Università degli Studi di Udine
Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Relazione progetto per il corso di Architetture parallele
A.A. 2022/2023

Individuazione di tutti i matching per il problema dei matrimoni stabili

Stefano Travasci
Mat. 133014
travasci.stefano@spes.uniud.it

Riccardo Zamolo
Mat. 135047
zamolo.riccardo@spes.uniud.it

Indice

1 Introduzione **2**
1.1 Il problema dei matrimoni stabili . . . 2
1.2 Estensioni del problema . . . 2
1.3 Trovare tutte le soluzioni . . . 3

2 Soluzione seriale **4**
2.1 Le basi teoriche . . . 4
2.2 Descrizione generale . . . 4
2.3 Le strutture dati . . . 6
2.4 L'algoritmo seriale . . . 6
2.5 La complessità . . . 15

3 Soluzione parallela **16**
3.1 Introduzione alla soluzione parallela . . . 16
3.2 Descrizione generale . . . 17
3.3 Le strutture dati . . . 20
3.4 La memoria . . . 20
3.5 L'algoritmo parallelo . . . 21
3.6 Possibili miglioramenti . . . 25

4 Risultati sperimentali e osservazioni **27**

5 Istruzioni di compilazione **31**

Elenco delle figure

1 Contributo in percentuale dei vari sotto-algoritmi seriali al tempo totale. . . . 17
2 Comparazione dei tempi totali delle due versioni. . . . 27
3 Comparazione delle due versioni di Build Graph. . . . 28
4 Tempi medi degli algoritmi della versione seriale. . . . 29
5 Tempi medi degli algoritmi della versione parallela. . . . 29
6 Comparazione tra tempi medi di esecuzione effettiva del kernel e di overhead. . . . 30

1 Introduzione

La presente relazione ha come obiettivo quello di presentare una soluzione algoritmica seriale e una parallela (sfruttando l'estensione CUDA del linguaggio di programmazione C) per una estensione del problema dei matrimoni stabili. In particolare, si richiede di trovare tutti i possibili matching stabili e non soltanto uno solo, come invece previsto dalla forma classica del problema.

1.1 Il problema dei matrimoni stabili

Il problema dei matrimoni stabili[1] consiste nel trovare un **matching stabile** tra due insiemi aventi la stessa cardinalità, basandosi sulle preferenze espresse da ciascun elemento dei due insiemi. Un **matching**, nel contesto del problema in esame, consiste nella creazione di una serie di accoppiamenti tra insiemi del primo e del secondo insieme. Il numero di coppie presenti nel matching sarà uguale alla cardinalità degli insiemi. Un matching è **stabile** se non esiste alcuna coppia (A, B) tale che A preferisce B al suo attuale partner e viceversa. Se invece esiste almeno una tale coppia, allora il matching si dice instabile.

Per facilitare la comprensione del problema, si può sfruttare la metafora dei matrimoni: dati n uomini e n donne, si richiede di creare n coppie uomo-donna (matrimoni), tali per cui non esistono coppie uomo-donna che si preferiscono a vicenda rispetto ai loro attuali partner.

1.2 Estensioni del problema

Esistono svariati problemi simili a quello dei matrimoni stabili, come per esempio lo *stable roommates problem*[2], in cui l'unica differenza è che gli elementi non vengono suddivisi in due insiemi, ma appartengono tutti allo stesso pool iniziale. Generalizzazioni di questo problema permettono di imporre delle condizioni ai matching, ad esempio coppie che devono necessariamente essere incluse o escluse oppure membri di un gruppo che devono essere accoppiati allo stesso elemento. Altro tipo di generalizzazione è quello che estende la cardinalità del problema da *one-to-one* (un uomo per ogni donna, e viceversa) a cardinalità di tipo *one-to-many* o *many-to-many*. Ad esempio nel *rural hospitals theorem*[3] dei dottori devono essere assegnati a degli ospedali: ogni ospedale ha più di un posto disponibile e solitamente il numero di posti disponibili è superiore a quello dei dottori; inoltre alcuni partecipanti potrebbero essere disposti ad essere assegnati solo ad un sottoinsieme dei membri dell'altro lato del matching. Si hanno ulteriori variazioni del problema a seconda che l'ordinamento delle preferenze sia stretto, per cui non è possibile che il membro di un gruppo valuti due alternative come equivalenti, oppure non stretto, quando in una lista di preferenze alcune alternative possono essere valutate come qualitativamente equivalenti; nel secondo caso si parla di *stable marriage with indifference*.[4] Nel presente testo vogliamo calcolare tutti i matching stabili relativi a un'istanza del problema classico, in cui ogni elemento fa parte di una e una sola coppia (*one-to-one*) e le preferenze sono ordinate in modo stretto.

1.3 Trovare tutte le soluzioni

In questo lavoro affrontiamo il problema di trovare tutte le possibili soluzioni ad una particolare istanza del problema dei matrimoni stabili, con le preferenze espresse con un ordine stretto.

Sappiamo da [5] che l'enumerazione di tutti i matching stabili è un problema #P-completo.[1] Essendo il conteggio dei matching stabili #P-completo, lo deve essere anche elencarli. Scendendo più in dettaglio per quanto riguarda il numero di soluzioni, [6] trova un upper bound di $O(3,55^n)$ e [7] trova dei lower bound comunque esponenziali. Ne segue che un qualsiasi algoritmo che trovi tutte le soluzioni debba impiegare un tempo almeno esponenziale.

Fra i lavori precedenti che hanno trattato il nostro stesso problema, menzioniamo [8], [9] e [10]. [11], [12] e [13] si occupano invece di trovare tutte le soluzioni in variazioni del problema originale.

Si noti che alcuni lavori, come [14], si occupano di trovare una rappresentazione implicita dei risultati attraverso il grafo delle rotazioni. In questo lavoro faremo anche il passo successivo, ricavando l'elenco esplicito di tutti i matching stabili.

[1] Da [5]: "*The concept of #P-completeness was introduced by Valiant [...] in order to describe those enumeration problems that are "at least as hard" as NP-complete problems.*"

2 Soluzione seriale

Nella sezione corrente si presentano prima le nozioni teoriche e la soluzione algoritmica seriale ad alto livello. Quindi sono introdotte le strutture dati utilizzate e, infine, lo pseudocodice della soluzione.

2.1 Le basi teoriche

L'insieme delle soluzioni al problema dei matrimoni stabili può essere articolato in un reticolo (*lattice of stable matchings*) i cui elementi sono le singole soluzioni.[15] Una rotazione è l'insieme dei cambiamenti necessari per trasformare un matching in un altro matching ad esso adiacente nel reticolo. Inoltre, dato un matching, una rotazione si dice *esposta* qualora sia possibile applicarla al matching. In ogni matching stabile, una rotazione può essere esposta o meno. Si considerino ad esempio i seguenti matching stabili M_i e M_{i+1}

$$M_i = \Big\{(1,2),(2,3),(3,1),(4,4)\Big\}$$

$$M_{i+1} = \Big\{(1,3),(2,2),(3,1),(4,4)\Big\}$$

I cambiamenti necessari per passare da M_i a M_{i+1} costituiscono una rotazione ρ, così definita

$$\rho = (1,2),(2,3)$$

I matching in cima e in fondo al reticolo dei matching stabili sono, rispettivamente, la soluzione ottimale per gli uomini (e peggiore per le donne) e quella ottimale per le donne (e peggiore per gli uomini). Entrambe possono essere calcolate in tempo $O(n^2)$ usando l'algoritmo di Gale–Shapley.[16] Per calcolare il matching in fondo è sufficiente invertire il ruolo degli uomini e delle donne nell'algoritmo di Gale–Shapley. Si noti che, a causa dell'inversione dell'ordine dei parametri, se il matching è rappresentato come un vettore, il significato di indici e valori sarà invertito; è possibile facilmente invertire nuovamente indici e valori in tempo $O(n)$.

2.2 Descrizione generale

L'idea alla base del nostro algoritmo è partire dal matching migliore per gli uomini ed esplorare il reticolo delle soluzioni dall'alto verso il basso attraverso una depth-first search, spostandoci ogni volta di una singola rotazione.

Per fare ciò in modo efficiente, troviamo tutte le rotazioni e successivamente, costruiamo un grafo che codifica le dipendenze fra le rotazioni. Per compiere entrambi gli step, seguiamo gli algoritmi proposti in [10] e [17]. In particolare, per trovare le rotazioni è stato implementato l'algoritmo descritto ad alto livello nel terzo paragrafo di [10]. Per costruire il grafo delle rotazioni ci si è basati sull'algoritmo descritto a pagina 112 di [17][2], applicando una correzione. Come spiegato in [14], infatti, l'algoritmo lì proposto contiene un errore. [14] propone un algoritmo alternativo, ma nell'appendice B identifica e propone una soluzione per l'algoritmo

[2]Lo stesso algoritmo è descritto anche in [10], ma la spiegazione è molto più comprensibile nel libro.

originale; pur trovando correttamente il punto in cui l'algoritmo di [17] fallisce, la natura esatta dell'errore non è compresa correttamente, e dunque la soluzione proposta non è adeguata.

Per risolvere questo errore, ci siamo basati sulla definizione di predecessori data in [14] (*Definition 4.4*)[3]. Studiando più accuratamente questa definizione, risulta evidente che l'algoritmo di [17] verifica solo la prime delle due condizioni necessarie per identificare i predecessori di tipo 2. Dunque, per correggere l'algoritmo, abbiamo aggiunto, nella fase iniziale di etichettatura, una nuova sequenza di etichettature: così come nell'algoritmo originale scansioniamo la lista delle preferenze delle donne per aggiungere le etichette di tipo due, facciamo lo stesso per la lista di preferenze degli uomini. Nel passo successivo, verranno aggiunti solo gli archi di tipo due che sono stati etichettati due volte, ovvero sia scandendo la lista delle preferenze delle donne che quella degli uomini. Si noti che ogni coppia non stabile può essere etichettata al più due volte, una per ogni lista. Questa piccola modifica permette di aggiungere gli archi di tipo 2 che effettivamente rispettano entrambi i requisiti.

Il grafo che costruiamo è simile a quello descritto nel terzo paragrafo di [18], con una piccola differenza: mentre quel grafo rappresenta la relazione di *predecessore immediato* (*immediate predecessor*) tra le rotazioni, nel nostro grafo alcuni archi sono ripetuti più volte. Questi archi possono essere facilmente rimossi, con una piccola modifica alla nostra soluzione. Si può facilmente notare che il grafo dei *predecessori immediati* in [18] è un sottografo di quello creato qui (in particolare, ha gli stessi nodi e un sottoinsieme degli archi). Entrambi i grafi sono funzionali per gli scopi di questo lavoro; mentre quello più piccolo permetterebbe successivamente di eseguire meno operazioni, quello più grande è più facile da calcolare e il numero maggiore di archi non aumenta la complessità asintotica dell'algoritmo.

