Parallel Automata Minimization

Roberto Borelli
borelli.roberto@spes.uniud.it
Stefano Rocco
rocco.stefano.1@spes.uniud.it

Sommario

Il problema di minimizzazione di un automa è centrale nella teoria degli automi e ha svariate implicazioni pratiche. In questo lavoro, tenteremo di ottenere una versione parallela del ben noto algoritmo di Moore, il quale esegue classicamente in tempo $\mathcal{O}(n^2)$. Nei capitoli 1 e 2 richiameremo le nozioni e i problemi fondamentali del campo e discuteremo l'algoritmo seriale analizzandone il codice, alcune proprietà teoriche e la complessità temporale nei casi d'interesse. Utilizzando il modello di programmazione OpenMP, nel capitolo 3, otterremo sei diverse versioni parallele dell'algoritmo. Le prime quattro versioni, più efficienti, si basano su una suddivisione del ciclo principale in task da eseguire in parallelo. Nella quinta versione, tratteremo la questione di unire più iterazioni del ciclo di raffinamento in una. Nella sesta e ultima versione, più scalabile, tenteremo un approccio basato sulla parallelizzazione di RadixSort e, in ultima analisi, di CountingSort, poi ripreso e sviluppato in CUDA nel capitolo 4. Suddivideremo CountingSort in tre fasi e per ciascuna di esse proporremo diverse soluzioni implementative utilizzando tale modello di programmazione. Le implementazioni OpenMP e CUDA dell'algoritmo di Moore saranno eseguite e testate su un insieme significativo di istanze, consentendo di mettere a confronto (capitolo 5) i due modelli di programmazione con specifica applicazione al problema di minimizzazione di un automa.

Indice

Capitolo 1

Introduzione

In questo lavoro studieremo il problema di minimizzazione di automi su architetture parallele. In questa sezione richiamiamo le nozioni basi di teoria degli automi. Vediamo la definizione di automa, di linguaggio accettato e la relazione di Myhill-Nerode e definiamo formalmente il problema di minimizzazione.

Automi a stati finiti

Un automa deterministico a stati finiti (DFA) è una quintupla $(Q, \Sigma, q_0, \delta, F)$ dove Q è un insieme finito di n stati, Σ è un alfabeto di m simboli, $q_0 \in Q$ è lo stato iniziale, $\delta : Q \times \Sigma \to Q$ è la funzione di transizione e $F \subseteq Q$ è l'insieme di stati finali. Diciamo che una parola $w = w_1 \dots w_j \in \Sigma^*$ è accettata da $\mathcal{A}$ se $\delta(\dots \delta(\delta(q_0, w_1), w_2) \dots, w_j) = \hat{\delta}(q_0, w) \in F$. Il linguaggio accettato dall'automa $\mathcal{A}$ è l'insieme $\mathcal{L}(\mathcal{A}) = \left\{ w \in \Sigma^* : \hat{\delta}(q_0, w) \in F \right\} \subseteq \Sigma^*$.

Stati raggiungibili

Diciamo che uno stato p è *raggiungibile* se possiamo raggiungere p dallo stato iniziale, formalmente $\exists w \in \Sigma^* \ (\delta(q_0, w) = p)$. Diciamo invece che p è *co-raggiungibile* se possiamo raggiungere uno stato finale a partire da p, formalmente $\exists w \in \Sigma^* \ (\delta(p, w) \in F)$. Parliamo di automa raggiungibile (co-raggiungibile) se tutti i suoi stati sono raggiungibili (co-raggiungibili). Si noti che tutti gli stati che non sono raggiungibili o co-raggiungibili sono stati *inutili*, nel senso che la loro rimozione non modifica il linguaggio accettato dall'automa. In letteratura, si usa anche il termine *accessibile* al posto di *raggiungibile* e *co-accessibile* al posto di *co-raggiungibile*.

La relazione di Myhill-Nerode

Il ben noto teorema di Myhill-Nerode implica che per ogni DFA $\mathcal{A}$, all'interno della classe degli automi che accettano lo stesso linguaggio $\{\mathcal{B} : \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})\}$, esiste un automa $\mathcal{C}$ con il minimo numero di stati. Questo automa è unico a meno di isomorfismo. Vediamo ora la relazione di equivalenza usata nel teorema. Sia $\mathcal{A}$ un DFA. Per ogni coppia di stati $p, q \in Q$, definiamo $\sim \subseteq Q \times Q$ come segue:

$$p \sim q \iff \forall w \in \Sigma^* (\delta(p, w) \in F \iff \delta(q, w) \in F)$$

Se p e q sono nella stessa classe di equivalenza rispetto a $\sim$, diciamo che sono indistinguibili e possono essere collassati in un unico stato senza modificare il linguaggio dell'automa.

Il problema di minimizzazione

Problem 1 (DFA MINIMIZATION) *Dato un DFA $\mathcal{A}$, trovare un DFA $\mathcal{A}'$ tale che $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ e con numero degli stati $|Q'|$ minimo.*

In letteratura sono stati proposti vari approcci, tutti che in qualche modo basati sul calcolo della relazione $\sim$. Gli approcci *Bottom Up* partono con due classi $\{F, Q - F\}$ e raffinano le partizioni ad ogni iterazione. Tra questi metodi, i due algoritmi più famosi sono l'algoritmo di Moore e l'algoritmo di Hopcroft. Il primo ha tempo limitato da $\mathcal{O}(n^2)$ ed è molto facile da implementare. Il secondo è più efficiente (ha tempo limitato da $\mathcal{O}(n \log n)$) ma utilizza strutture dati più complesse. Gli approcci *Top Down* partono invece da una classe per ogni stato $\{\{q\} : q \in Q\}$ e ad ogni passo due o più classi vengono unite. Gli algoritmi in questo campo sono meno noti in letteratura e prendono tutti tempo quadratico. Non è ancora stato trovato un algoritmo lineare per il problema, ma è stato dimostrato che, se esiste, allora utilizza l'approccio bottom up. Un algoritmo efficiente per il problema in questione è fondamentale; infatti, può avere molti utilizzi pratici:

- Nel design di un compilatore, e in particolare nel design dell'analizzatore lessicale, si utilizzano degli strumenti che permettono di definire un linguaggio tramite espressioni regolari. Tali strumenti poi creano un automa a partire dall'espressione regolare ed è importante per questioni di efficienza che l'automa sia minimo.
- Nel design di componenti hardware, utilizzando un algoritmo di minimizzazione di automi, è possibile ridurre il numero di componenti logiche necessarie per implementare un determinato comportamento, riducendo così i costi di produzione e di potenza consumata. Inoltre, l'ottimizzazione dell'hardware design contribuisce a migliorare le prestazioni complessive del sistema, rendendolo più veloce ed efficiente.

L'automa minimo è anche utile in contesti di informatica teorica, ad esempio:

- L'automa minimo può essere usato per testare se due linguaggi sono uguali o diversi.
- Se l'automa minimo è Counter-Free allora il linguaggio accettato è Star-Free

Motivazioni del lavoro

Nelle sezioni sucessive, ci concentriamo sulla parallelizzazione dell'algoritmo di Moore. Abbiamo scelto di tentare di parallelizzare questo algoritmo e non altri per diverse ragioni:

- La versione seriale è di facile implementazione e comprensione.

- Nella pratica, l'algoritmo di Moore è più efficiente dell'algoritmo di Hopcroft. Quest'ultimo, infatti, garantisce alcune buone proprietà per il caso pessimo, ma al prezzo di un overhead computazionale.
- L'algoritmo di Hopcroft utilizza strutture dati la cui gestione parallela potrebbe casusare delle complicazioni non banali.

Capitolo 2

Algoritmo di Moore

In questa sezione, richiamiamo brevemente l'algoritmo seriale di Moore. Vediamo prima qualche proprietà teorica; in seguito, analizziamo il codice dell'algoritmo e concludiamo con l'analisi della complessità.

2.1 La teoria

L'algoritmo di Moore è sostanzialmente un modo incrementale di calcolare la relazione $\sim$ di Myhill-Nerode.

Per ogni $i \in \mathbb{N}$ definiamo la relazione $\sim_i \subseteq Q \times Q$ come segue:

$$p \sim_i q \iff \forall w \in \Sigma^{\leq i}(\delta(p,w) \in F \iff \delta(q,w) \in F)$$

Le seguenti proprietà aiuteranno a capire il funzionamento e la correttezza dell'algoritmo:

1. La relazione $\sim_0$ discrimina stati finali e non finali. Si noti infatti che l'unica parola $w \in \Sigma^{\leq 0}$ è la parola vuota ε.
2. La relazione $\sim_{i+1}$ raffina $\sim_i$. Questo significa che ogni classe di $\sim_{i+1}$ è contenuta in un unica classe di $\sim_i$.
3. Possiamo calcolare $\sim_{i+1}$ se abbiamo già calcolato $\sim_i$ e se guardiamo gli stati a distanza un carattere da ogni stato. In particolare, vale:
$$p \sim_{i+1} q \iff p \sim_i q \wedge \forall a \in \Sigma(\delta(p,a) \sim_i \delta(q,a))$$
4. Esiste un punto fisso per cui $\sim_i$ è precisamente $\sim$. Formalmente,
$$(\exists k \leq |Q|)(\forall j \geq k) \quad \sim_k = \sim_j = \sim$$
5. Per testare se $\sim_i$ è uguale a $\sim_{i+1}$ è sufficiente fare un controllo sul numero delle classi. Formalmente, vale:
$$\sim_{i+1} \neq \sim_i \implies |Q/_{\sim_{i+1}}| > |Q/_{\sim_i}|$$

L'automa minimo è l'automa quoziente $\mathcal{A}/_\sim$:

$$\mathcal{A}/_\sim = (Q/_\sim, \Sigma, [q_0], \delta^*, \{[f] : f \in F\}) \text{ dove } \delta^*([q], a) = [\delta(q,a)]$$

2.2 L'algoritmo

In base alle proprietà viste, dato un DFA, sappiamo calcolare $\sim_0$, sappiamo calcolare $\sim_{i+1}$ data $\sim_i$ e sappiamo dire facilmente se siamo arrivati al punto fisso. Vediamo il codice dell'algoritmo di Moore. La procedura `minimize` esegue un ciclo principale (linee 6-8 del seguente frammento) in cui ad ogni passo si raffina la relazione $\sim_i$ per ottenere $\sim_{i+1}$. Quando $\sim$ è stata calcolata, viene computato e restituito l'automa quoziente (linee 10-15).

```
1  DFA minimize(DFA A=(Q, Σ, q0, δ, F))
2    n ← |Q|, n_class ← 0, n_class' ← 2
3    alloc int[n] tilde
4
5    forall (q ∈ Q) tilde[i] ← ((q ∈ F) ? 1 : 2)
6    while(n_class != n_class')
7      n_class ← n_class'
8      n_class' ← refine(tilde)
9
10   alloc DFA A'=(Q', Σ', q0', δ', F') : Σ' = Σ, q0' = tilde[q0]
11   n' ← n_class, alloc int[n] Q'{1,...,n'}
12   forall(p ∈ Q')
13     q ← pick any q ∈ Q : tilde[q] == p
14     if (q ∈ F) F' ← F' ∪ {p}
15     forall (a ∈ Σ) δ'(p,a) ← tilde[δ(q,a)]
16
17   free tilde
18   return A'
```

