019. URL: https://en.wikipedia.org/wiki/Convex_hull.