La ricerca effettiva di tutte le soluzioni è diversa da quella in [10], pur ispirandosi al concetto di ordinamento delle rotazioni lì presentato. Nonostante le differenze, i due approcci sono tuttavia alquanto simili e fondamentalmente equivalenti.

Ogni chiamata di `RECURSIVE_SEARCH` riceve come input un matching e una lista di rotazioni esposte rispetto a quel matching. Per ogni rotazione ρ nella lista di rotazioni, essa viene applicata singolarmente al matching di input, ottenendo così un nuovo matching. Quindi viene effettuata una chiamata ricorsiva sul nuovo matching e sulla parte della lista di rotazioni successiva a ρ, a cui vengono aggiunte eventuali rotazioni rese esposte dall'applicazione di ρ; il significato di questo passo è il seguente: dopo aver trovato un nuovo matching applicando una rotazione ρ, si trovano tutti i matching ottenibili applicando le rotazioni che seguono ρ nella lista delle rotazioni originale e le rotazioni che sono diventate esposte dopo l'applicazione di ρ. Si noti che non solo in questo modo vengono trovate tutti i possibili matching, ma che vengono inoltre trovati tutti una sola volta, rendendo superfluo controllare se il nuovo matching era stato già trovato in precedenza. Infatti, essendo escluse dalla chiamata ricorsiva le eventuali rotazioni che precedono la rotazione applicata nella lista, i successori non ancora esposti posso essere resi tali solo applicando i suoi predecessori nello stesso ordine in cui appaiono nella lista delle rotazioni. In altre parole, perché siano applicati tutti i predecessori, allora essi devono essere applicati nell'ordine definito dalla lista: ad esempio se applicassimo il secondo predecessore senza aver applicato il primo, nelle chiamate ricorsive successive il primo predecessore non sarà mai nella lista, dunque non sarà mai applicato e il successore non diventerà mai esposto.

[3]Una definizione equivalente è data in [17], ma con una formulazione di difficile comprensione. È plausibile che proprio la formulazione poco chiara di questa definizione sia stata l'origine dell'errore nell'algoritmo.

Partendo dal matching migliore per gli uomini e dalla lista delle rotazioni nel grafo che non hanno predecessori, viene così effettuata una visita DFS che definisce implicitamente un albero di copertura del reticolo dei matrimoni stabili.

Si noti infine il gran numero di parametri delle funzioni: molti di essi non forniscono dati alle funzioni, ma passano delle strutture dati create in precedenza, permettendo di diminuire il consumo di memoria e soprattutto il numero di operazioni sulla memoria.

2.3 Le strutture dati

In questo paragrafo presentiamo le strutture dati principali usate nello pseudocodice. In particolare, per quanto riguarda le liste, le distinguiamo in due categorie: le liste "semplici" e le liste "complesse". Le liste "semplici" sono composte esclusivamente dagli elementi della lista, che contengono solo il proprio valore e un puntatore al prossimo elemento della lista (o in caso a `NULL`). Le liste "complesse" invece permettono di accedere in tempo costante sia al primo che all'ultimo elemento della lista, in modo da rendere costante l'operazione di `append`; dal punto di vista realizzativo, una implementazione plausibile prevede l'utilizzo di liste "semplici" per costruire le liste "complesse", che possono contenere semplicemente un puntatore al primo e all'ultimo elemento della lista. L'utilizzo di questi due tipi di liste permette di migliorare l'efficienza di alcune operazioni senza complicare eccessivamente il resto del codice.

Di seguito si presenta un elenco più dettagliato di tutte le principali strutture dati utilizzate nello pseudocodice:

- **results_list** è la struttura dati che contiene tutte le soluzioni del problema, ovvero tutti i matching stabili. Si tratta di una lista "complessa" di puntatori ad array di interi, dove ogni array rappresenta un matching stabile (indici = uomini, valori = donne).

- **list_el**: è una lista "semplice" contente una rotazione, ogni elemento della lista (rotazione) è una coppia di interi uomo-donna.

- **rotation_node**: è il nodo del grafo che rappresenta una rotazione. Ogni nodo è costituito dai campi: `rotation` - un puntatore alla lista della rotazione, `index` - l'indice intero univoco della rotazione, utilizzato per ridurre, con opportuni controlli, il numero di archi nel grafo delle rotazioni, `missing_predecessors` - intero che tiene traccia dei predecessori mancanti, `successors` - la lista ("semplice") dei successori di questa rotazione.

- **free_rotations_list**: è la lista di rotazioni esposte rispetto a un particolare matching, ovvero la lista di nodi del grafo che non hanno predecessori. In termini implementativi, è una lista di puntatori a nodi del grafo. A seconda della funzione in cui appare, può essere usata come lista "complessa" o come lista "semplice", utilizzando direttamente gli elementi della lista.

2.4 L'algoritmo seriale

Di seguito si presenta lo pseudocodice seriale per l'enumerazione di tutti i matching stabili. Si noti che nello pseudocodice non vengono riportate le righe inerenti alla liberazione della memoria allocata delle strutture dati.

Algorithm 1: ALL_STABLE_MATCHINGS

```
Data: n, men_preferences, women_preferences
Result: results_list

results_list = NEW_LIST_EL()
top_matching = GALE_SHAPLEY(n, men_preferences, women_preferences)
inverted_bottom_matching = GALE_SHAPLEY(n, women_preferences, men_preferences)
bottom_matching = int[n]
for i = 0; i < n do
    bottom_matching[inverted_bottom_matching[i]] = i

//termina subito se non ci sono rotazioni
only_one_matching = true
for i = 0; i < n do
    if top_matching[i] ≠ bottom_matching[i] then
        only_one_matching = 0
        break
if only_one_matching then
    results_list.first = NEW_LIST_EL()
    results_list.first.value = top_matching
    results_list.first.next = NULL
    results_list.last = results_list.first
    return results_list

//copia top matching
top_matching_copy = int[n]
for i = 0; i < n do
    top_matching_copy[i] = top_matching[i]

//crea la lista di rotazioni
rotations_list = FIND_ALL_ROTATIONS(men_preferences, women_preferences, n, top_matching_copy, bottom_matching)

//crea il grafo delle rotazioni
BUILD_GRAPH(n, rotations_list, top_matching, men_preferences, women_preferences)

//calcolo la lista delle rotazioni libere
free_rotations_list = NEW_EMPTY_LIST()
free_rotations_list.first = NULL
free_rotations_list.last = NULL
list_el = rotations_list.first
while list_el ≠ NULL do
    if list_el.value.missing_predecessors == 0 then
        free_rotations_list.append(list_el.value)
    list_el = list.el.next

//aggiungo top matching ai risultati
results_list.first = NEW_LIST_EL()
for i = 0; i < n do // per non lavorare sul matching salvato tra i risultati
    top_matching_copy[i] = top_matching[i]
results_list.first.value = top_matching_copy
results_list.first.next = NULL
results_list.last = results_list.first

if rotations_list.first ≠ NULL then
    RECURSIVE_SEARCH(top_matching, n, free_rotations_list.first, results_list)

return results_list
```

Algorithm 2: GALE-SHAPLEY

```
Data: n, men_preferences, women_preferences
Result: matching (men = indexes, women = values)

women_partners = int[n]
men_free = bool[n]
for i = 0; i < n do
    women_partners[i] = -1
    men_free[i] = True

while True do
    //cerco il primo uomo libero m
    m = -1
    for i = 0; i < n do
        if men_free[i] then
            m = i
            break
    if m = -1 then
        break

    //itero sulle preferenze di m
    for i = 0; i < n && men_free[m] do
        w = men_preferences[m][i]
        if women_partners[w] == -1 then
            women_partners[w] = m
            men_free[m] = False
        else
            //w non libera, prendo m1 (attuale) e lo confronto con m
            m1 = women_partners[w]
            //se w preferisce m a m1, sciolgo (w,m1) e creo (w,m)
            if ACCEPT_PROPOSAL(women_preferences, n, w, m, m1) then
                women_partners[w] = m
                men_free[m] = False
                men_free[m1] = True

matching = int[n]
for i = 0; i < n; i++ do
    matching[women_partners[i]]=i
return matching
```

Algorithm 3: ACCEPT_PROPOSAL

```
Data: men_preferences, n, w, m, m1
Result: True se w accetta la proposta di m, False se preferisce rimanere con m1

for i = 0; i < n do
    if women_preferences[w][i] == m then
        return True
    if women_preferences[w][i] == m1 then
        return False
return False
```

Algorithm 4: FIND_ALL_ROTATIONS

```
Data: men_preferences, women_preferences, n, top_matching, bottom_matching
Result: rotations_list

rotations_list = NEW_EMPTY_LIST()
rotations_list.first = NULL
rotations_list.last = NULL
m_i = int[n]

marking = int[n] //-1 unmarked, n marked ma non associata, altri sono la donna precedente
rotation_index = 0 //per indicizzare le rotazioni

for j = 0; j < n do
    m_i[j] = top_matching[j]
    marking[j] = -1
men_preferences_indexes = int[n]

//escludiamo le preferenze sicuramente non presenti in un matrimonio stabile
for j = 0; j < n do
    k = 0
    while men_preferences[j][k] ≠ m_i[j] do
        k++
    men_preferences_indexes[j] = k + 1 //quelle attuali verranno scartate successivamente

old_m = 0
while True do
    //STEP 1
    m = -1
    //troviamo il primo uomo diverso tra m_i e bottom_matching
    for j = old_m; j < n do
        if m_i[j] ≠ bottom_matching[j] then
            m = j
            old_m = j
            break

    if m < 0 then // m_i == bottom_matching
        break

    //STEP 2
    w = m_i[m]
    marking[w] = n
    BREAKMARRIAGE(m_i, m, n, men_preferences, men_preferences_indexes, women_preferences, marking,
      rotations_list, &rotation_index, w)

return rotations_list
```