La procedura `minimize` si basa sulla procedura `refine`, la quale calcola una firma per ogni stato e, usando un algoritmo di ordinamento, restituisce il numero di classi dopo il processo di raffinamento. Le firme sono implementate da un array `sign` di n stringhe di lunghezza $m + 1$.

```
1  int refine(int[] tilde : ref)
2    alloc string[n] sign
3
4    forall(q ∈ Q)
5      sign[q] = (tilde[q],tilde[δ(q,s1)],...,tilde[δ(q,sm)])
6    π ← radixSort(sign)
7
8    class ← 1
9    tilde[π[1]] ← class
10   for(i ← 2 upto n)
11     if (sign[π[i]] != sign[π[i-1]]) class++
12     tilde[π[i]] ← class
13
14   free sign, π
15   return class
```

La procedura `refine` utilizza l'algoritmo `radixSort`, di cui riportiamo il codice per completezza. Si noti che l'algoritmo `countingSort` è composto da 3 fasi seriali: nella prima fase si crea l'istogramma dell'array ricevuto in input (linee 10-12 del seguente frammento di codice), nella seconda se ne calcola una prefix-sum (linee 13-14), nella terza si

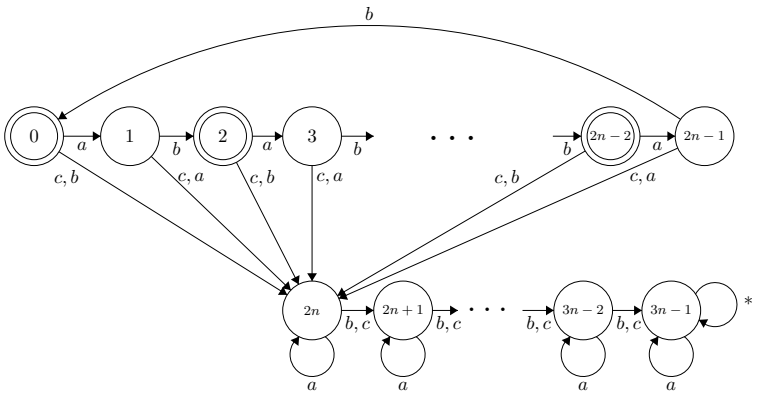


Figura 2.1: L'automa $\mathcal{A}_n$

esegue effettivamente l'ordinamento (linee 15-18). Si noti che l'istogramma viene creato nella forma di un array di k elementi chiamato `count`. La procedura `countingSort` riceve in input una permutazione π che viene modificata dopo l'esecuzione dell'algoritmo. Alla prima iterazione del `for` in cui viene chiamato `countingSort`, alla linea 3, la permutazione in input è la permutazione identica.

```
1  int[] radixSort(string[n] sign)
2    alloc int[n] π{1,...,n}
3    for(j ← 1 upto m+1) countingSort(sign, π, j)
4    return π
5
6  void countingSort(string[n] sign, int[n] π: ref, int j)
7    k ← n_classes // max value we can find in sign
8    alloc int[n] out, int[k] count
9
10   for(i ← 1 upto n)
11       index ← sign[π[i]][j]
12       count[index]++
13   for(i ← 2 upto k)
14       count[i] ← count[i] + count[i - 1]
15   for(i ← n downto 1)
16       index ← sign[π[i]][j]
17       outIndex ← --count[index]
18       out[outIndex] ← π[i]
19   copyArray(out, π)
```

2.3 Analisi della complessità

L'algoritmo di Moore riceve in input DFA completi e raggiungibili. Brevemente il caso migliore è *lineare* $\mathcal{O}(mn)$, il caso peggiore è *quadratico* $\mathcal{O}(mn^2)$. Il caso medio si dimostra essere limitato da $\mathcal{O}(mn \log n)$.

Istanza per il caso migliore. In figura 2.1 vediamo l'automa $\mathcal{A}_n$. Per ogni i, l'automa $\mathcal{A}_i$ riconosce lo stesso linguaggio $(ab)^*$. Si noti che $\mathcal{A}_n$ ha $3n$ stati, di cui n sono finali. L'automa $\mathcal{A}_1$ è il DFA minimo per il linguaggio $(ab)^*$ ed è sempre trovato in sole

due iterazioni dall'algoritmo di Moore. Siccome ogni iterazione richiede tempo lineare, l'algoritmo prende tempo lineare.

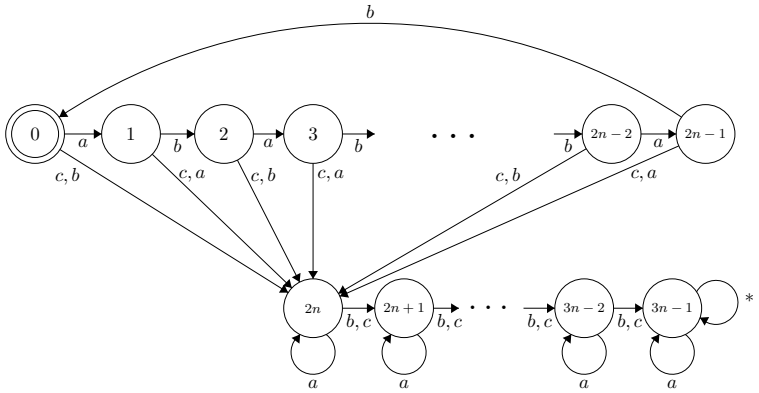


Figura 2.2: L'automa $\mathcal{B}_n$

Istanza per il caso peggiore. In figura 2.2 vediamo l'automa $\mathcal{B}_n$. Per ogni i, l'automa $\mathcal{B}_i$ riconosce il linguaggio $((ab)^i)^*$. Si noti che $\mathcal{B}_n$ ha $3n$ stati di cui solo lo stato 0 è finale. Gli stati $2n+1 \dots 3n-1$ non sono co-raggiungibili e se li rimuoviamo otteniamo l'automa minimo per $((ab)^n)^*$. Dato $\mathcal{B}_n$, la soluzione è sempre trovata in $2n$ iterazioni dall'algoritmo di Moore. Il tempo richiesto dall'algoritmo è quadratico.

Il caso medio. Per caso medio intendiamo il caso in cui in input sia fornito un automa *casuale* e *accessibile*. Iniziamo elencando alcune proprietà dimostrate in letteratura:

- La probabilità che un atuoma casuale accessibile con $m \geq 3$ simboli sia minimo tende ad 1 al tendere all'infinito del numero degli stati.
- Il numero di iterazioni richieste dall'algoritmo di Moore su un DFA $\mathcal{A}$ accessibile è lo stesso che sul DFA minimo per $\mathcal{L}(\mathcal{A})$.
- L'algoritmo di Moore applicato a un DFA minimo con n stati ha complessità $\Omega(\frac{m}{\log m} n \log\log n)$ per un alfabeto di cardinalità $m \geq 2$.

In letteratura è stato mostrato che l'algoritmo di Moore applicato a un DFA casuale accessibile con la condizione che l'insieme degli stati finali F sia scelto in maniera uniforme ha complessità media $\mathcal{O}(mn \log n)$.

2.4 Istanze di test

Per testare i programmi dei capitoli 3 e 4 utilizziamo le seguenti istanze.

ist1B È l'automa $\mathcal{A}_{2000000,30}$ ossia l'automa $\mathcal{A}_{2000000}$ con $m = 30$.

ist2B È l'automa $\mathcal{B}_{5000,20}$ ossia l'automa $\mathcal{B}_{5000}$ con $m = 20$.

ist3B È l'automa $\mathcal{C}_{2000000,30}$ ossia un automa casuale accessibile con 2000000 stati e $m = 30$.

ist1S È l'automa $\mathcal{A}_{20000000,2}$ ossia l'automa $\mathcal{A}_{20000000}$ con $m = 2$.

ist2S È l'automa $\mathcal{B}_{15000,2}$ ossia l'automa $\mathcal{B}_{15000}$ con $m = 2$.

ist3S È l'automa $\mathcal{C}_{20000000,2}$ ossia un automa casuale accessibile con 20000000 stati e $m = 2$.

Le prime 3 istanze hanno un alfabeto *grande*. Le ultime 3 invece hanno un alfabeto di soli due caratteri. Le istanze di tipo 1 testano il caso migliore in cui la procedura `refine` viene chiamata solo due volta. Le istanze di tipo 2 testano il caso peggiore in cui la procedura `refine` viene chiamata un numero lineare di volte. Le istanze di tipo 3 rappresentano il caso medio.

Capitolo 3

Implementazione OpenMP

In questo capitolo descriviamo varie soluzioni parallele utilizzando OpenMP. Partiamo dal codice seriale visto in sezione 2.2 e tentiamo di migliorarlo in maniera progressiva. Le sezioni 3.1-3.6 descrivono sei diversi programmi (che chiameremo rispettivamente programmi 1-6). In ogni sezione descriveremo l'idea principale e i dettagli implementativi e accenneremo ai problemi riscontrati nei test che motiveranno lo sviluppo dei programmi delle sezioni successive. Per testare gli overhead dei vari programmi e il comportamento dei thread sono stati utilizzati gli strumenti *Linaro MAP* e *Performance Analyzer* della suite *Oracle Developer Studio*. Nella sezione 3.7 descriveremo la struttura dei file nel codice e, infine, in sezione 3.8 studieremo approfonditamente l'andamento reale di ciascuno dei sei programmi e lo confronteremo con le aspettative teoriche.

3.1 Uso della direttiva for

In riferimento al codice visto in sezione 2.2, si osservi che ogni chiamata a `radixSort` causa $m+1$ chiamate a `countingSort`. Ogni esecuzione di `countingSort` considera un *digit* dell'array da ordinare e svolge il suo lavoro tramite un array monodimensionale `count`. Una prima idea che viene in mente è di calcolare tutti[1] gli istogrammi (nella variabile `count`) e le prefix-sum in parallelo. Per applicare questa idea, la variabile `count` diventa una matrice di dimensione $(m+1) \times n$. Utilizziamo la direttiva `parallel for` in combinazione con `atomic update` sulla variabile `count`. Si noti che il primo *digit* della firma `sign[...][1]` è calcolato in maniera differente rispetto agli altri m. Le righe 1-6 calcolano il primo *digit*, mentre le righe 9-13 calcolano gli altri m. Si noti inoltre che nel `for` a linea 2 si è utilizzata l'opzione `nowait` che è fondamentale per evitare overhead e si può utilizzare in questo caso siccome il `for` a linea 9 può essere eseguito in maniera indipendente. Alla linea 14, vi è una barriera implicita che non può essere inibita, infatti il calcolo delle prefix-sum richiede che il calcolo degli istogrammi (sulle variabili `count`) sia terminato. A linea 8, si è anche utilizzata l'opzione `collapse`. Si noti infine che in tutti i `for` si è utilizzato uno scheduling statico in quanto tutte le iterazioni hanno indicativamente lo stesso carico.