Algorithm 5: BREAKMARRIAGE

```
Data: M, m, n, men_preferences, men_preferences_indexes, women_preferences, marking, rotations_list, *rotation_index,
      first_woman

former_wife = M[m] //il w dell'articolo, donna con con cui l'uomo è accoppiato e si deve separare
reversed_M = int[n]
old_reversed_M = int[n]
for i = 0; i < n do
    reversed_M[M[i]] = i
    old_reversed_M[M[i]] = i
w, m1, breakmarriage_fail, k, old_marking
previous_woman = first_woman

men_preferences_indexes[m] += 1 //tolgo dalle preferenze la donna con cui m era fidanzato

while True do
    breakmarriage_fail = True
    //itero sulle preferenze di m
    //m diventa m' dell'articolo all'iterno del ciclo
    for i = men_preferences_indexes[m]; i < n do
        w = men_preferences[m][i] //il w dell'articolo
        //prendo m1 (attuale compagno di w) e lo confronto con m
        m1 = reversed_M[w]
        //se w preferisce m a m1, sciolgo (w,m1) e creo (w,m)
        if marking[w] < 0 && ACCEPT_PROPOSAL(women_preferences, n, w, m, m1) then // step 2a
            k = men_preferences_indexes[m] //aggiorniamo indice
            while men_preferences[m][k] ≠ w do
                k++
            men_preferences_indexes[m] = k
            reversed_M[w] = m
            marking[w] = previous_woman
            previous_woman = w
            m = m1 //nuovo uomo non accoppiato
            breakmarriage_fail = False
            break
        else if marking[w] >= 0 && ACCEPT_PROPOSAL(women_preferences, n, w, m, old_reversed_M[w]) then
         // step 2b
            breakmarriage_fail = False
            old_marking = marking[w]
            reversed_M[w] = m
            PAUSE_BREAKMARRIAGE(marking, M, reversed_M, old_reversed_M, rotations_list, w, previous_woman,
              rotation_index)
            reversed_M[w] = m1

            k = men_preferences_indexes[m] //aggiorniamo indice
            while men_preferences[w][i] ≠ w do
                k++
            men_preferences_indexes[m] = k

            if former_wife == w then // 3c:  w = w'
                reversed_M[w] = m
                return //al passo 1
            else // 3d
                if !ACCEPT_PROPOSAL(women_preferences, n, w, m, m1) then
                    marking[w] = old_marking
                    previous_woman = w
                    continue //al passo 2
                else
                    reversed_M[w] = m
                    m = m1
                    previous_woman = M[m]
                    break
    if breakmarriage_fail then // non abbiamo trovato un accoppiamento stabile per m
        return
```

Algorithm 6: PAUSE_BREAKMARRIAGE

Data: marking, M, reversed_M, old_reversed_M, rotations_list, w, previous_woman, *rotation_index

```
prev_list_el = NULL
w2 = w
go_on = True
rotation_node = NEW_NODE()
rotation_node.missing_predecessors = 0
rotation_node.index = *rotation_index
rotation_node.successors = NULL
*rotation_index += 1
list_el = NULL

while w2 ≠ w || go_on do
    go_on = False
    //costruire lista della rotazione dalla coda alla testa
    list_el = NEW_LIST_EL()
    list_el.man = old_reversed_M[w2]
    list_el.woman = w2
    list_el.next = prev_list_el
    prev_list_el = list_el

    //aggiorniamo old_reversed_M (i = i + 1)
    old_reversed_M[w2] = reversed_M[w2]
    //aggiorna M
    M[old_reversed_M[w2]] = w2
    //resettare marking
    marking[w2] = -1
    //update previous_woman
    w2 = previous_woman
    previous_woman = marking[w2]

rotation_node.rotation = prev_list_el
rotations_list.append(rotation_node)

return
```

Algorithm 7: BUILD_GRAPH

```
Data: n, rotations_list, top_matching, men_preferences, women_preferences

rotation_list_element = rotations_list.first
label_matrix = *List_of_rotations[n][n]
is_stable_matrix = bool[n][n]
label_second_condition = bool[n][n]
last_labelled_woman_index = int[n]
last_labelled_man_index = int[n]
applied_rotations = int[rotations_list.last.index + 1]
* pair_list_element
man, woman, index, k, first_woman, next_woman
* p_star
* new_successor
for i = 0; i < n do
    for j = 0; j < n do
        label_matrix[i][j] = NULL
        is_stable_matrix[j][i] = False
        label_second_condition[j][i] = False
    woman = top_matching[i]
    last_labelled_woman_index[woman]= n - 1
    while women_preferences[woman][last_labelled_woman_index[woman]] ≠ i do
        last_labelled_woman_index[woman]−−
    last_labelled_man_index[i] = 0
    while men_preferences[i][last_labelled_man_index[i]] ≠ woman do
        last_labelled_man_index[i]++
    is_stable_matrix[top_matching[i]][i] = True

for i = 0; i < rotations_list.last.index + 1 do
    applied_rotations[i] = False

while rotation_list_element ≠ NULL do
    pair_list_element = rotation_list_element.value.rotation
    first_woman = pair_list_element.woman
    while pair_list_element ≠ NULL do
        man = pair_list_element.man
        if pair_list_element.next == NULL then
            next_woman = first_woman
        else
            next_woman = pair_list_element.next.woman
        woman = pair_list_element.woman
        //aggiorna rispetto alla donna
        index = last_labelled_woman_index[next_woman]-1
        while women_preferences[next_woman][index] ≠ man do
            label_matrix[][] = rotations_list_element.value
            index−−
        last_labelled_woman_index[next_woman] = index
        //aggiorna rispetto all'uomo
        index = last_labelled_man_index[man] + 1
        while men_preferences[man][index] ≠ next_woman do
            label_second_condition[man][men_preferences[man][index]] = True
            index++
        last_labelled_man_index[man] = index

        is_stable_matrix[next_woman][man] = True
        label_matrix[woman][man] = rotations_list_element.value
        pair_list_element = pair_list_element.next
    rotations_list_element = rotations_list_element.next
```

continua...

```
for m = 0; m < n do
    k = 0
    while top_matching[m] ≠ men_preferences[m][k] do
        k++
    p_star = NULL
    for j = k; j < n do
        woman = men_preferences[m][j]  if label_matrix[woman][m] == NULL then
            continue
        if is_stable_matrix[woman][m] then // label di tipo 1
            if p ≠ NULL then
                new_successor = NEW_LIST_EL()
                new_successor.next = p.successors
                new_successor.value = label_matrix[woman][m]
                p_star.successors = new_successor
                label_matrix[woman][m].missing_predecessors++
            p_star = label_matrix[woman][m]
            applied_rotations[label_matrix[woman][m].index] = True
        else if !applied_rotations[label_matrix[woman][m].index]) && label_second_condition[m][woman] then
          // label di tipo 2
            new_successor = NEW_LIST_EL()
            new_successor.next = label_matrix[woman][m].successors
            new_successor.value = p_star
            label_matrix[woman][m].successors = new_successor
            applied_rotations[label_matrix[woman][m].index] = True
            p_star.missing_predecessors++
    //resettare applied_rotations
    for j = k; j < n do
        woman = men_preferences[m][j]
        if label_matrix[woman][m] == NULL then
            continue
        applied_rotations[label_matrix[woman][m].index] = False
```

Algorithm 8: RECURSIVE_SEARCH

```
Data: matching, n, free_rotations_list, results_list

while free_rotations_list ≠ NULL do
    //applica la rotazione
    rotation = free_rotations_list.value.rotation

    first_woman = rotation.woman
    while rotation.next ≠ NULL do
        matching[rotation.man] = rotation.next.woman
        rotation = rotation.next
    matching[rotation.man] = first_woman

    //aggiungo il matching ai risultati
    new_matching = int[n]
    for i = 0; i < n do
        new_matching[i] = matching[i]
    results_list.append(new_matching)

    //creo la nuova lista delle rotazioni per la chiamata ricorsiva
    //tolgo la rotazione applicata e aggiungo le nuove rotazioni
    first_new_rotation = NULL
    new_free_rotations_list = free_rotations_list.next
    successors_list = free_rotations_list.value.successors
    while successors_list ≠ NULL do
        successor = successors_list.value
        successor.missing_predecessors − = 1
        if successor.missing_predecessors == 0 then
            //aggiungi questa rotazione in cima lista
            new_list_el = NEW_LIST_EL()
            new_list_el.value = successor
            new_list_el.next = new_free_rotations_list
            new_free_rotations_list = new_list_el
            if first_new_rotation == NULL then // per liberare successivamente la memoria
                first_new_rotation = new_list_el
        successors_list = successors_list.next

    //chiamata ricorsiva
    RECURSIVE_SEARCH(matching, n, new_free_rotations_list, results_list)

    //ripristino le rotazioni
    //ripristino i missing_predecessors dei successori
    successors_list = free_rotations_list.value.successors
    while successors_list ≠ NULL do
        successor = successors_list.value
        successor.missing_predecessors += 1
        successors_list = successors_list.next

    //ripristina rotazione
    rotation = free_rotations_list.value.rotation
    while rotation ≠ NULL do
        matching[rotation.man] = rotation.woman
        rotation = rotation.next

    //ad ogni iterazione, togliamo una rotazione dalla lista
    free_rotations_list = free_rotations_list.next
```

2.5 La complessità

Come detto precedentemente nel paragrafo 1.3, il numero di soluzioni è esponenziale, dunque la complessità temporale del nostro algoritmo dovrà essere almeno esponenziale.

Per calcolare la complessità della nostra soluzione, esaminiamo la complessità dei suoi componenti, osservando cioè le varie chiamate all'interno di `ALL_STABLE_MATCHINGS`. Le chiamate a `GALE_SHAPLEY` per trovare il top e il bottom matching richiedono $O(n^2)$ ciascuna.[16] I due `for` presenti richiedono tempo $\Theta(n)$. `FIND_ALL_ROTATIONS` si basa sull'algoritmo delineato in [10] per calcolare tutte le rotazioni, che ha una complessità temporale pari a $O(n^2)$. La costruzione del grafo di dipendenze fra le rotazioni richiede un tempo aggiuntivo uguale a $\Theta(n^2)$.

Per poter calcolare la complessità della chiamata a `RECURSIVE_SEARCH`, notiamo che viene eseguita una singola iterazione del ciclo `while` e un singola chiamata a `RECURSIVE_SEARCH` per ogni soluzione. Fa eccezione il top matching, per il quale non viene eseguita nessuna iterazione del `while`. Essendo tutte le operazioni di `RECURSIVE_SEARCH` collocate all'interno del ciclo, il costo della chiamata alla funzione per ogni matching è costante $O(1)$. Il costo di una singola iterazione del ciclo `while` è invece $O(n)$, in particolare $\Theta(n)$ a causa del salvataggio del nuovo matching. Il costo per trovare ogni nuova soluzione è dunque $O(1) + \Theta(n)$, ovvero $\Theta(n)$. Sia S l'insieme di tutte le soluzioni e $|S|$ la sua cardinalità, ne segue che la complessità di `RECURSIVE_SEARCH` è $O(|S| * n)$, che grazie a [6] sappiamo essere uguale a $O(3,55^n * n)$.

Mettendo insieme i vari componenti dell'algoritmo, otteniamo che la complessità di `ALL_STABLE_MATCHINGS` è $O(n^2) + O(n^2) + \Theta(n) + O(n^2) + \Theta(n^2) + \Theta(n) + O(|S| * n)$; essendo $|S|$ esponenziale rispetto a n, l'ultima componente domina le altre, per cui **la complessità dell'intero algoritmo è $\boldsymbol{O(|S| * n)}$ o $\boldsymbol{O(3,55^n * n)}$**.

Il costo lineare per trovare ogni soluzione è lo stesso raggiunto in [10] per enumerare tutte le soluzioni. In particolare l'algoritmo proposto in quell'articolo ha una complessità di $O(n^2 + n * |S|)$, in linea con quella di `ALL_STABLE_MATCHINGS`.