[1] Si intende *tutti* gli istogrammi e le prefix-sum che verrebbero calcolate in una *singola* esecuzione di `radixSort`.

```
1  #pragma omp parallel num_threads(N)
2    #pragma omp for schedule(static) nowait
3    for(i ← 1 upto n)
4      index ← (sign[i][1] ← tilde[i])
5      #pragma omp atomic update
6      count[1][index]++
7
8    #pragma omp for schedule(static) collapse(2)
9    for(i ← 1 upto n)
10     for(j ← 2 upto m+1)
11       index ← (sign[i][j] ← tilde[ δ(i,j-1) ])
12       #pragma omp atomic update
13       count[j][index]++
14
15   #pragma omp for schedule(static) nowait
16   for(j ← 1 upto m+1)
17     for(i ← 2 upto k)
18       count[j][i] ← count[j][i] + count[j][i - 1]
```

La funzione `radixSort` utilizza i contatori già calcolati ed esegue la fase di ordinamento in seriale. Si noti che per l'ordinamento vengono utilizzati solamente due array `in` e `out`. Alla prima iterazione del ciclo a linea 3 (del prossimo frammento di codice), la variabile `in` rappresenta la permutazione in input (che nel nostro caso è la permutazione identica). Sempre alla prima iterazione, il risultato dell'ordinamento viene salvato nella variabile `out`. In seguito, vengono scambiati i riferimenti di questi due array (linea 8) in modo che la permutazione in input della seconda iterazione sia esattamente la permutazione in output della prima iterazione. Questo fornisce, di per sé, un miglioramento rispetto al codice di `radixSort` presentato in sezione 2.2 che allocava un array `out` ad ogni chiamata di `countingSort`.

```
1  alloc int[n] out
2  in ← π
3  for(j ← 1 upto m+1)
4    for(i ← n downto 1)
5      index ← sign[in[i]][j]
6      outIndex ← --count[j][index]
7      out[outIndex] ← in[i]
8    swap(in, out)
```

Il problema principale di questa versione del programma è che (analizzando con vari strumenti) si nota che la maggior parte del tempo è speso nella funzione `radixSort` che non è stata parallelizzata.

3.2 Uso della direttiva task

L'obiettivo di questa versione è quello di ridurre la parte di codice di `radixSort` non parallelizzata. A questo fine, per $1 \leq j \leq (m+1)$, consideriamo l'uso delle variabili `count[j][1],...,count[j][n]`. Per ogni tale j, definiamo quattro task:

- A_j: inizializzare l'array `count[j]`, e creare `sign[i][j]` per ogni stato `i`

- B_j: riempire `count[j][i]` per ogni $1 \leq i \leq n$
- C_j: calcolare la prefix-sum dell'array `count[j]`
- D_j: usare l'array `count[j]` per eseguire `countingSort` riferito al *digit* `j`

I task sono definiti dentro un ciclo `for` all'interno di una regione `single` (linea 1 del seguente frammento) ossia eseguita da un singolo thread. I task iniziano ad essere eseguiti alla prima barriera che viene trovata. Ovviamente, i task che abbiamo definito hanno delle dipendenze; in particolare:

- B_j dipende da A_j
- C_j dipende da B_j
- D_j dipende da C_j e da D_{j-1}

Se si implementasse questa idea e si analizzasse il comportamento del programma, si otterrebbe che i task di prefix-sum C_j hanno un carico molto inferiore rispetto agli altri. Per rendere di nuovo i task bilanciati, scegliamo di unire i task C_j con i counting-task B_j. I vari task sono quindi:

- A_j: task di inizializzazione e creazione della firma sul *digit* j (righe 3-5 del prossimo frammento di codice)
- B_j: counting-task + task di prefix-sum (righe 6-10)
- C_j: sorting-task (righe 11-16)

Per gestire le dipendenze, nel codice, utilizziamo degli array fittizi `depSig`, `depSum` e `depSort` nella definizione dei task tramite le opzioni `depend(in:)` e `depend(out:)`. Il codice della funzione `refine`, con le idee appena discusse, è il seguente:

```
1  #pragma omp single
2    for(j ← 1 upto m+1)
3      #pragma omp task firstprivate(j) depend(out:depSig[j])
4        for(i ← 1 upto n)
5          sign[i][j] ← tilde[ δ(i,j-1) ]
6      #pragma omp task firstprivate(j) depend(in:depSig[j])
           depend(out:depSum[j])
7        for(i ← 1 upto n)
8          count[j][ sign[i][j] ]++
9        for(i ← 2 upto k)
10         count[j][i] ← count[j][i] + count[j][i - 1]
11     #pragma omp task firstprivate(j) depend(in:depSum[j])
           depend(in:depSort[j-1]) depend(out:depSort[j])
12       for(i ← n downto 1)
13         index ← sign[in[i]][j]
14         outIndex ← --count[j][index]
15         out[outIndex] ← in[i]
16       swap(in, out)
```

Si noti che avendo gestito le cose in questo modo, il sorting-task C_j viene eseguito in parallelo ad altri task; inoltre, non vi è più bisogno di avere primitive atomiche. Eseguendo il codice e analizzandolo con vari strumenti, si nota che molto tempo viene speso nello scheduling dei task che avviene dinamicamente.

3.3 Uso della direttiva sections

Per risolvere i problemi di overhead dovuti alla schedulazione dei task, possiamo provare ad utilizzare la direttiva `parallel sections`. Il punto cruciale sta nel fatto che conosciamo già la forma di una schedulazione ottimale dei task, quindi possiamo implementare quest' ordine staticamente con la direttiva `sections` piuttosto che dinamicamente con la direttiva `task`. Riportiamo in tabella 3.1 la schedulazione ottimale dei task. Si

time instant	**1**	**2**	**3**	**4**				**m**	**m+1**	**m+2**	**m+3**
create `sign`	A_1	A_2	A_3	A_4			A_{m-1}	A_m	A_{m+1}		
count and p-sum		B_1	B_2	B_3	B_4			b_{m-1}	B_m	B_{m+1}	
sort			C_1	C_2	C_3	C_4			C_{m-1}	C_m	C_{m+1}

Tabella 3.1: Schedulazione ottimale dei task A_j, B_j e C_j.

noti che con 3 thread, usiamo solo $m+3$ unità di tempo rispetto alle $3(m+1)$ unità di tempo richieste da un esecuzione seriale.

Il codice della funzione `refine` avrà la seguente forma.

```
1  #pragma omp single    // { A_1 task }
2  #pragma omp sections
3    #pragma omp section // { A_2 task }
4    #pragma omp section // { B_1 task }
5
6  for(j ← 3 upto m+1)
7    #pragma omp sections
8      #pragma omp section  // { A_j task   }
9      #pragma omp section  // { B_j-1 task }
10     #pragma omp section  // { C_j-2 task }
11
12 #pragma omp sections
13   #pragma omp section // { B_m+1 task }
14   #pragma omp section // { C_m task   }
15 #pragma omp single    // { C_m+1 task }
```

In linea di principio, questa soluzione dovrebbe garantire un'efficienza triplicata, ma questo solo se ogni sezione occupa le stesse unità di tempo. Analizzando con vari strumenti il comportamento, si trova che la sezione per il sorting-task prende il doppio del tempo rispetto alle altre.

3.4 Raffinare le sezioni

L'obiettivo della quarta versione del programma è di cercare di dividere in due il task di ordinamento che limitava i tempi della versione precedente. L'idea è di creare due

time instant	**1**	**2**	**3**	**4**				**m**	**m+1**	**m+2**	**m+3**
create `sign`	A_1	A_2	A_3	A_4			A_{m-1}	A_m	A_{m+1}		
count and p-sum		B_1	B_2	B_3	B_4			b_{m-1}	B_m	B_{m+1}	
sort_1			C_1	C_2	C_3	C_4			C_{m-1}	C_m	C_{m+1}
sort_2			D_1	D_2	D_3	D_4			D_{m-1}	D_m	D_{m+1}

Tabella 3.2: Scehdulazione dei task A_j, B_j, C_j e D_j. Gli ultimi due sono i task di ordinamento. Da confrontare con la tabella 3.1.

`section` da eseguire in parallelo che mirano ad ordinare due regioni dell'array non in sovrapposizione. La schedulazione desiderata è mostrata in tabella 3.2. Per ogni j, creiamo una variabile `th[j]` (*threshold* per il *digit* j) con il seguente significato: un thread (quello incaricato del task C_j) ordina solo i valori sotto la soglia `th[j]`, mentre un altro thread (quello incaricato del task D_j) ordina solo i valori sopra la soglia. Il codice delle due sezioni è il seguente:

```
1  #pragma omp section
2    for(i ← n downto 1)
3      index ← sign[in[i]][j]
4      if(index <= th[j]) continue
5      outIndex ← --count[j][index]
6      out[outIndex] ← in[i]
7  #pragma omp section
8    for(i ← n downto 1)
9      index ← sign[in[i]][j]
10     if(index > th[j]) continue
11     outIndex ← --count[j][index]
12     out[outIndex] ← in[i]
```

Il test delle linee 4 e 10 determina quali elementi vengono ordinati dai rispettivi thread. I cicli delle linee 2 e 8 eseguono comunque n iterazioni, ma quelle in cui il test ha successo (ossia nelle quali vengono ignorate rispettivamente le linee 5-6 e 11-12) hanno un carico molto inferiore rispetto alle altre. Rimane da discutere come calcolare e scegliere il valore di `th[j]` in modo che le due sezioni si dividano equamente il carico di lavoro.

Iniziamo osservando alcune proprietà:

- Dopo che per il *digit* j è stata eseguita la prefix-sum, si ha banalmente che la sequenza `count[j][1],...,count[j][k]` è non decrescente.
- Se vale `count[j][i] = v` allora significa che vi sono v valori minori uguali ad i e $n - v$ valori più grandi di i.

Per ogni *digit* j vorremmo trovare un indice `th[j]` tale per cui i numeri `count[j][th[j]]` e `n-count[j][th[j]]` siano *bilanciati*, ossia abbiano differenza in valore assoluto minima. Risolvere in maniera esatta il problema di minimizzazione risulterebbe in uno spreco di tempo. Optiamo quindi per una soluzione euristica che sia facile da implementare e che funzioni bene in molti casi. L'idea è di scegliere `th[j]` come il più piccolo indice tale per cui `count[j][th[j]]` sia maggiore di $n/2$. Questa idea si può implementare in tempo lineare in k, numero di elementi dell'array `count[j]`. Il codice è il seguente.

```
1  th[j] ← rand(1, k-1)
2  for(i ← 2 upto k-1)
3    if(count[j][i] >= (n/2))
4      th[j] ← i
5      break
```

Come discuteremo nella sezione dei risultati, la divisione ottenuta non sarà sempre ottima. Si pensi ad esempio al caso in cui $k = 2$, ossia al caso in cui ci sono solo due classi. In questo caso, il thread incaricato del task C_j ordinerà gli elementi di classe 0 e il thread incaricato del task D_j ordinerà gli elementi di classe 1. Evidentemente, se il numero di elementi di una classe è molto maggiore di quello dell'altra, i due thread non avranno lo stesso carico di lavoro e saranno sbilanciati, producendo un overhead aggiuntivo. Inoltre, la soluzione non scala col numero di thread: usarne più di 4, infatti, non porta benefici.