Come emergerà nelle prossime sezioni, con istanze delle dimensioni che tratteremo in questo lavoro, il numero di soluzioni sarà nell'ordine di grandezza di `n`, piuttosto che esponenziale. Per questo motivo, l'elemento quadratico $O(n^2)$ (in particolare, ma non esclusivamente, il $\Theta(n^2)$ relativo alla costruzione del grafo) sarà prevalente rispetto a quello esponenziale del calcolo di tutti i matching.

3 Soluzione parallela

In questa è presentata la proposta di parallelizzazione di una parte della soluzione seriale, con l'obiettivo di ridurre i tempi di esecuzione.

3.1 Introduzione alla soluzione parallela

Come accennato nelle ultime righe di 2.5, la parte dominante in termini di tempi è quella quadratica, in particolare quella relativa alla costruzione del grafo. Per verificare che il tempo effettivamente impiegato fosse coerente con le ipotesi derivate dal calcolo della complessità, si è deciso di valutare alcuni indici statistici degli algoritmi che costituiscono la soluzione: `GALE_SHAPLEY`, `FIND_ALL_ROTATIONS`, `BUILD_GRAPH` e `RECURSIVE_SEARCH`. Per chiarezza, si vuole far notare che i tempi medi relativi a `GALE_SHAPLEY` corrispondono a due chiamate di tale algoritmo e non a una sola. Questo perché è necessario calcolare sia il top-matching che il bottom-matching. La misurazione è stata fatta generando 8 campioni da 50 istanze pseudo-casuali ciascuno, partendo da dimensione 500 fino a 4000, con step di 500. Nella tabella 1 sono riportati i risultati ottenuti. Ogni cella presenta 4 valori: il primo è il tempo medio, il secondo è la varianza, il terzo e quarto corrispondono al 25esimo e al 75esimo percentile.

n	Gale-Shapley	Find All Rotations	Build Graph	Recursive Search
500	2.20 0.20 2.00 2.00	1.44 0.25 1.00 2.00	4.48 0.25 4.00 5.00	0.58 0.41 0.00 1.00
1000	9.40 0.61 9.00 10.00	6.72 0.65 6.00 7.00	21.92 1.50 21.00 23.00	3.28 1.51 3.00 4.00
1500	21.50 2.34 21.00 22.00	16.08 2.44 15.00 17.00	61.84 4.14 61.00 63.00	8.52 21.07 5.25 10.00
2000	38.34 8.19 36.00 40.00	30.34 5.00 29.00 32.00	122.48 17.56 120.00 125.00	16.88 36.48 12.00 19.00
2500	61.48 15.56 59.00 63.75	49.82 11.99 48.00 52.75	200.24 46.96 196.00 204.00	27.74 172.07 20.00 31.00
3000	89.76 27.12 86.00 93.00	78.62 22.18 76.00 82.00	301.96 120.98 295.00 308.75	38.86 207.18 29.00 48.75
3500	124.80 53.22 120.00 129.00	113.50 64.62 108.00 119.00	423.40 131.30 414.75 430.00	69.66 994.47 46.25 86.00
4000	164.74 79.38 161.00 169.00	153.10 104.87 149.25 158.00	565.78 187.69 557.25 574.50	82.92 957.38 63.25 94.75

Tabella 1: Tempi medi, varianza, 25esimo e 75esimo percentile degli algoritmi che costituiscono la soluzione seriale.

È evidente come, per tutte le dimensioni di input considerate, la maggior parte del tempo viene impiegata per costruire il grafo. Come è visibile nel seguente grafico, la costruzione del grafo occupa più di metà del tempo, avvicinando al 60% per $n > 1000$.

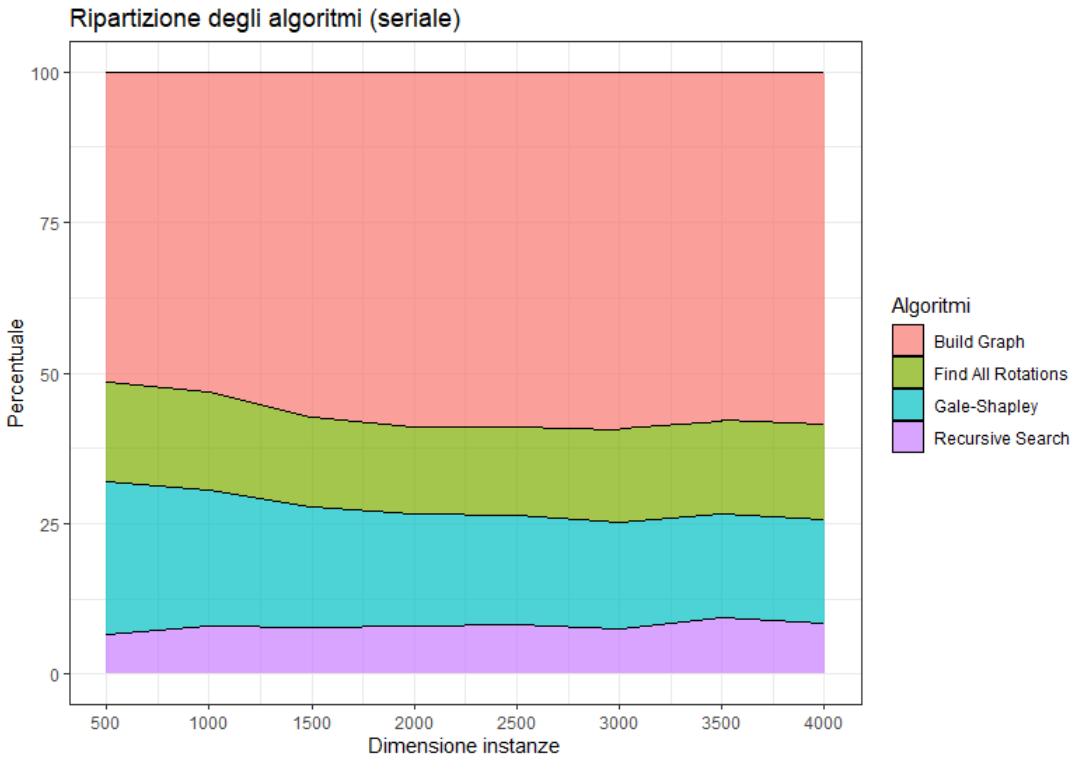

Figura 1: Contributo in percentuale dei vari sotto-algoritmi seriali al tempo totale.

Si è quindi deciso di implementare una soluzione parallela per velocizzare la parte di programma più costosa in termini di tempo, cioè la costruzione del grafo. Per le altri parti del programma, verranno mantenute le versioni seriali dei vari sotto-algoritmi.

3.2 Descrizione generale

L'algoritmo parallelo è molto simile a quello seriale, con alcune importanti differenze. In questo paragrafo, ci concentreremo soprattutto su ciò in cui la versione seriale e quella parallela differiscono.

La prima questione da affrontare è stata trovare delle strutture dati appropriate per la versione parallela. Le strutture dati utilizzate all'interno del kernel sono o uguali o molto simili a quelle della controparte seriale, essendo tutte sostanzialmente matrici. Le differenze si trovano soprattutto nelle strutture dati usate nell'input e nell'output. In particolare, sono due i problemi più importanti che abbiamo affrontato nello scegliere queste strutture dati:

- la soluzione seriale fa un grosso uso di puntatori all'interno delle strutture dati, specialmente per quanto riguarda le rotazioni. La versione parallela dovrebbe o utilizzare strutture dati equivalenti che non fanno uso di puntatori, oppure ricostruire da zero tutte le strutture dati all'interno della memoria del device; è evidente che, nella maggior parte dei casi, la prima soluzione è preferibile alla seconda.

- ridurre il consumo della memoria del device: la memoria disponibile all'interno del device è molto inferiore a quella che l'host può utilizzare. Questo è un problema importante

in quanto la nostra soluzione, e questo problema in generale, richiedono l'uso di molta memoria.

Abbiamo quindi deciso di sostituire le strutture dati presenti nella versione seriale con altre equivalenti che non fanno uso di puntatori, eliminando inoltre tutte le informazioni non strettamente necessarie a questo passo della computazione e in generale riducendo al minimo la memoria occupata. Dunque, il kernel riceve come input un vettore contiguo contenente tutte le coppie presenti nelle rotazioni e un secondo vettore per permettere di distinguere dove finiscono le coppie di una rotazione e dove iniziano quelle di un'altra. L'output del kernel è invece una semplice matrice triangolare, contenuta in un vettore, che rappresenta le relazioni di dipendenza tra le diverse rotazioni. Queste strutture dati sono spiegate in più dettaglio nel paragrafo 3.3. Si noti che, grazie all'utilizzo di questa matrice triangolare, il grafo definito da questa procedura non presenta archi ridondanti, a differenza di quello che viene trovato dalla sua controparte seriale. Va anche sottolineato però che questa differenza non è significativa in quanto, come specificato nel paragrafo 2.2, con una piccola modifica è possibile rimuovere questa ridondanza anche dalla versione seriale.

Per supportare l'uso di queste nuove strutture dati, è stato necessario aggiungere al codice host del codice che gestisca l'inizializzazione degli input e l'applicazione dell'output alla struttura dati che rappresenta il grafo nei successivi passi seriali dell'algoritmo. La complessità asintotica di queste operazioni è $O(n^2)$.

Si è deciso che tutti i thread sarebbero stati eseguiti in **un unico blocco**. La ragione è che, durante la creazione delle etichette, gli accessi alle caselle della matrice delle etichette avvengono in modo non ordinato, ma secondo le coppie presenti nelle rotazioni: non è quindi possibile suddividere il problema della costruzione del grafo in diversi problemi uguali ma più piccoli e indipendenti fra loro. Se ciò è possibile per la fase di calcolo delle dipendenze fra le rotazioni e può essere ottenuto in qualche modo nella fase di inizializzazione, non è però possibile suddividere l'insieme delle rotazioni in modo che ogni sottoinsieme coinvolga esclusivamente una sezione contigua della matrice delle etichette (lo stesso vale anche per `label_second_condition`).

Vengono lanciati tanti thread quanto il massimo tra n e il numero di rotazioni, senza comunque superare il limite di 1024 thread che possono essere lanciati nello stesso blocco. La ragione di questi numeri sarà chiara tra poco.

Il codice del kernel può essere diviso in tre parti: l'inizializzazione delle varie strutture dati ad esso interne, il calcolo della tabelle delle etichette (e di `label_second_condition`) e il calcolo delle dipendenze tra le rotazioni. Dei comandi `__syncthreads()` dividono le varie sezioni, in quanto ognuna è dipendente dal completamento di tutta la sezione precedente.

All'inizio dell'esecuzione, tutti i thread aspettano che il thread 0 allochi lo spazio per le strutture dati interne. Seguono quindi diverse fasi di inizializzazione, in cui ad ogni thread sono assegnati degli uomini/donne o delle rotazioni, a seconda della fase, rispetto ai quali inizializzare le strutture dati. Gli uomini/donne e le rotazioni sono suddivisi in modo uguale tra i diversi thread, in modo che, se il loro numero fosse maggiore di 1024, ogni thread cicli su un piccolo sottoinsieme di essi. I thread a cui non è assegnato nulla rimangono in attesa che gli altri finiscano. Vedremo che questo modo di operare sarà utilizzato anche nelle sezioni successive. Tra le varie fasi sono presenti dei comandi `__syncwarp()` per migliorare la sincronizzazione e quindi l'efficienza dell'esecuzione e da un comando `__syncthreads()` dove la mancanza di sincronizzazione potrebbe introdurre errori.

Nella seconda parte del codice del kernel, vengono riempite la matrice contenente le etichette e `label_second_condition`. Il codice è molto simile a quello seriale, riadattato in modo da funzionare con le nuove strutture dati. Ad ogni thread è assegnata una o più rotazioni, a seconda del numero di rotazioni. Il ciclo `for` principale, ovvero quello che cicla fra tutte le rotazioni assegnate al thread, contiene al suo interno dei comandi `__syncwarp()`, così assicurare la sincronizzazione tra i thread e migliorare le prestazioni. Per rendere questo possibile, anche i thread a cui in quel particolare ciclo non è assegnata alcuna rotazione entrano all'interno del ciclo `for`: grazie a degli `if` questi thread non eseguono alcuna parte della logica interna al ciclo, ma riescono ugualmente a raggiungere i comandi di sincronizzazione, evitando così *deadlock*. Per ottenere ciò, viene calcolato a priori il numero di iterazioni necessarie al ciclo. Si noti che le stesse strategie per la sincronizzazione non sono state impiegate per i cicli interni, in quanto richiederebbero una grande complicazione del codice che difficilmente porterebbe a dei miglioramenti delle prestazioni. Una importante differenza con la versione seriale riguarda il modo in cui viene riempita la matrice delle etichette. L'algoritmo seriale richiede che le rotazioni vengano scansionate in ordine, in modo che nessuna rotazione venga esaminata prima di uno dei suoi predecessori. Inoltre, nell'algoritmo seriale i vettori `last_labelled_man_index` e `last_labelled_woman_index` permettono di aggiungere la nuova etichetta partendo dall'ultima donna etichettata per quell'uomo da una delle rotazioni precedenti; questo non è possibile nella nostra soluzione parallela, perché solo un'esecuzione seriale può assicurare che tutte le etichette definite dalle rotazione precedenti siano già state calcolate. La soluzione parallela affronta questo problema scandendo la matrice delle etichette partendo sempre dalla donna associata all'uomo in questione nel top matching. La nuova etichetta viene scritta nella matrice solo se non era presente alcuna etichetta per quella coppia, o se l'etichetta associata a quella coppia identificava una rotazione successiva a quella attuale. Per evitare che accessi concorrenti introducano errori, viene usata l'operazione atomica `atomicMin()`. Date le grosse dimensioni della matrice delle etichette, è poco probabile che due thread cerchino di eseguire contemporaneamente una operazione atomica sulla stessa cella della matrice. Si noti che in questo modo vengono scansionate (e potenzialmente scritte) molte più etichette che nella versione seriale, in cui invece ogni coppia viene scansionata in questa fase al più una sola volta; la **complessità** di questa parte dell'algoritmo e, di conseguenza, di tutto l'algoritmo di creazione del grafo, diventa $O(n^3)$ invece che $O(n^2)$.[4]

Nell'ultima fase del calcolo delle dipendenze, ad ogni thread è assegnato uno o più uomini, usando le stesse strategie di sincronizzazione usate nella fase precedente. Questa sezione dell'algoritmo è probabilmente la più simile alla versione parallela. Le differenze principali sono gli adattamenti necessari per usare le nuove strutture dati e l'utilizzo di una matrice invece di un vettore per memorizzare le rotazioni già applicate rispetto ad un singolo uomo. La seconda modifica è dovuta al fatto che, a causa alla scansione concorrente delle liste di più uomini, non è più possibile usare un singolo vettore che viene resettato al termine della scansione della lista di ogni uomo.

Una volta che tutti i thread hanno completato anche l'ultima fase, il thread 0 libera la memoria allocata per le strutture dati interne.

[4]La complessità asintotica è calcolata come il numero di passi che vengono eseguiti complessivamente da tutti i thread.

3.3 Le strutture dati

In questo paragrafo sono presentate le strutture dati utilizzate nel codice parallelo.

- **triangular_matrix:** non potendo utilizzare strutture dati dinamiche nella costruzione del grafo, si è dovuto trovare una struttura dati alternativa per rappresentare le dipendenze tra le rotazioni. La rappresentazione scelta è una matrice triangolare inferiore con dimensione pari al numero di rotazioni meno 1. La matrice contiene valori booleani; il booleano nella posizione (x, y) della matrice contiene la risposta alla domanda: "*la rotazione* $y+1$ *dipende dalla rotazione* x*?*". Si noti che, grazie a questa definizione, ogni casella della matrice identifica una relazione di dipendenza effettivamente possibile. La matrice è memorizzata sotto forma di un unico vettore contiguo, che viene indicizzato secondo la formula $(y-1)*y/2+x$.[5]

  Ad esempio, la matrice triangolare per un totale di 5 rotazioni avrebbe la seguente struttura, con B che rappresenta un valore booleano:

$$\begin{matrix} B & \cdot & \cdot & \cdot \\ B & B & \cdot & \cdot \\ B & B & B & \cdot \\ B & B & B & B \end{matrix}$$

- **rotations_vector** contiene tutte le coppie di tutte le rotazioni, ordinate secondo l'indice delle rotazioni. Per evitare accessi disallineati alla memoria, la prima metà del vettore contiene solo gli uomini delle coppie delle rotazioni, mentre la seconda metà contiene le donne.

- **end_displacement_vector** contiene, per ogni rotazione, l'indice del suo ultimo uomo in `rotations_vector`. Per ottenere l'offset delle donne nel vettore, è sufficiente sommare all'indice dell'uomo in numero totale di coppie in tutte le rotazioni.

- **rotation_vector** contiene nella posizione `i` il puntatore al `RotationNode` della `i`-esima rotazione. È utilizzato per velocizzare l'applicazione dei risultati contenuti nella matrice triangolare ai `RotationNode` che rappresentano il grafo.

3.4 La memoria

Come introdotto nei paragrafi precedenti, questo algoritmo ha bisogno di molta memoria. Nelle nostre prove, al crescere di n la memoria disponibile nel dispositivo diventa insufficiente ben prima che i tempi di esecuzione diventino proibitivi. Con l'hardware a nostra disposizione, ossia una Nvidia Rtx 2080 e una Nvidia Rtx 3060, la memoria del dispositivo è sufficiente per eseguire le istanze fino a $n = 5049$: per $n > 5049$, l'allocazione delle strutture interne fallisce per mancanza di spazio e il programma termina con un errore. Per questo motivo, i confronti con la versione parallela verranno fatti solo per istanze inferiori a questo numero. Inoltre, quando verranno paragonati i tempi necessari per le due soluzioni, va sempre tenuto in mente che la

[5] Si noti che la prima rotazione ha indice 0.

dimensione della memoria della scheda video limita la dimensione massima delle istanze che possono essere eseguite con questo algoritmo. Una scheda video con una maggiore capacità di memoria sarebbe in grado di eseguire su istanze di dimensioni maggiori.

A causa di questo grande consumo di memoria, non sono state possibili grandi ottimizzazioni riguardo gli accessi alla memoria. Infatti, le strutture dati utilizzate sono di dimensioni tali da dover stare obbligatoriamente in memoria globale. Solo i puntatori alle strutture dati possono stare in una memoria diversa, oltre ovviamente alle variabili intere locali ai singoli thread. I puntatori sono stati dunque messi in memoria shared, per velocizzarne l'accesso. Un altro problema che ha impedito di effettuare grandi miglioramenti è stato il tipo di accesso ai dati effettuato dall'algoritmo: la maggior parte delle locazioni di memoria viene letto al più un paio di volte, solitamente in zone del codice lontane tra loro. Per questo motivo e per la dimensione dei dati, non è stato possibile ottimizzare l'uso della cache. Le inizializzazioni delle strutture dati sono state pensate in modo da allineare gli accessi in scrittura da memoria. Inoltre, come spiegato nel paragrafo 3.3, il vettore `rotation_vector` è stato strutturato in modo da allineare gli accessi alla memoria.

3.5 L'algoritmo parallelo

In questa sezione, l'algoritmo parallelo viene descritto ad alto livello. Per il codice CUDA effettivo, si rimanda ai file del progetto. Lo pseudocodice lato host contiene solo le parti modificate per le esigenze del codice device; il resto è da intendersi uguale alla versione seriale.

Algorithm 9: ALL_STABLE_MATCHINGS_CUDA (lato host)

Data: n, men_preferences, women_preferences

[...]

scorre la lista delle rotazioni e la lista delle coppie di ogni rotazione per calcolare il numero totale di rotazioni (`number_of_rotations`) e il numero totale di coppie nelle rotazioni (`total_number_of_pairs`)

alloca con memoria mapped uno spazio di `total_number_of_pairs` * 2 interi per `rotations_vector`, uno spazio di `number_of_rotations` interi per `end_displacement_vector` e ((`number_of_rotations`-1)*`number_of_rotations`)/2 interi per `triangular_matrix`

alloca nella memoria dell'host `number_of_rotations` puntatori a nodi delle rotazioni per `rotation_vector`

scorre nuovamente tutta la lista delle rotazioni e la lista delle coppie di ogni rotazione, così da riempire `rotations_vector`, `end_displacement_vector` e `rotation_vector`

mette nella memoria del device il top matching e le preferenze degli uomini e delle donne

`//lancio del kernel`
NumThPerBlock = min(max(number_of_rotations, n), 1024)
BUILD_GRAPH_CUDA<<<1, NumThPerBlock>>>(n, number_of_rotations, total_number_of_pairs, rotations_vector, end_displacement_vector, top_matching, women_preferences, dev_men_preferences, triangular_matrix)

for *ogni rotazione nella lista delle rotazioni* **do**
  for *ogni elemento nella riga di* `triangular_matrix` *relativa alla rotazione* **do**
    if *c'è una dipendenza* **then**
      incrementa di 1 `missing_predecessors` della rotazione corrente
      aggiungi la rotazione corrente alla lista dei successori della rotazione da cui dipende, che viene reperita indicizzando `rotation_vector`

libera la memoria

[...]