3.5 Loop Unfolding

Fino a questo punto, abbiamo provato due approcci per parallelizzare una singola esecuzione della funzione `refine`.

1. Tramite la direttiva `parallel for`, abbiamo cercato di parallelizzare ogni singola fase di `refine` (creazione delle firme, counting e prefix-sum) in maniera indipendente e poi abbiamo eseguito serialmente le varie fasi. In questo modo non è stata parallelizzata la fase di sorting. Possiamo dire che con questo approccio abbiamo usato una **parallelizzazione verticale dei task**.

2. Abbiamo provato a spezzare le varie fasi per ogni *digit* j in modo che possano essere eseguite in parallelo (secondo le dipendenze identificate). Sono state utilizzate le direttive `parallel task` e `parallel section`. Si noti che in questo caso anche la fase di sorting avviene in parallelo alle altre. Possiamo dire che con questo approccio abbiamo usato una **parallelizzazione orizzontale dei task**.

Come vedremo nella sezione dei risultati, il vantaggio del primo approccio è che scala con il numero di thread, ma siccome la fase di sorting non è stata , avremo delle prestazioni insoddisfacenti. Per quanto riguarda il secondo approccio otterremo dei buoni risultati in termini di prestazioni, ma il numero di thread eseguibili in ogni istante è limitato (3 o 4).

Quello che non abbiamo provato fino ad ora è *srotolare* le varie iterazioni del ciclo `while` della funzione `minimize`. In questa sezione proviamo quindi a unire due esecuzioni successive della funzione `refine` a cui applicheremo la parallelizzazione orizzontale delle fasi. A questo fine, abbiamo bisogno di studiare ulteriormente la relazione di Myhill-Nerode. I lemmi e le proprietà che seguono (al meglio delle mie ricerche) rappresentano dei contributi originali non presenti in letteratura (a differenza delle proprietà 1-5 riportate in sezione 2.1 che giustificano la correttezza dell'algoritmo di Moore).

Dalla proprietà 3 di sezione 2.1 possiamo calcolare $\sim_{i+2}$ a partire da $\sim_i$ e $\sim_{i+1}$ nel seguente modo:

$$
\begin{aligned}
p \sim_{i+2} q &\iff p \sim_{i+1} q \wedge (\forall a \in \Sigma\ \delta(p,a) \sim_{i+1} \delta(q,a)) && (3.1)\\
&\iff p \sim_i q \wedge (\forall a \in \Sigma\ \delta(p,a) \sim_i \delta(q,a)) && (3.2)\\
&\wedge (\forall a \in \Sigma\ \delta(p,a) \sim_i \delta(q,a) \wedge (\forall b \in \Sigma\ \delta(p,ab) \sim_i \delta(q,ab)))\\
&\iff p \sim_i q \wedge (\forall a \in \Sigma\ \delta(p,a) \sim_i \delta(q,a)) && (3.3)\\
&\wedge (\forall a,b \in \Sigma\ \delta(p,ab) \sim_i \delta(q,ab))
\end{aligned}
$$

L'equivalenza (3.1) segue direttamente dalla proprietà 3. L'equivalenza (3.2) è l'applicazione della proprietà 3 alle due relazioni $\sim_{i+1}$. L'equivalenza (3.3) segue dal fatto che il termine $\forall a \in \Sigma\ (\delta(p,a) \sim_i \delta(q,a))$ compare due volte. Se volessimo ora utilizzare i conti appena svolti per unire due iterazioni successive della funzione `refine`, le firme (ossia gli array `sign[i]` per ogni stato i) dovrebbero avere $1 + m + m^2$ *digit*. Fissato uno stato i:

- La variabile `sign[i][1]` memorizzerebbe la classe di equivalenza dello stato i.
- Le variabili `sign[i][2],...,sing[i][m+1]` memorizzerebbero le classi di equivalenza degli stati raggiungibili dallo stato i con ciascuno degli m caratteri.
- Le variabili `sign[i][m+2],...,sign[i][m*m+m+1]` memorizzerebbero le classi di equivalenza degli stati raggiungibili dallo stato i con ciascuna delle m^2 stringhe di lunghezza 2.

L'intuizione è che gli m *digit* per le classi di equivalenza degli stati raggiungibili da un carattere siano ridondanti e in qualche modo questa informazione sia già inclusa negli m^2 *digit* per le classi di equivalenza degli stati a distanza di 2 caratteri. Quest'intuizione è formalizzata dal seguente lemma.

Lemma 1 (Unfolding Property) *Per ogni $i > 0$, si ha*

$$
p \sim_{i+2} q \iff p \sim_i q \wedge (\forall a,b \in \Sigma\ \delta(p,ab) \sim_i \delta(q,ab))
$$

Intuitivamente, il termine $p \sim_i q$ garantisce che valga $p \sim_{i+2} q$ per parole di lunghezza $0,\dots,i$. Il termine $\forall a,b \in \Sigma\ (\delta(p,ab) \sim_i \delta(q,ab))$ garantisce che valga $p \sim_{i+2} q$ per parole di lunghezza $2 \dots i+2$. Affinché l'unione di questi intervalli sia contigua dobbiamo richiedere che valga $i > 0$. Sfruttando questo lemma possiamo calcolare $\sim_{i+2}$ a partire da $\sim_i$ con firme di soli $m^2 + 1$ *digit*. Si noti che $\sim_2$ non può essere calcolata con questo metodo. Poniamoci ora nel caso particolare in cui $m = 2$. Supponiamo di avere già calcolato $\sim_1$; contiamo il numero di unità di tempo richieste per calcolare $\sim_3$ in due casi:

Caso 1. Calcoliamo $\sim_2$ e poi $\sim_3$ usando due volte le firme da $m + 1 = 2 + 1 = 3$ *digit*.

Caso 2. Calcoliamo $\sim_3$ usando una singola volta la firma da $m^2 + 1 = 2^2 + 1 = 5$ *digit*.

In tabella 3.3 vediamo che sono richieste 10 unità di tempo per il primo caso. In tabella 3.4 vediamo che sono richieste solo 7 unità di tempo per il secondo caso, e quindi otteniamo un risparmio di tempo del 30%.

			$\sim_2$					$\sim_3$		
time instant	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
create `sign`	A_1	A_2	A_3			A_1	A_2	A_3		
count and p-sum		B_1	B_2	B_3			B_1	B_2	B_3	
sort			C_1	C_2	C_3			C_1	C_2	C_3

Tabella 3.3: Caso 1: Calcoliamo $\sim_3$ passando per $\sim_2$

time instant	**1**	**2**	**3**	**4**	**5**	**6**	**7**
create `sign`	A_1	A_2	A_3	A_4	A_5		
count and p-sum		B_1	B_2	B_3	B_4	B_5	
sort			C_1	C_2	C_3	C_4	C_5

Tabella 3.4: Caso 2: Calcoliamo $\sim_3$ direttamente da $\sim_1$

Si noti infine che calcolare $\sim_3$ direttamente da $\sim_1$ con le firme di $m^2 + 1$ *digit* permette di ottenere un risparmio di tempo anche con un'esecuzione seriale: nel caso 1, infatti, sarebbero necessarie 18 unità di tempo, mentre nel caso 2 sarebbero sufficienti solo 15 unità di tempo. Quindi, con un'esecuzione seriale, otteniamo un risparmio di unità di tempo del $16,6\%$.

Proviamo ora a svolgere gli stessi conti nel caso particolare in cui $m = 3$:

Caso 1. Utilizziamo due firme di $m + 1 = 3 + 1 = 4$ *digit*. Lo scheduling parallelo per eseguire due volte i task $A_j, B_j, C_j : j = 1 \dots 4$ richiede $6 \times 2 = 12$ unità di tempo. In seriale, invece, sono richieste $4 \times 3 \times 2 = 24$ unità di tempo.

Caso 2. Utilizziamo una firma di $m^2 + 1 = 3^2 + 1 = 10$ *digit*. Lo scheduling parallelo per eseguire $A_j, B_j, C_j : j = 1 \dots 10$ richiede 12 unità di tempo. In seriale, invece, sono richieste $10 \times 3 = 30$ unità di tempo.

Si noti quindi che con $m = 3$ perdiamo tutti i benefici. Con $m = 4$ e successivi, calcolare $\sim_3$ in maniera diretta da $\sim_1$ è meno efficiente in termini di unità di tempo rispetto al modo classico. Siccome l'idea analizzata funziona solo per $m = 2$, il programma che analizzeremo nella sezione dei risultati funziona come segue:

- Se $m = 2$, viene prima calcolata $\sim_1$ in maniera *classica*, e poi $\sim_3, \sim_5, \sim_7 \dots$ sfruttando il *loop unfolding*.
- Se $m > 2$, vengono calcolate $\sim_1, \sim_2, \sim_3, \sim_4 \dots$ in maniera *classica*.

Concludiamo questa sezione enunciando una generalizzazione del lemma 1.

Lemma 2 (Generalized Unfolding Property) *Per ogni $k \geq 2$, per ogni $i \geq k - 1$, si ha*

$$p \sim_{i+k} q \iff p \sim_i q \wedge (\forall a_1, \dots, a_k \in \Sigma \ \delta(p, a_1 \dots a_k) \sim_i \delta(q, a_1 \dots a_k))$$

Si noti che il lemma 1 è il caso speciale in cui $k = 2$. In poche parole, il lemma asserisce che per calcolare $\sim_{i+k}$ direttamente da $\sim_i$, possiamo utilizzare delle firme di

$1+m^k$ *digit* piuttosto che delle firme di $1+m+m^2+m^3+\cdots+m^k$ *digit* che sarebbero richieste se avessimo sfruttato in maniera naive la proprietà 3 di sezione 2.1. Il lemma 2 potrebbe essere utilizzato per unire un numero arbitrario di iterazioni della funzione `refine` in un'unica iterazione.

A titolo di esempio, poniamoci nel caso in cui $k = 3$ e $m = 2$. Supponiamo di avere già a disposizione $\sim_2$; contiamo il numero di unità di tempo richieste per calcolare $\sim_5$ in due casi:

Caso 1. Calcoliamo $\sim_3$, $\sim_4$ e $\sim_5$ usando tre volte le firme da $m + 1 = 2 + 1 = 3$ *digit*.

Caso 2. Calcoliamo $\sim_5$ usando una singola volta la firma da $m^3 + 1 = 2^3 + 1 = 9$ *digit*.

Supponiamo che i task A_j, B_j, C_j siano eseguiti in parallelo. Nel caso 1 sono necessarie $(3 + 2) \times 3 = 5 \times 3 = 15$ unità di tempo. Nel caso 2 sono sufficienti $9 + 2 = 11$ unità di tempo. Il risparmio (teorico) di unità di tempo è di circa il 27%, che è quindi un peggioramento rispetto al caso in cui avevamo utilizzato il lemma 1 (avevamo ottenuto un risparmio di tempo del 30%). Unire tre iterazioni in una potrebbe comunque avere dei vantaggi pratici che potrebbero far tendere ad una maggior efficienza effettiva. Si pensi al caso in cui dopo ogni iterazione della funzione `refine` vi sia un overhead non trascurabile. In questo caso, unire 3 iterazioni in una, creerebbe meno overhead complessivo rispetto ad unire 2 iterazioni in una.