Algorithm 10: BUILD_GRAPH_CUDA (lato device)

Data: n, number_of_rotations, total_number_of_pairs, rotations_vector, end_displacement_vector, top_matching, women_preferences, men_preferences, triangular_matrix

```
//Allocazione della memoria
if threadIdx.x == 0 then
    alloca n^2 interi per label_matrix
    alloca n^2 char per is_stable_matrix
    alloca n^2 char per label_second_condition
    alloca number_of_rotations * n interi per applied_rotations
    alloca n interi per first_men_preferences_index
    alloca n interi per first_women_preferences_index
__syncthreads()

//Inizializzazione delle strutture dati interne
for i=threadIdx.x; i<n; i+=blockDim.x do
    inizializza tutti i valori della j-esima riga di label_matrix a number_of_rotations (indica l'assenza di etichette)
    inizializza tutti i valori della j-esima riga di is_stable_matrix a False
    inizializza tutti i valori della j-esima riga di label_second_condition a False
__syncthreads()

for i=threadIdx.x; i<n; i+=blockDim.x do
    is_stable_matrix[top_matching[i]][i]=True
    for j=0; j<n; j++ do
        if top_matching[j] == men_preferences[j][i] then
            first_men_preferences_index[j] = i
        if j == women_preferences[top_matching[j]][i] then
            first_women_preferences_index[top_matching[j]] = i
__syncwarp()

for i=threadIdx.x; i<number_of_rotations; i+=blockDim.x do
    inizializza tutti i valori della j-esima riga di applied_rotations a False
__syncwarp()

for i = threadIdx.x; i < ((number_of_rotations-1)*number_of_rotations)/2; i+=blockDim.x do
    triangular_matrix[i] = False;
__syncthreads()
```

continua...

```
//Calcolo delle etichette
iterations = ceiling(number_of_rotations/blockDim.x)
for i=threadIdx.x; i<iterations*blockDim.x; i+=blockDim.x do
    if i < number_of_rotations then
        usando end_displacement_vector, calcola la posizione del primo uomo della rotazione i-esima in
          rotations_vector e assegnalo a j
    __syncwarp()
    if i < number_of_rotations then
        first_woman è la donna che appare nella prima coppia della i-esima rotazione
        for j≤end_displacement_vector[i]; j++ do
            man = rotations_vector[j]
            next_woman = la donna della coppia successiva (first_woman se è stata raggiunta la fine della rotazione)
            woman = rotations_vector[total_number_of_pairs+j]

            //Aggiorna rispetto alle preferenze della donna
            k = first_women_preferences_index[next_woman]-1
            while women_preferences[next_woman][k] ≠ man do
                atomicMin(label_matrix[next_woman][women_preferences[next_woman][k]],i)
                k−
            //Aggiorna rispetto alle preferenze dell'uomo
            k = first_men_preferences_index[man]+1
            while men_preferences[man][k] ≠ next_woman do
                label_second_condition[man][men_preferences[man][k]]=True
                k++

            is_stable_matrix[next_woman][man]=True
            atomicMin(label_matrix[woman][man],i)
    __syncwarp()
__syncthreads()

//Calcolo delle dipendenze
iterations = ceiling(n/blockDim.x)
for man = threadIdx.x; man < iterations*blockDim.x; man += blockDim.x do
    if man < n then
        k = 0
        while top_matching[man] ≠ men_preferences[man][k] do
            k++
    __syncwarp()
    if man < n then
        p_star = -1
        for j = k; j < n; j++ do
            woman = men_preferences[man][j]
            if la coppia (woman,man) non è etichettata then
                continue
            //Etichetta di tipo 1
            if is_stable_matrix[woman][man] then
                if p_star ≠ -1 then
                    triangular_matrix[(label_matrix[woman][man]-1)*number_of_rotations+p_star]=True
                p_star = label_matrix[woman][man]
                applied_rotations[man][label_matrix[woman][man]] = True
            //Etichetta di tipo 2
            else if ¬applied_rotations[man][label_matrix[woman][man]] ∧ label_second_condition[man][woman] then
                triangular_matrix[p_star-1][label_matrix[woman][man]] = True
                applied_rotations[man][label_matrix[woman][man]] = True
    __syncwarp()
__syncthreads()

il thread 0 libera la memoria che aveva allocato
```

3.6 Possibili miglioramenti

Sono possibili diversi approcci per provare a migliorare le prestazioni di questo algoritmo parallelo.

In primo luogo, cambiando la rappresentazione dei dati usati dai sotto-algoritmi seriali, in modo che sia la stessa usata dal sotto-algoritmo parallelo, sarebbe possibile ridurre gli overhead per la conversione dei dati di input e output.

Per rendere possibile l'utilizzo dell'algoritmo parellelo su istanze più grandi, si potrebbero tentare nuove strategie per ridurre ulteriormente l'uso della memoria sul device. Alcune di queste strategie potrebbero ridurre l'uso della memoria a discapito della semplicità del codice, rendendolo più complesso sia in termini di righe di codice che di numero di operazioni che devono essere eseguite. Si sottolinea, in questo contesto, che CUDA non permette operazioni atomiche su tipi di dati più piccoli di 32 bit, per cui per trasformare `label_matrix` in una matrice di `short` invece che di `int` sarebbe necessario implementare una alternativa ad `atomicMin()`. Sono stati fatti dei veloci tentativi di utilizzare, per alcune strutture dati, `short` invece che `int`, senza però ottenere miglioramenti riguardo la dimensione dell'istanza più grande che il dispositivo riesce a gestire; ulteriori approfondimenti sarebbero necessari.

Per quanto sia molto difficile che porti a miglioramenti, data la grossa dimensione delle matrici rispetto a quella piuttosto limitata della cache L2, si potrebbe provare a dare maggiore priorità nella cache alle due matrici delle preferenze, essendo esse lette più frequentemente delle altre strutture dati.

Se non ci fossero le limitazioni della memoria del device che abbiamo riscontrato, si potrebbe aumentare la parallelizzazione separando la fase di riempimento della matrice delle etichette da quella di calcolo delle dipendenze. In questo modo, eseguendo le due fasi con kernel distinti, sarebbe possibile eseguire i thread in blocchi separati, in quanto non sarebbero più necessarie sincronizzazioni tra i kernel della stessa grid. Potrebbe essere necessario spostare anche la fase di inizializzazione delle strutture dati su un kernel separato. Con le limitazioni riscontrate con i device a nostra disposizione, difficilmente gli overhead potrebbero giustificare un approccio di questo tipo.

Descriviamo infine un approccio che potrebbe permettere di utilizzare questo algoritmo nonostante i limiti di memoria. Si potrebbero lanciare in successione diverse grid di thread, ad ognuna delle quali viene assegnato un determinato numero di righe adiacenti della matrice delle etichette (e di `label_second_condition`), ovvero una sezione della matrice la cui dimensione non supera lo spazio disponibile nel device. Ogni thread si occuperebbe di scansionare una delle righe assegnate al grid e le matrici triangolari calcolate dalle diverse grid andrebbero unite usando degli `OR`. Se i thread facessero tutti parte dello stesso blocco, sarebbe possibile far calcolare ad ogni grid la parte di matrice di cui ha bisogno, rieseguendo ogni volta l'algoritmo per calcolare tutta la matrice ma effettuando solo le operazioni di scrittura relative alla zona della matrice pertinente a quella grid; una alternativa, probabilmente migliore, sarebbe calcolare lato host la matrice delle etichette, in modo che ogni grid non debba ricalcolarla, il che renderebbe possibile eseguire i thread anche su blocchi diversi. Va comunque sottolineato che, a causa delle limitazioni della memoria, i miglioramenti ottenibili eseguendo i thread su più blocchi sarebbero ridotti: ad ogni thread, infatti, vengono al momento assegnate al più 5 righe ($5049/1024 = 4,93$ identifica il limite superiore 5), il che implica che il miglioramento delle prestazioni ottenuto distribuendo i thread su più blocchi non può essere superiore ad un fattore di 5.

Un ultimo approccio che si può sperimentare è di dividere le coppie delle rotazioni a seconda della zona della matrice delle etichette che influenzano, in modo da poter applicare la prima versione della variazione precedente scansionando esclusivamente gli elementi delle rotazioni che modificano la sezione della matrice delle etichette assegnata al grid.

4 Risultati sperimentali e osservazioni

In questa sezione analizziamo i risultati sperimentali che abbiamo ottenuto, confrontando la versione seriale con quella parallela, con l'ausilio di grafici esplicativi. Sono stati generati 10 campioni casuali da 50 istanze, per n da 500 a 5000 con passi da 500; entrambe le versioni sono state valutate con gli stessi campioni. Prima però occorre descrivere le specifiche del dispositivo sul quale sono stati effettuati i test. I test sono stati eseguiti sul sistema operativo Windows 11, mentre le specifiche hardware sono riassunte in tabella 2:

	AMD Ryzen 5700G	**nVidia GeForce RTX 3060**
Cores	8	3584 CUDA cores
Clock base/boost	3.8GHz/4.6GHz	1320MHz/1777MHz
FLOPS	460.7GFLOPS	12.74TFLOPS
Cache	4+16MB (L2+L3)	3MB (L2)
Memoria	32GB	12GB
Tipo memoria	DDR4	GDDR6

Tabella 2: Specifiche hardware su cui sono stati effettuati i test.

Analizziamo come prima cosa i tempi impiegati dalle versioni. I grafici 2, 3 e la tabella 3 mostrano come ci sia stata una importante diminuzione dei tempi. La versione parallela all'incirca dimezza i tempi totali della versione seriale. Invece `Build Graph` parallelo, paragonato a quello seriale, mostra per le dimensioni considerate un andamento addirittura lineare invece che quadratico, con un miglioramento di un fattore maggiore di 5 per la dimensione massima considerata.

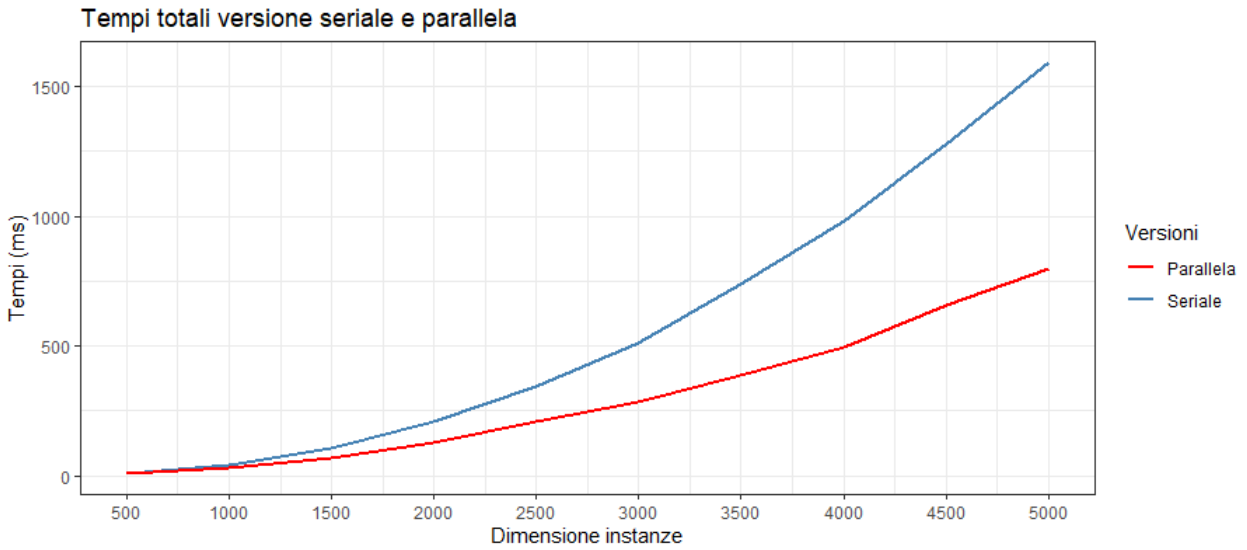

Figura 2: Comparazione dei tempi totali delle due versioni.