I conti eseguiti in relazione al lemma 2 sono puramente teorici e non è stata realizzata alcuna reale implementazione. Come lavoro futuro rimane quindi da investigare la reale utilità del lemma 2.

3.6 RadixSort parallelo

L'ultima idea che proviamo su OpenMP è quella di cercare di parallelizzare `radixSort`. Il programma parallelo quindi utilizzerà delle direttive `parallel for` per creare le firme; poi, verrà lanciato `radixSort` parallelo. In questa sezione descriviamo una semplice idea di base. Questa idea verrà ripresa e sviluppata in maniera approfondita nel capitolo 4 in cui trattiamo CUDA.

Come abbiamo visto in sezione 2.2, il codice di `radixSort` è basato su quello di `countingSort`. Cerchiamo quindi di parallelizzare quest'ultimo algoritmo. Descriviamo l'idea tramite l'esempio della figura 3.1. Supponiamo di volere ordinare un array `arr` di n elementi e di avere a disposizione N thread. Supponiamo inoltre che gli elementi di `arr` siano compresi tra 1 e k. Al fine di rendere l'esposizione più chiara, trattiamo il caso semplice in cui N è multiplo di n. In figura, vogliamo ordinare un array `arr` di 20 elementi (indicizzati da 1 a 20) con $k = 3$ e abbiamo a disposizione $N = 4$ thread (indicizzati da 1 a 4). Ogni thread i lavora su n/N elementi, e precisamente sugli elementi nelle posizioni da $((i - 1) \times 5) + 1$ a $i \times 5$ (estremi inclusi). Ad esempio, in figura, il thread 2 lavora sugli elementi nelle posizioni 6, 7, 8, 9 e 10 colorate in verde.

Passo 1 Ogni thread i alloca un array locale $\texttt{count}_\texttt{i}$ di k elementi che viene riempito in modo che se vale $\texttt{count}_\texttt{i}\texttt{[j] = v}$, allora nella porzione dell'array su cui lavora il thread i vi sono v elementi di valore j (in altri termini, viene creato l'istogramma di tale porzione di array). Ad esempio, in figura, considerando gli elementi del

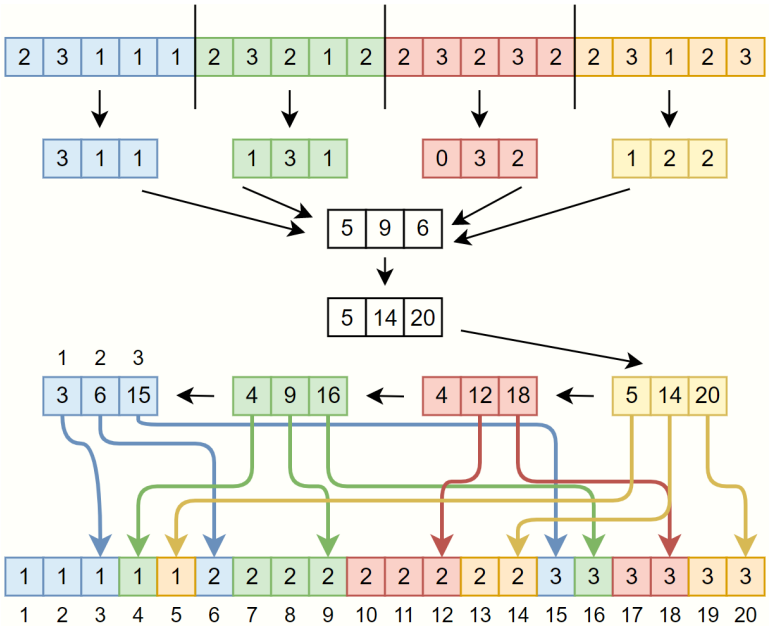


Figura 3.1: Flusso di esecuzione di `countingSort` paralello.

thread 2, vi sono 1 elemento di valore 1, 3 elementi di valore 2 e 1 elemento di valore 3. A questo primo passo, tutti gli N thread possono lavorare in parallelo.

Passo 2 Consiste nel creare un array `globalCount` di k elementi tale per cui valga

$$\texttt{globalCount[j]} = \texttt{count}_1\texttt{[j]} + \ldots + \texttt{count}_N\texttt{[j]}$$

In altri termini, viene calcolato l'istogramma dell'intero array. A questo secondo passo, lavorano solo i primi k thread. Il thread i calcola `globalCount[i]` accedendo a $\texttt{count}_1\texttt{[i]},\ldots,\texttt{count}_N\texttt{[i]}$.

Passo 3 Il thread 1 esegue la prefix sum dell'array `globalCount`.

Passo 4 Consiste nel ridistribuire le informazioni sugli array locali $\texttt{count}_i$. In particolare, dopo il processo di ridistribuzione, se vale $\texttt{count}_i\texttt{[j]} = \texttt{v}$, allora il thread i dovrà scrivere i valori j a partire dalla posizione v. Vediamo ora come avviene il processo di ridistribuzione; per comodità, indichiamo con $\texttt{count'}_i\texttt{[j]}$ il valore di $\texttt{count}_i\texttt{[j]}$ prima del processo. Per ogni indice $1 \leq j \leq k$, i passaggi eseguiti sono, nell'ordine:

- $\texttt{count}_N\texttt{[j]} \leftarrow \texttt{globalCount[j]}$
- $\texttt{count}_{N-1}\texttt{[j]} \leftarrow \texttt{count}_N\texttt{[j]} - \texttt{count'}_N\texttt{[j]}$
- ...
- $\texttt{count}_1\texttt{[j]} \leftarrow \texttt{count}_2\texttt{[j]} - \texttt{count'}_2\texttt{[j]}$

Il quarto passo può essere implementato parallelamente con k thread (ognuno che esegue gli N passaggi di cui sopra).

Passo 5 Consiste nell'ordinamento effettivo dell'array `arr`. Ogni thread parte dall'estremo destro della porzione di array su cui lavora. Supponiamo che il thread i stia considerando la posizione j di `arr` e che valga `arr[j] = v`. Il thread i accede

dunque a `count_i[v]`, esegue un post-decremento e scrive v nell'array di output, nella posizione restituita dal post-decremento. Il thread i poi procede considerando la posizione $j-1$ di `arr`. Il processo termina quando ogni thread ha scritto tutti gli n/N elementi su cui lavora. Anche quest'ultimo passo, come il primo, può essere eseguito da tutti gli N thread in parallelo.

I passi 2-4 possono essere implementati in maniera diretta in OpenMP utilizzando varie direttive `parallel for`; vale la pena discutere i dettagli implementativi dei passi 1 e 5. Nel prossimo frammento di codice vediamo il passo 1. La variabile `chunkSize` denoterà il numero di elementi su cui lavora ciascun thread (in questo caso, n/N supponendo che N sia multiplo di n). Si noti che secondo la specifica di OpenMP 5.0, i thread eseguiranno iterazioni contigue: in particolare, il thread di indice i eseguirà serialmente le iterazioni nell'intervallo $[((t-1) \times \frac{n}{N}) + 1, t \times \frac{n}{N}]$ per un qualche valore di t non necessariamente uguale ad i. Sempre secondo la specifica di OpenMP, la mappatura fra i thread e i gruppi di iterazioni è arbitraria, e in particolare, non vi è nessuna garanzia che al thread i corrisponda il gruppo di iterazioni i-mo.

```
1  chunkSize ← n/N
2  i ← omp_get_thread_num()
3  #pragma omp for schedule(static, chunkSize)
4    for (j ← 1 upto n)
5      count_i[arr[j]]++
```

Nel prossimo frammento di codice vediamo il passo 5, nel quale ogni thread i deve considerare gli *stessi* elementi di `arr` che ha considerato nel passo 1, ma in ordine inverso. Il problema principale risiede nel fatto che è necessario che il `for` del passo 5 esegua la stessa mappatura fra i thread e i gruppi di iterazioni eseguita dal `for` del passo 1. In altri termini, se il `for` del passo 1 associa ad ogni thread i le iterazioni $[((t_i-1) \times \frac{n}{N}) + 1, t_i \times \frac{n}{N}]$ e il `for` del passo 5 associa ad ogni thread i le iterazioni $[((t'_i-1) \times \frac{n}{N}) + 1, t'_i \times \frac{n}{N}]$; dobbiamo richiedere che valga $t_i = t'_i$ per ogni $i = 1, \ldots, N$. Seguendo sempre la specifica di OpenMP 5.0, ciò che vogliamo è garantito, scegliendo `schedule(static)`, se avvengono le seguenti condizioni:

1. Le due regioni di worksharing-loop hanno lo stesso numero di iterazioni;
2. Se si usa la clausola `schedule(static,n)`, le due regioni devono avere lo stesso numero n specificato;
3. Le due regioni di worksharing-loop sono legate alla stessa regione `omp parallel`;
4. Nessun loop è associato a un costrutto SIMD.

Queste quattro condizioni sono soddisfatte dall'implementazione dei passi 1 (frammento precedente) e 5 (frammento successivo). Affinché la mappatura avvenga in modo corretto, è necessario che il `for` proceda per valori crescenti (anziché decrescenti, come di solito avviene nelle implementazioni di CountingSort). La linea 3 calcola l'indice corretto con cui accedere ad `arr`. Le linee 4-6 salvano il risultato nell'array `out`.

```
#pragma omp for schedule(static, chunkSize)
  for (j ← 1 upto n)
    ind ← ...
    el ← arr[ind]
    outIndex ← --count_i[ind]
    out[outIndex]← el
```

In questo modo, rispetto ai programmi descritti precedentemente, l'intero programma scala con il numero di thread.

Come accennato, riprenderemo queste idee nella trattazione di CUDA (capitolo 4); anticipiamo che in tale trattazione, il passo 1 sarà noto come fase 1, i passi 2-4 come fase 2 e il passo 5 come fase 3.

3.7 Note implementative

La struttura dei file è la seguente

```
OPENMP/
├── 0_main_serial.c
├── 1_main_for_parallelism.c
├── 2_main_task.c
├── 3_main_sections.c
├── 4_main_sections_splittedsort.c
├── 5_main_doublesign.c
├── 6_main_parallelradixsort.c
├── dfa.h
├── Makefile
├── test.sh
├── multi_test.sh
├── test_cases/
│   ├── test_case1.txt
│   ├── test_case2.txt
│   └── ...
└── csv/
    ├── test_res1.csv
    ├── test_res2.csv
    └── ...
```

Il programma `0_main_serial.c` è il programma seriale discusso nella sezione 2.2. Per $i > 0$, il programma `i_main_....c` è il programma discusso nella sezione 3.i. Il file `dfa.h` contiene la definzione della struttura `dfa` usata dai vari programmi e contiene i metodi per generare gli automi delle istanze di test. Il `Makefile` ha quattro comandi principali.

- `make` crea tutti gli eseguibili, e in particolare:
  - Per ogni $i > 0$, crea un eseguibile `i_parallel...` che si ottiene compilando il programma `i_main_....c` con l'opzione `-fopenmp`.

  - Per ogni $i > 0$, crea un eseguibile `i_serial` che si ottiene compilando il programma `i_main_....c` *senza* l'opzione `-fopenmp`. L'eseguibile ottenuto può quindi essere eseguito solo in maniera seriale.
  - Viene creato l'eseguibile `0_serial` compilando il file `0_main_serial.c`.

- `make debug` crea gli eseguibili abilitando i simboli di debug.
- `make clean` elimina gli eseguibili creati da uno dei due comandi precedenti.
- `make test` esegue dei test su tutti gli eseguibili. Se non si visualizzano scritte in rosso allora tutti i test sono andati a buon fine. Il comando può richiedere alcuni minuti prima di completare.

Il `Makefile` contiene inoltre sei comandi: `make ist1B`, `make ist2B`, `make ist3B`, `make ist1S`, `make ist2S` e `make ist3S`. Tali comandi sono presenti anche nel `Makefile` dell'implementazione CUDA (capitolo 4) e la loro utilità sarà discussa nel capitolo 5.

La sintassi per invocare gli eseguibili è la seguente

```
./nome_eseguibile <n> <m> <threads> <language> [<output>]
```

I primi due parametri e il quarto decidono l'automa di input. L'ultimo parametro decide cosa verrà stampato e il terzo parametro decide con che modalità verrà eseguito il programma. In particolare:

- n è il numero di stati dell'automa da minimizzare
- m è la cardinalità dell'alfabeto dell'automa da minimizzare
- *threads* è il numero di thread con cui il programma verrà eseguito. Si noti che questo parametro viene ignorato per gli eseguibili `i_serial`.
- *language* può valere 0, 1 oppure 2. Nei rispettivi casi, verrà ricevuto in input l'automa $\mathcal{A}_{n,m}$, $\mathcal{B}_{n,m}$ e $\mathcal{C}_{n,m}$.
- *output* è un parametro opzionale e può valere 0 1 oppure 2. Se vale 0 non verrà stampato nulla in output. Se vale 1 verrà stampato solo l'automa minimizzato. Se vale due verrà stampato sia l'automa di input che l'automa minimizzato. Il valore di default è 2.

Finiamo la descrizione dei vari elementi nella cartella `OPENMP`:

- Il programma bash `test.sh` può essere invocato con la sintassi

  ```
  ./test.sh <n> <m> <threads> <language> <output>
  ```

  ed esegue tutti gli eseguibili con le opzioni specificate e stampa in output i tempi di esecuzione di ciascun programma.

- Il programma bash `multi_test.sh` può essere invocato con la sintassi

```
./multi_test.sh <n> <m> <threads> <language> <output>
                <ITERATIONS>
```

ed esegue il programma `test.sh` per il numero di iterazioni specificate. Inoltre nella cartella `csv` viene creato un file che contiene i tempi di esecuzione di ciascun programma ad ogni esecuzione.

- La cartella `test_cases` contiene i vari test che vengono eseguiti con `make test`. Il nome del file indica l'input dei programmi e ogni file contiene l'output atteso per tale input.

3.8 Risultati e discussione

Discutiamo ora i risultati dei programmi considerati. I test sono stati eseguiti su una macchina equipaggiata con un processore *11th Gen Intel(R) Core(TM) i7-1165G7 at 2.80GHz* e 16GB di RAM. Se `P` è un programma OpenMP, chiamiamo con $\texttt{P}_\texttt{p}\texttt{(n)}$ il programma eseguito con n thread in **p**arallelo. Chiamiamo $\texttt{P}_\texttt{s}$ il programma compilato senza l'opzione `-fopenmp` e quindi eseguito in maniera **s**eriale. Per ogni programma parallelo (di indice i), confrontiamo i tempi

- Con il programma 0, ossia il programma seriale;
- Con il programma precedente (ossia di indice $i-1$);
- Con lo stesso programma ma eseguito in seriale.

Nelle tabelle 3.5 e 3.6 mostriamo rispettivamente i tempi dei programmi sulle istanze *Big* e sulle istanze *Small*. Per ogni programma i, mostriamo quattro dati:

- t (s) è il tempo medio (in secondi) preso dal programma i su 4 esecuzioni.
- 0 (%) è la percentuale di tempo di esecuzione risparmiato del programma i rispetto al programma 0.
- i_s (%) è la percentuale di tempo di esecuzione risparmiato del programma i_p rispetto al programma i_s. Questo numero è definito solo per i programmi i_p, ossia per i programmi compilati con `-fopenmp`.
- $i-1_p$ (%) è la percentuale di tempo di esecuzione risparmiato del programma i_p rispetto al programma $i-1_p$. Anche questo numero è definito solo per i programmi i_p.

Discutiamo ora i risultati ottenuti dai vari programmi delle sezioni 3.1-3.6.

1. Il programma che calcola in anticipo le variabili `count` e che usa le direttive `parallel for`, ad una prima vista, mostra qualche leggero miglioramento rispetto al programma 0. In realtà, i miglioramenti sono dovuti ad una ristrutturazione generale del codice e ad un migliore accesso alle variabili. Notiamo infatti che il programma 1 eseguito in seriale presenta miglioramenti simili e nelle istanze ist2B, ist1S e ist2S conviene utilizzare la versione seriale rispetto a quella parallela. Analizzando la situazione, questo fenomeno è dovuto al fatto che la maggior

parte del tempo è speso nell'esecuzione (che rimane seriale) di `radixSort`; inoltre, le regioni di codice OpenMP creano più overhead di sincronizzazione che vantaggi effettivi.

2. Il programma che utilizza la direttiva `parallel task` (eseguito in parallelo) presenta dei peggioramenti nelle istanze ist1B e ist3B rispetto al programma precedente (eseguito in parallelo). Nelle istanze *Small* invece abbiamo un leggero miglioramento. Questo comportamento è in linea con le aspettative. La direttiva `parallel task` assegna i `task` dinamicamente e questo crea molto overhead. Più sono i task da eseguire e più vi è overhead. Ricordando che il numero di task è proporzionale alla cardinalità dell'alfabeto m, questo spiega perché il peggioramento avviene solo nelle istanze *Big*.

3. Il programma che usa le tre sezioni presenta finalmente dei miglioramenti significativi. Ottiene il miglior risultato di tutti i programmi testati nelle istanze ist1B, ist2B, ist3B e ist1S. I miglioramenti rispetto al programma 0, allo stesso programma eseguito in seriale e al programma parallelo precedente sono ottimi con picchi rispettivamente del 75% (ist1B), 53% (ist3B) e 72% (ist3B). In generale il programma ha più benefici su istanze con m alto. Questo è in linea con le aspettative: dato un valore di m, le sezioni A_j, B_j e C_j richiedono $m+3$ unità di tempo per un'esecuzione parallela e $3\times(m+1)$ unità di tempo per un'esecuzione seriale. Nel caso in cui $m = 20$ (all'incirca come nelle istanze *Big*) si richiedono 23 unità di tempo rispetto a 63, ossia un risparmio di unità di tempo del 64%. Nel caso in cui $m = 2$ (come nelle istanze *Small*), invece, si richiedono 5 unità di tempo rispetto invece di 9, ossia un risparmio di unità di tempo del 44%. I valori teorici sono in accordo con i dati reali. Facendo la media delle colonne i_s (%) delle istanze *Big* e *Small*, otteniamo un risparmio di tempo rispettivamente del 43% e 22%. I dati reali sono quantitativamente inferiori rispetto ai valori teorici poiché i secondi parlano di *unità di tempo*; tuttavia nella pratica, i task A_j, B_j e C_j non prendono esattamente lo stesso tempo e vi sono overhead di sincronizzazione. Inoltre, la parallelizzazione delle sezioni non rappresenta la totalità del codice, quindi altri overhead influiscono sulla differenza dei risultati teorici e pratici. Si noti infine che questo programma è stato eseguito con soli 3 thread (ossia il massimo numero di sezioni eseguibili parallelamente), e aggiungere ulteriori thread (quantomeno sulla macchina su cui sono stati eseguiti i test) non aiuta le prestazioni e anzi crea ulteriore overhead.

4. Il programma 4 è identico al programma 3, eccezion fatta per due sezioni che eseguono parallelamente il compito di ordinamento. I due thread ordinano regioni diverse e non in sovrapposizione dell'array. Questo programma può essere eseguito da 4 thread invece che da 3. Lo scopo di dividere il compito di ordinamento in due sarebbe quello di bilanciare maggiormente le unità di tempo (siccome il compito di ordinamento era quello più pesante) e ridurre l'overhead dovuto alla sincronizzazione. Guardando ai risultati, nella pratica, non si ottiene un miglioramento, e anzi in alcuni casi si ottiene un peggioramento. Le istanze su cui si comporta meglio sono le istanze del tipo 3. Questo può essere dovuto al fatto che l'euristica di splitting non è ottimale nei casi degli automi $\mathcal{A}$ e $\mathcal{B}$ ma funziona meglio nel caso in cui l'automa di input sia costruito in maniera casuale. Sull'istanza ist3S, il programma ottiene un risultato migliore rispetto a tutti gli altri programmi, con un risparmio di tempo rispetto al programma seriale del 63%.

5. Il programma 5 è costruito a partire dal programma 4. Nel caso in cui $m \neq 2$, il programma 5 esegue esattamente le stesse operazioni del programma 4. Escludiamo quindi dalla discussione tutte le istanze *Big*. In questo caso, il miglioramento o peggioramento rispetto al programma 4 è dovuto al caso e non ad una ragione teorica. Nel caso in cui $m = 2$ (quindi sulle istanze *Small*), il programma parte calcolando $\sim_1$ in maniera classica e poi (se serve) calcola $\sim_3$, $\sim_5$, $\sim_7$, ecc. Nella sezione 3.5 sono già stati discussi i vantaggi teorici nel calcolare $\sim_{i+2}$ direttamente da $\sim_i$ senza passare per $\sim_{i+1}$. In questo caso si dovrebbe ottenere un risparmio di unità di tempo del 30%. Questo vantaggio è ottenuto sull'istanza ist2S con un risparmio di tempo rispetto al programma precedente del 26%. Su questa istanza, il programma 5 ottiene il risultato migliore fra tutti i programmi, con un risparmio di tempo del 60% rispetto al programma seriale 0. Guardiamo ora all'istanza ist1S. Il programma 4 calcola $\sim_1$, $\sim_2$ e termina. In questo caso vengono utilizzate $2 \times (m + 3) = 2 \times 5 = 10$ unità di tempo. Il programma 5 invece calcola $\sim_1$, $\sim_3$ e poi termina. In questo, sono richieste 5 unità di tempo per il calcolo di $\sim_1$ e 7 unità di tempo per il calcolo di $\sim_3$, per un totale di 12 unità di tempo, e quindi abbiamo una perdita di unità di tempo teorica del 20% rispetto al programma 4. Nella pratica, abbiamo ottenuto una perdita di tempo del 15%. In conclusione, la perdita di tempo sull'istanza ist1S è dovuta al fatto che il programma, per terminare, potrebbe fermarsi al calcolo di $\sim_2$; tuttavia viene calcolata anche $\sim_3$. Un ragionamento simile si applica all'istanza ist3S, sulla quale otteniamo una perdita di tempo del 19%. Sappiamo che un'automa casuale con un numero così alto di stati con alta probabilità è già minimo. Nel caso non lo sia, il numero di iterazioni richieste per la minimizzazione è comunque molto basso. Per giustificare la perdita di tempo, si consideri il caso in cui, per terminare, è richiesto solo il calcolo di $\sim_i$; tuttavia, il programma 5 calcola $\sim_{i+1}$. Il ragionamento è analogo a quello dell'istanza ist1S.