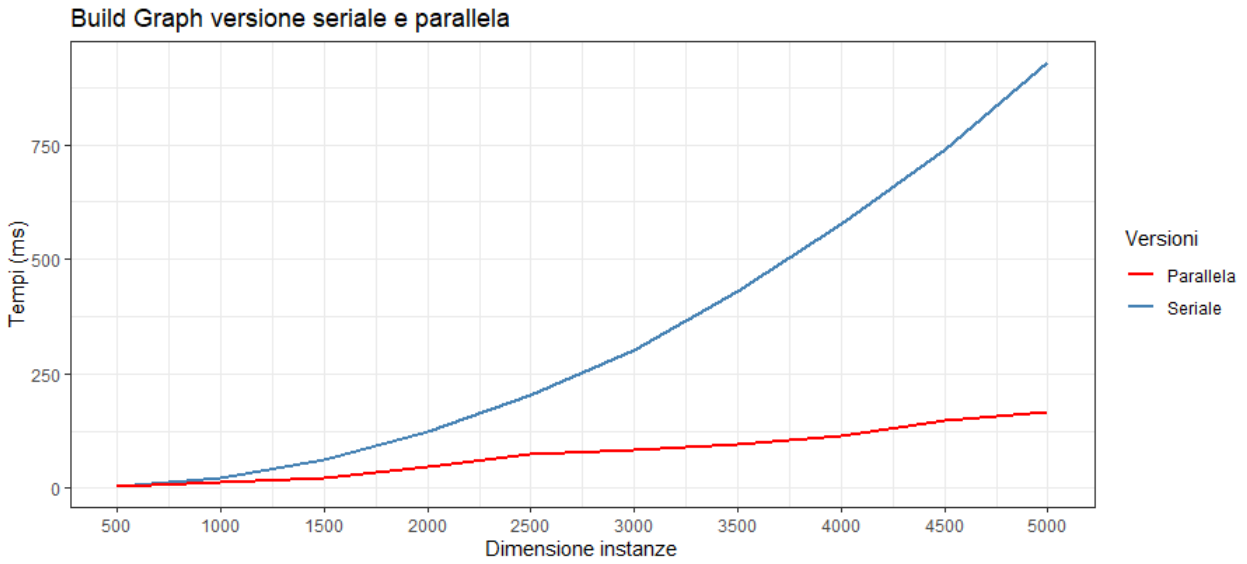

Figura 3: Comparazione delle due versioni di Build Graph.

n	Tot. seriale	Tot. parallela	Build Graph seriale	Build Graph parallela
500	8.86	7.38	4.46	4.36
1000	40.90	29.08	21.72	11.68
1500	107.42	67.24	61.68	23.78
2000	208.48	128.36	122.78	46.04
2500	343.74	208.28	203.76	74.24
3000	509.52	284.24	301.92	84.62
3500	738.70	387.96	429.28	94.48
4000	979.86	497.60	577.22	114.32
4500	1279.42	657.18	739.42	148.36
5000	1593.52	799.02	928.66	165.54

Tabella 3: Dati relativi alle misurazioni effettuate.

I prossimi due grafici mostrano il contributo dei diversi sotto-algoritmi nei tempi totali. È chiaro come, nella versione parallela, il tempo impiegato dalla costruzione del grafo sia nettamente diminuito, al punto che ora altri due sotto-algoritmi sono significativamente più lenti.

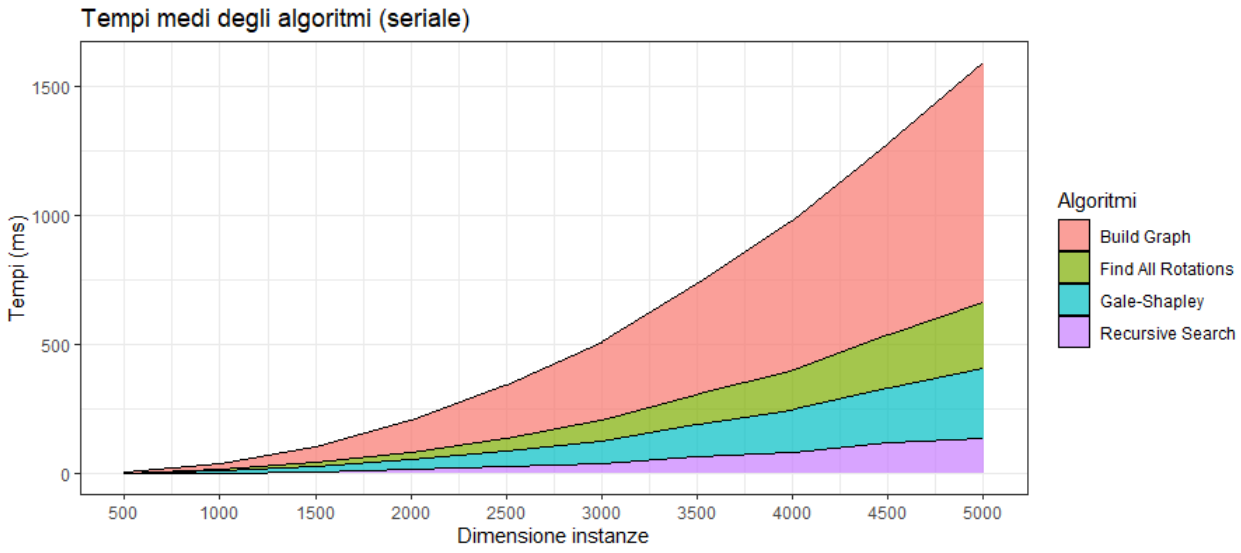

Figura 4: Tempi medi degli algoritmi della versione seriale.

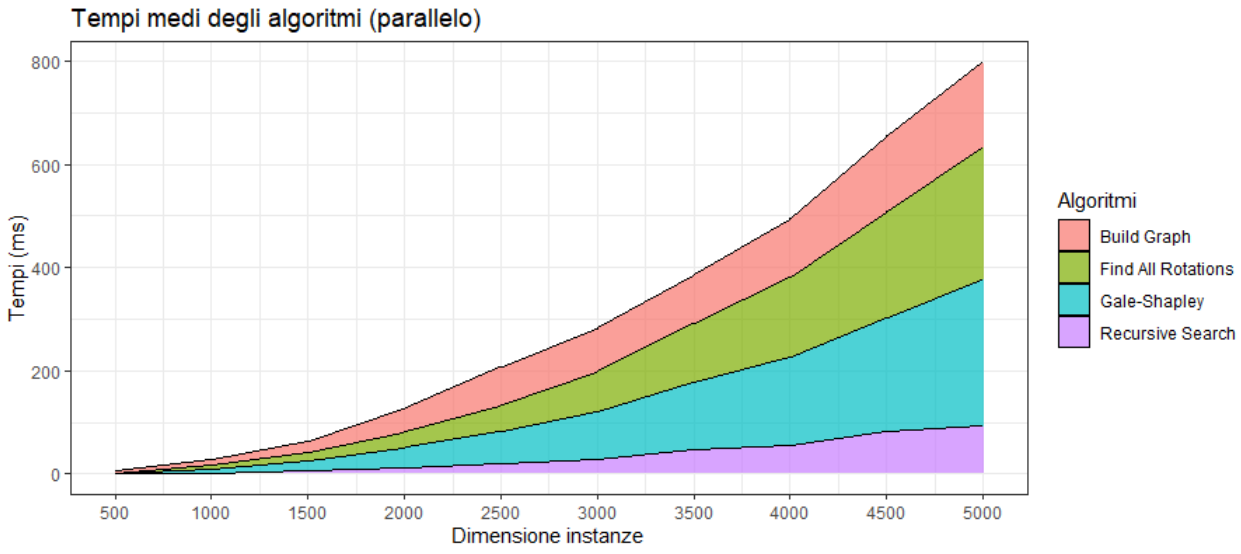

Figura 5: Tempi medi degli algoritmi della versione parallela.

Nel paragrafo 3.6 è stato accennato che l'algoritmo potrebbe essere ulteriormente velocizzato modificando i sotto-algoritmi seriali in modo che usino le stesse strutture dati dell'algoritmo parallelo. Il seguente grafico mostra come il kernel e l'overhead per la conversione e il trasferimento dei dati contribuiscono ai tempi totali per la costruzione del grafo. È evidente dal grafico che anche annullando completamente i tempi dell'overhead, cosa impossibile, il miglioramento dei tempi totali sarebbe marginale.

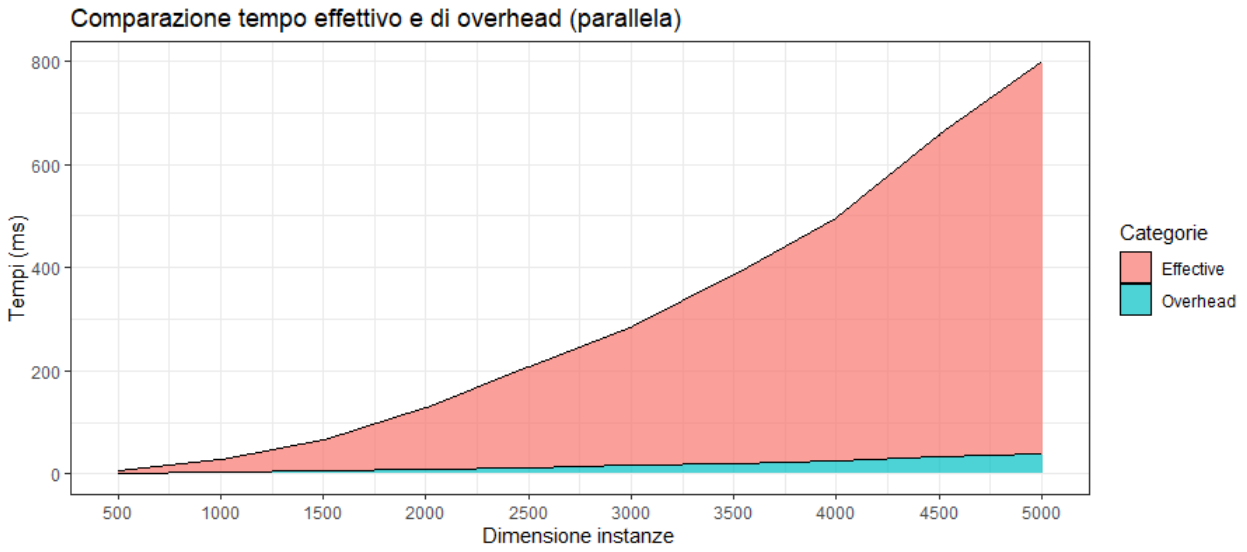

Figura 6: Comparazione tra tempi medi di esecuzione effettiva del kernel e di overhead.