6. Il programma 6 è basato su una parallelizzazione naive di `radixSort`. Il vantaggio di questo programma è che scala con il numero di thread e non è limitato a funzionare con 3 o 4 thread. In tabella, riporto i risultati con 4 thread. Dai risultati ottenuti, il programma 6 non migliora i precedenti ma si hanno comunque delle buone prestazioni sia rispetto al programma 0, sia rispetto alla versione seriale. Sebbene i dati non siano riportati in tabella, si è avuto modo di provare il programma 6 su una macchina con processore *Intel(R) Xeon(R) CPU E5-2609 v4 @ 1.70GHz* con 16 core fisici e si è notato che il programma funziona meglio con 16 thread rispetto ad 8 e funziona meglio con 8 thread rispetto a 4. Rimangono da investigare le prestazioni del programma utilizzando versioni più avanzate di `radixSort` parallelo e capire se si riesce a ottenere un risultato migliore rispetto ai programmi precedenti, quantomeno con un alto numero di core fisici.

PROGRAMMA		ist1B				ist2B				ist3B			
		t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)
0	naive	27.2	/	/	/	91.3	/	/	/	39.6	/	/	/
1	$\text{for}_p(4)$	16.1	41%	14%	/	79.9	12%	-20%	/	34.7	12%	16%	/
	for_s	18.8	31%	/	/	66.4	27%	/	/	41.5	-5%	/	/
2	$\text{task}_p(4)$	19.3	29%	33%	-19%	72.0	21%	8%	10%	38.3	3%	18%	-10%
	task_s	29.0	-7%	/	/	78.0	15%	/	/	46.7	-18%	/	/
3	$\text{section}_p(3)$	6.8	**75%**	47%	64%	47.7	**48%**	28%	34%	10.7	**73%**	53%	72%
	section_s	12.9	52%	/	/	65.8	28%	/	/	22.9	42%	/	/
4	$\text{splitsort}_p(4)$	7.6	72%	49%	-11%	51.0	44%	29%	-7%	10.7	73%	48%	1%
	splitsort_s	14.8	45%	/	/	71.7	21%	/	/	20.8	47%	/	/
5	$\text{unfold}_p(4)$	8.3	69%	45%	-9%	50.6	45%	30%	1%	10.4	74%	47%	3%
	unfold_s	15.2	44%	/	/	71.8	21%	/	/	19.8	50%	/	/
6	$\text{psort}_p(4)$	15.3	44%	44%	-85%	88.3	3%	2%	-75%	31.1	22%	55%	-198%
	psort_s	27.3	0%	/	/	90.5	1%	/	/	69.5	-75%	/	/

Tabella 3.5: Risultati dei programmi sulle istanze *Big*.

PROGRAMMA		ist1S				ist2S				ist3S			
		t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)
0	naive	19.7	/	/	/	158.74	/	/	/	109.9	/	/	/
1	$\text{for}_p(4)$	15.9	19%	-1%	/	138.36	13%	-36%	/	72.6	34%	25%	/
	for_s	15.8	20%	/	/	101.38	36%	/	/	96.7	12%	/	/
2	$\text{task}_p(4)$	14.9	24%	5%	7%	125.96	21%	-18%	9%	72.4	34%	19%	0%
	task_s	15.7	20%	/	/	106.92	33%	/	/	89.7	18%	/	/
3	$\text{section}_p(3)$	9.0	**54%**	31%	39%	88.24	44%	9%	30%	47.2	57%	28%	35%
	section_s	13.2	33%	/	/	97.36	39%	/	/	65.6	40%	/	/
4	$\text{splitsort}_p(4)$	10.2	48%	27%	-13%	86.13	46%	19%	2%	40.8	**63%**	14%	14%
	splitsort_s	14.0	29%	/	/	106.87	33%	/	/	65.5	40%	/	/
5	$\text{unfold}_p(4)$	11.7	40%	30%	-15%	63.92	**60%**	23%	26%	48.5	56%	37%	-19%
	unfold_s	16.9	14%	/	/	82.56	48%	/	/	76.8	30%	/	/
6	$\text{psort}_p(4)$	11.0	44%	38%	6%	104.43	34%	21%	-63%	63.4	42%	44%	-31%
	psort_s	17.9	9%	/	/	132.17	17%	/	/	112.5	-2%	/	/

Tabella 3.6: Risultati dei programmi sulle istanze *Small*.

Possiamo riassumere il comportamento dei vari programmi come segue:

1 Cattive prestazioni dovute alla parte di ordinamento seriale.

2 Cattive prestazioni dovute alla sincronizzazione dei task.

3 Buone prestazioni. Limite di 3 thread.

4 La pipeline usa 4 thread ma lo splitting non è sempre ottimale e può causare overhead aggiuntivo.

5 Buone prestazioni nel caso in cui $m = 2$ e il numero di iterazioni richiesto è alto.

6 Discrete prestazioni; `radixSort` parallelo permette di scalare con il numero di thread.

Rispondiamo ora alla domanda *Se devo minimizzare un automa, quale programma utilizzo?* con la seguente regola euristica:

> Con più di 16 processori fisici usare il programma **6**. Con 3 processori fisici usare il programma **3**. Con 4-16 processori fisici usare il programma **4**. Nel caso particolare in cui $m = 2$, se si sospetta che il numero di iterazioni richiesto non sia costante rispetto a n, usare il programma **5**.

In conclusione, siamo riusciti ad ottenere dei buoni risultati sfruttando l'architettura OpenMP. Come lavoro futuro rimangono da provare diverse versioni di `radixSort` parallelo; rimane inoltre da investigare come applicare il lemma 2 per unire più iterazioni della funzione `refine`.

Capitolo 4

Implementazione CUDA

In questo capitolo, discuteremo l'implementazione CUDA dell'algoritmo di Moore. Riprenderemo, sviluppandola, l'ultima versione parallela dell'algoritmo di cui al capitolo precedente, risultato della parallelizzazione di RadixSort e, in ultima analisi, di CountingSort. Nella sezione 4.1, tratteremo CountingSort parallelo: suddivideremo CountingSort in tre fasi e per ciascuna di esse proporremo diverse soluzioni implementative. Nella sezione 4.2, tratteremo RadixSort parallelo: vedremo RadixSort come un processo del tipo produttore-consumatore e ne proporremo una soluzione implementativa basata su una combinazione di stream, eventi e direttive OpenMP. Nella sezione 4.3, discuteremo brevemente la versione parallela risultante dell'algoritmo di Moore. Nella sezione 4.4, daremo alcune note implementative. Nella sezione 4.5, analizzeremo i tempi delle diverse soluzioni implementative e ne confronteremo brevemente i risultati.

4.1 CountingSort parallelo

Sia A l'array, e siano n la lunghezza di A e k il numero di classi di elementi di A, i.e.

$$k = |\{A[p] : p = 0, \ldots, n-1\}|$$

Nel contesto dell'algoritmo di Moore, possiamo assumere che le classi siano $0, \ldots, k-1$ con $k \leq n$.

Definizione. Dato un segmento A' di A:

- *L'*istogramma *di A' è un array B' di lunghezza k tale che per ogni $i = 0, \ldots, k-1$, $B'[i]$ è il numero di elementi di classe i che occorrono in A'.*
- *L'*indice *di A' è un array C' di lunghezza k tale che per ogni $i = 0, \ldots, k-1$, $C'[i]$ è la posizione immediatamente successiva a quella in cui occorre l'ultimo elemento di A' di classe i nell'ordinamento (stabile) di A.*

4.1.1 Implementazione principale

Immaginiamo di suddividere A in m segmenti $A_0, \ldots, A_{m-1}$; l'idea è che ogni blocco della griglia corrisponda a un segmento di A di lunghezza proporzionale alla sua dimensione. Proponiamo un'implementazione di CountingSort parallelo sulla base del seguente schema di alto livello in tre fasi:

1. Si calcola l'istogramma B_j di ogni A_j;
2. A partire dai B_j, si calcola l'indice C_j di ogni A_j;
3. A partire dai C_j, si calcola l'ordinamento stabile di A.

Precisamente, vedremo tre varianti delle fasi 1-3 e due varianti della fase 2.

Ora, vedremo in seguito che a meno di limitare il numero m di segmenti in cui suddividiamo A, l'implementazione di CountingSort parallelo che proponiamo si comporta male in senso spaziale per $k \gg 1$, quindi vogliamo innanzitutto ricondurci a $k \leq K$ per qualche $K = 2^H$ fissato. Sia $h = \lceil \log_2 k \rceil$; interpretiamo ogni classe di elementi di A come un numero binario di h bit. Consideriamo due diverse decomposizioni di A:

- *Decomposizione "top-down".* A ogni passo, separiamo gli elementi "più piccoli" da quelli "più grandi". Al primo passo, separiamo gli elementi di A di classe con primo bit 0 da quelli di classe con primo bit 1, a ottenere al più due segmenti A^0 e A^1. Al secondo passo, separiamo gli elementi di A^0 e A^1 di classe con secondo bit 0 da quelli di classe con secondo bit 1, a ottenere al più quattro segmenti A^{00}, A^{01}, A^{10} e A^{11}. E così via (fig. 4.1). Al termine della decomposizione, le posizioni corrispondenti agli elementi di ogni segmento sono note, quindi il loro inserimento può avvenire in parallelo.

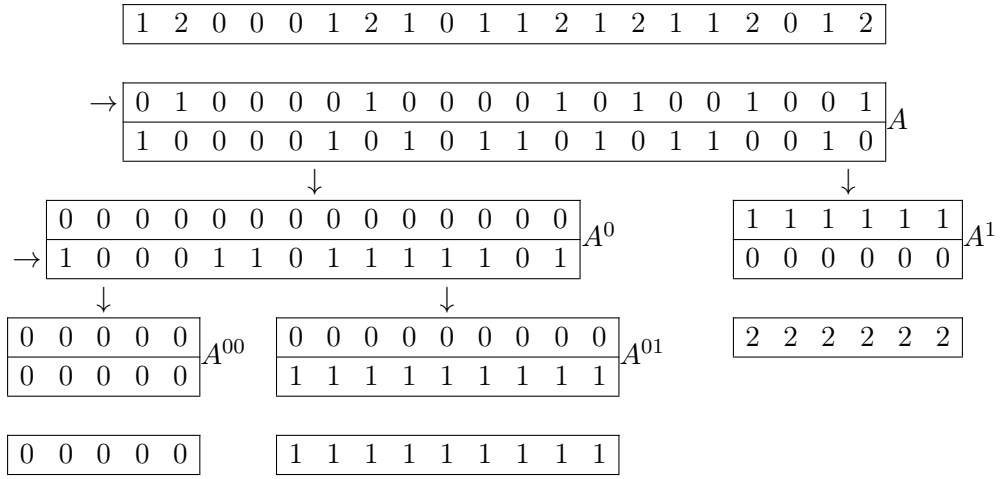


Figura 4.1: *Decomposizione top-down*

- *Decomposizione "bottom-up".* A ogni passo, separiamo gli elementi "pari" da quelli "dispari". Al primo passo, separiamo gli elementi di A di classe con h-mo bit 0 da quelli di classe con h-mo bit 1, a ottenere al più due segmenti che riuniamo in un unico array A'. Al secondo passo, separiamo gli elementi di A' di classe con $(h-1)$-mo bit 0 da quelli di classe con $(h-1)$-mo bit 1, a ottenere al più due segmenti che riuniamo in un unico array A''. E così via (fig. 4.2). Al termine della decomposizione, gli elementi dell'array sono già inseriti nelle posizioni corrispondenti.