Concludiamo questa sezione con una osservazione. Da tutti i campioni è stata tolta la prima misurazione dei tempi, in quanto includeva un tempo di "*warm up*" di **circa 2 secondi**. Il tempo di esecuzione di una singola istanza della versione parallela è dunque di circa 2 secondi superiore a quello appena mostrato. È possibile però migliorare le prestazioni lanciando in modo asincrono un kernel di "*warm up*", che renda la GPU immediatamente disponibile al kernel successivo.

5 Istruzioni di compilazione

Di seguito si illustrano le istruzioni per la compilazione del progetto. Vista la complessità del problema, si è reso necessario l'utilizzo di un `Makefile` per aiutare l'utente nella compilazione ed esecuzione. Inoltre, si vuole evidenziare che è stato necessario testare il progetto sotto diversi "punti di vista", per avere una misura quantomeno accettabile della correttezza. Prima di proseguire con la spiegazione dei comandi presenti nel Makefile, è doveroso fare alcune premesse:

- Il Makefile si trova all'interno della directory `\src`, pertanto è necessario posizionarsi all'interno di quella directory prima di lanciare i comandi che si illustreranno a breve.
- Il Makefile è stato scritto con sintassi per Windows; ciò significa che i comandi genereranno errori se eseguiti all'interno di ambienti diversi, come per esempio UNIX.
- Sono stati utilizzati i compilatori **gcc** per la parte seriale e **nvcc** per la parte parallela. Ogni comando del Makefile si occupa di compilare e richiamare il relativo eseguibile automaticamente.

Fatte le dovute precisazioni si può passare a descrivere ciascun comando presente all'interno del Makefile. Si noti che per questioni di spazio, gli argomenti di ogni comando verranno scritti su righe diverse, ovviamente a terminale vanno scritti di seguito separati da spazio.

Test su istanze singole: il primo passo per poter verificare la correttezza del programma è quello di testare singole specifiche istanze. Esempi di istanze di questo tipo si trovano nella directory `\tests`. All'interno si trovano file con estensione .txt contenenti in ordine: dimensione istanza (n), matrice di preferenze degli uomini e matrice di preferenze delle donne. I nomi di alcuni file sono auto-esplicativi ma in linea di massima la directory contiene istanze semplici (1, 2 o 3 soluzioni previste come output) e istanze più complesse, come quelle dei paper scientifici, grazie alle quali si è potuto verificare se i nostri output fossero uguali a quelli presentati nei paper. Nel Makefile sono presenti quindi due comandi `make` a tal proposito, uno per la parte seriale e l'altro per la parte parallela. In entrambi i casi, è necessario passare il file di input e indicare il file di output dove verranno stampati i risultati (se non presente, verrà creato automaticamente). Esempi di comandi di questo tipo possono essere:

```
make  serial  input_file=..\tests\1solution.txt
              output_file=output\serial_tests\output.txt


make  parallel  input_file=..\tests\1solution.txt
                output_file=output\parallel_tests\output.txt
```

Test su istanze multiple generate casualmente: il passo successivo è generalizzare il test su istanze singole a istanze multiple casuali. É necessario quindi specificare la dimensione di ciascuna istanza da generare, il numero di iterazioni o test da effettuare e il file di output dove verranno stampati i risultati. Inoltre, bisogna specificare una modalità di output che può essere di due tipi:

- **classic** il file .txt di output conterrà nella prima colonna il tempo necessario a calcolare le soluzioni, nella seconda colonna il numero di soluzioni trovate e nella terza la dimensione delle istanze.
- **times** Nella versione seriale il file .txt di output conterrà nelle prime 4 colonne i tempi di ogni sotto-algoritmo e nell'ultima il tempo totale per calcolare le soluzioni. Nella versione parallela, la colonna relativa ai tempi di Build Graph è sostituita dalle colonne relative ai tempi di esecuzione del kernel, di overhead e della somma dei due.

Vi è un ultimo argomento opzionale che si può passare ed è il seme dal quale verranno generate le istanze casuali. Se non viene specificato, viene utilizzato un seme di default. Anche in questo caso, vi è un comando per la versione seriale e uno per la versione parallela. Esempi di comandi di questo tipo possono essere:

```
make  tester  instance=100
              iterations=100
              mode=classic
              output_file=output\output.txt
              seed=9


make  cuda_tester  instance=100
                   iterations=100
                   mode=times
                   output_file=output\output.txt
```

Analisi dei tempi: Per effettuare le stesse misurazioni illustrate nelle sezioni precedenti, sono previsti i comandi `make serial_analysis` e `make parallel_analysis`. Come si può intuire, entrambi i comandi richiamano le rispettive versioni del tester, con argomenti impostati da noi. In particolare, le dimensioni delle istanze vanno da 500 a 5000, con step di 500. Le iterazioni sono 51, la modalità è `times` e i file di output vengono scritti rispettivamente in `\output\serial_tests\` e `\output\parallel_tests\`.

Comparazione degli output delle due versioni: come ulteriore prova di correttezza, è previsto anche un comando che si occupa di controllare che i risultati provenienti dalle due versioni siano esattamente gli stessi, sia in termini di numeri, che in termini di composizione dei singoli matching stabili. Se viene trovato un errore durante il controllo, vengono riportate su console le informazioni principali (matrici di preferenze, output seriale e parallelo) e l'esecuzione termina immediatamente. Bisogna passare la dimensione delle istanze da generare, le iterazioni ed eventualmente il seed. Il comando è

```
make  comparator  instance=100
                  iterations=50
```

demo: è stata predisposta una demo (`make demo`) funzionante del programma con parametri fissati che si occupa di effettuare chiamate multiple a solutions_comparator. In particolare,

la dimensione delle istanze vanno da 500 a 4000 con step di 500, le iterazioni sono 50 e il seed è quello di default.

Pulizia di file inutili: è previsto un ultimo comando `make clean` per la pulizia di file oggetto, eseguibili, ecc.

Si noti infine che tra i sorgenti vi è il file `\correctness_tester.c` per il quale non è stato previsto un comando nel makefile. Il motivo è che tale file è stato impiegato solamente a scopo di debug una volta terminata l'implementazione seriale. In particolare si effettua un controllo sui risultati ottenuti dall'esecuzione seriale del nostro programma con i risultati dell'implementazione naïve, la quale consiste nell'enumerazione di tutti i matching possibili (stabili e non) e per ognuno, si controlla la condizione di stabilità. Si può facilmente intuire che per motivi di tempistiche dovute alla complessità, la dimensione delle istanze testabili difficilmente supera n=12. Per compilare ed eseguire bisogna digitare i comandi

```
gcc correctness_tester.c -o correctness

.\correctness 3 5
```

Riferimenti bibliografici

[1] Wikipedia contributors, *Stable marriage problem — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Stable_marriage_problem&oldid=1183298322, [Online; accessed 8-November-2023], 2023.

[2] Wikipedia contributors, *Stable roommates problem — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Stable_roommates_problem&oldid=1171589621, [Online; accessed 15-November-2023], 2023.

[3] Wikipedia contributors, *Rural hospitals theorem — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Rural_hospitals_theorem&oldid=1064508418, [Online; accessed 16-November-2023], 2022.

[4] Wikipedia contributors, *Stable marriage with indifference — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Stable_marriage_with_indifference&oldid=1183899721, [Online; accessed 16-November-2023], 2023.

[5] R. W. Irving e P. Leather, «The complexity of counting stable marriages,» *SIAM Journal on Computing*, vol. 15, n. 3, pp. 655–667, 1986.

[6] C. Palmer e D. Pálvölgyi, «At most 3.55n stable matchings,» in *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, 2022, pp. 217–227. DOI: 10.1109/FOCS52979.2021.00029.

[7] E. G. Thurber, «Concerning the maximum number of stable matchings in the stable marriage problem,» *Discrete Mathematics*, vol. 248, n. 1, pp. 195–219, 2002, ISSN: 0012-365X. DOI: https://doi.org/10.1016/S0012-365X(01)00194-7. indirizzo: https://www.sciencedirect.com/science/article/pii/S0012365X01001947.

[8] D. G. McVitie e L. B. Wilson, «The Stable Marriage Problem,» vol. 14, n. 7, pp. 486–490, lug. 1971, ISSN: 0001-0782. DOI: 10.1145/362619.362631. indirizzo: https://doi.org/10.1145/362619.362631.

[9] D. G. McVitie e L. B. Wilson, «Algorithm 411: Three Procedures for the Stable Marriage Problem,» *Commun. ACM*, vol. 14, n. 7, pp. 491–492, lug. 1971, ISSN: 0001-0782. DOI: 10.1145/362619.362632. indirizzo: https://doi-org.altais.uniud.it/10.1145/362619.362632.

[10] D. Gusfield, «Three Fast Algorithms for Four Problems in Stable Marriage,» *SIAM Journal on Computing*, vol. 16, n. 1, pp. 111–128, 1987. DOI: 10.1137/0216010. eprint: https://doi.org/10.1137/0216010. indirizzo: https://doi.org/10.1137/0216010.

[11] V. M. Dias, G. D. da Fonseca, C. M. de Figueiredo e J. L. Szwarcfiter, «The stable marriage problem with restricted pairs,» *Theoretical Computer Science*, vol. 306, n. 1, pp. 391–405, 2003, ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(03)00319-0. indirizzo: https://www.sciencedirect.com/science/article/pii/S0304397503003190.

[12] P. Eirinakis, D. Magos, I. Mourtos e P. Miliotis, «Finding All Stable Pairs and Solutions to the Many-to-Many Stable Matching Problem,» *INFORMS Journal on Computing*, vol. 24, pp. 245–259, mag. 2012. DOI: 10.1287/ijoc.1110.0449.

[13] G. Gutin, P. R. Neary e A. Yeo, *Finding all stable matchings with assignment constraints*, 2023. arXiv: 2204.03989 [econ.TH].

[14] L. Cai e C. Thomas, *Representing All Stable Matchings by Walking a Maximal Chain*, 2019. arXiv: 1910.04401 [cs.GT].

[15] Wikipedia contributors, *Lattice of stable matchings — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Lattice_of_stable_matchings&oldid=1171759774, [Online; accessed 8-November-2023], 2023.

[16] Wikipedia contributors, *Gale–Shapley algorithm — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Gale%E2%80%93Shapley_algorithm&oldid=1182520690, [Online; accessed 8-November-2023], 2023.

[17] D. Gusfield e R. W. Irving, *The Stable Marriage Problem: Structure and Algorithms*. Cambridge, MA, USA: MIT Press, 1989, ISBN: 0262071185.

[18] R. W. Irving, P. Leather e D. Gusfield, «An efficient algorithm for the "optimal" stable marriage,» *J. ACM*, vol. 34, pp. 532–543, 1987. indirizzo: https://api.semanticscholar.org/CorpusID:8336713.