Entrambe le decomposizioni costituiscono un'alternativa a CountingSort parallelo. Da un lato, ogni passo può essere calcolato ricorrendo a CountingSort su due sole "macroclassi" di elementi; dall'altro, può essere calcolato senza ricorrere a CountingSort come

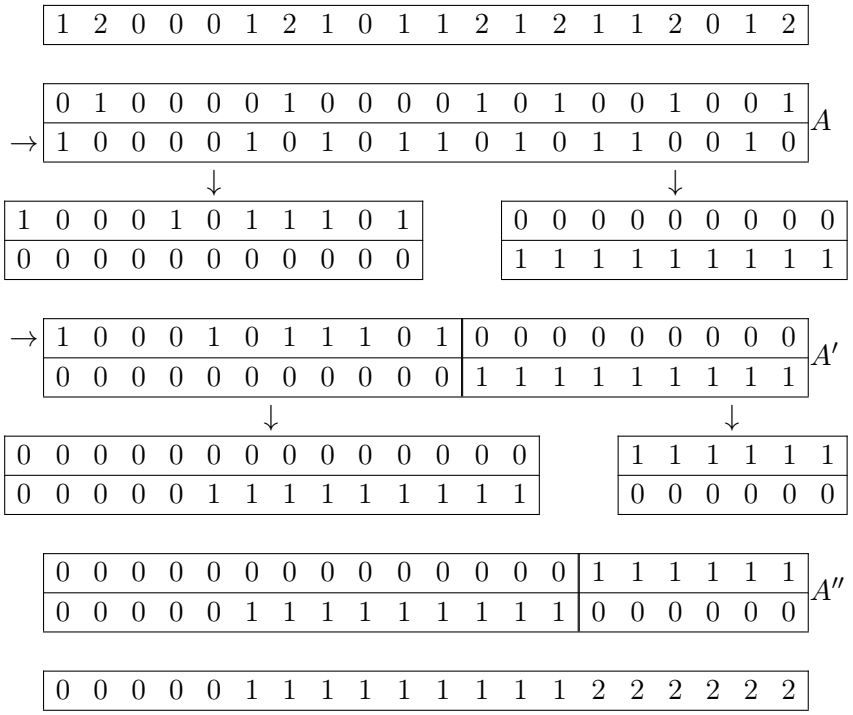


Figura 4.2: *Decomposizione bottom-up*

coppia di estrazioni di elementi che soddisfano condizioni complementari. Discuteremo in seguito della decomposizione "bottom-up" quale alternativa a CountingSort parallelo.

Tornando a CountingSort parallelo, la decomposizione "bottom-up" consente, in una direzione, di vedere ogni istanza di CountingSort su classi "a grana fine" $(0, \ldots, k-1)$ come un'istanza di RadixSort, la quale può essere decomposta in una sequenza di h istanze di CountingSort su classi "a grana grossa" $(0, 1)$ e, nell'altra direzione, di ricomporre sequenze di H tali istanze di CountingSort in un'istanza di RadixSort, la quale può essere vista come un'istanza di CountingSort su classi "a grana media" $(0, \ldots, K-1)$. Ne consegue che possiamo sempre ricondurci a $k \leq K$, invece che una sola istanza di CountingSort su classi "a grana fine", risolvendo $\lceil h/H \rceil$ istanze di CountingSort su classi "a grana media". Il ragionamento può essere esteso all'istanza di RadixSort nel contesto dell'algoritmo di Moore; discuteremo in seguito di una tale estensione. Lo svantaggio è che prima di risolvere ogni istanza di CountingSort successiva, occorre riordinare A sulla base della permutazione calcolata risolvendo l'istanza di CountingSort precedente, nel quale riordinamento gli accessi in memoria globale non sono coalescenti.

Abbiamo accennato che l'implementazione di CountingSort parallelo che proponiamo si basa su uno schema di alto livello in tre fasi: nel seguito, vediamo ciascuna fase nelle diverse varianti.

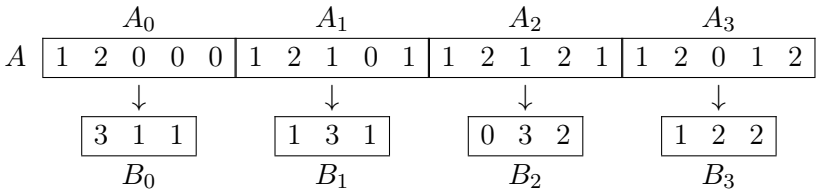
Fase 1

Di nuovo, immaginiamo di suddividere A in m segmenti $A_0, \ldots, A_{m-1}$, e lanciamo una griglia unidimensionale di m blocchi unidimensionali di K thread ciascuno, dove s'intende che ogni blocco j opera sul segmento A_j, possibilmente in block-strided loop. Per semplicità, assumeremo che n sia multiplo di m e tutti gli A_j abbiano la stessa lunghez-

za n/m; inoltre, nei frammenti di codice di esempio, assumeremo che tale lunghezza sia pari alla dimensione di un blocco, i.e. $n/m = K$.

Ogni blocco j calcola l'istogramma di A_j, i.e. un'array B_j di lunghezza k tale che per ogni $i = 0, \dots, k-1$,

$$B_j[i] = |\{p : A_j[p] = i\}|$$



Osserviamo che la griglia prende spazio $\Theta(km)$, i.e. prende tanto più spazio 1) quanto più è alto il numero di classi (k) e 2) quanto più suddividiamo il lavoro (m); inoltre, i B_j sono riempiti per al più n/mk. Al limite, se $k = n$, allora la griglia prende spazio $\Theta(nm) = \mathcal{O}(n^2)$ e i B_j sono riempiti per al più $1/m$, i.e. in modo sparso; se inoltre n è abbastanza grande, allora devono essere calcolati in memoria globale. Siccome invece ci siamo ricondotti a $k \leq K$, la griglia prende spazio $\Theta(Km) = \mathcal{O}(n)$ e i B_j sono riempiti per almeno $1/K$, i.e. in modo denso; se inoltre K è abbastanza piccolo, allora possono essere calcolati in memoria condivisa.

Riguardo al calcolo dei B_j, individuiamo tre soluzioni (di seguito semplificate):

(a) *Ogni thread $i = 0, \dots, k-1$ di ogni blocco j incrementa $B_j[i]$ in un ciclo* `for`*:*

```
1  int i = threadIdx.x;
2  if (i < k)
3      B_j[i] = 0;
4  for (int p = 0; p < blockDim.x; p++) {
5      if (A_j[p] == i)
6          B_j[i]++;
7  }
```

(b) *Per $i = 0, \dots, k-1$, ogni blocco j esegue* `__syncthreads_count` *sulla condizione $A_j[p] \stackrel{?}{=} i$, dove p è il thread del blocco:*

```
1  int p = threadIdx.x;
2  for (int i = 0; i < k; i++) {
3      int b = __syncthreads_count(A_j[p] == i);
4      if (p == i)
5          B_j[i] = b;
6  }
```

(c) *Ogni thread p di ogni blocco j esegue un'operazione atomica di incremento su $B_j[A_j[p]]$:*

```
{
    int i = threadIdx.x;
    if (i < k)
        B_j[i] = 0;
}

__syncthreads();

{
    int p = threadIdx.x;
    atomicInc(&B_j[A_j[p]], INT_MAX);
}
```

Rispetto alle prime due soluzioni, un vantaggio di questa terza soluzione è che non ingenera divergenza fra i thread, al costo di eseguire operazioni atomiche.

Si può assumere che ogni B_j sia calcolato in tempo

$$\begin{aligned} &\mathcal{O}(n/m) = \mathcal{O}(K) && \text{nella soluzione (a)} \\ &\mathcal{O}(k) && \text{nella soluzione (b)} \\ &\mathcal{O}(\max_i B_j[i]) \overset{(1)}{=} \mathcal{O}(n/mk) \overset{(2)}{=} \mathcal{O}(1) && \text{nella soluzione (c)} \end{aligned}$$

sotto l'ipotesi, nella soluzione (c), che le classi siano distribuite in modo equo (1) e $k \approx K$ (2).

Fase 2

Variante "orizzontale". *La griglia calcola l'istogramma di A, i.e. un array B di lunghezza k tale che*

$$B = \sum_{j=0}^{m-1} B_j$$

i.e. B è la somma elemento per elemento dei B_j; dopodiché un blocco calcola l'indice di A, i.e. un array C di lunghezza k tale che

$$C = P_B$$

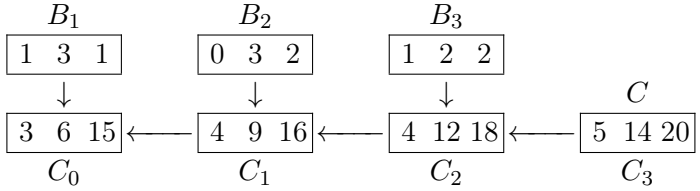
dove P_B denota la prefix-sum inclusiva di B.

B_0	3	1	1
B_1	1	3	1
B_2	0	3	2
B_3	1	2	2
	↓	↓	↓
B	5	9	6
		↓	
C	5	14	20

Riguardo al calcolo di B, l'idea è che per ogni $i = 0, \ldots, k-1$, il thread i di ogni blocco j esegua un'operazione atomica di somma con $B_j[i]$ su $B[i]$; il calcolo di P_B è ben noto.

Ogni blocco j calcola l'indice di A_j, i.e. un array C_j di lunghezza k tale che

$$C_j = \begin{cases} C & \text{se } j = m-1 \\ C_{j+1} - B_{j+1} & \text{altrimenti} \end{cases}$$



Osserviamo che per ogni $j = 0, \ldots, m-1$,

$$C_j = C_{j+1} - B_{j+1} = \cdots = C_{m-1} - B_{m-1} - \cdots - B_{j+1} = C - \sum_{j'>j} B_{j'}$$

Ne consegue che nessun C_j può essere calcolato a partire dai B_j finché C non sia calcolato. Lanciamo quindi due griglie seriali di dimensione intera (m blocchi), separate da un punto di sincronizzazione con lo host. Al termine della prima griglia, tutti i B_j sono calcolati e inoltre si può assumere che C sia calcolato, e inizia la seconda griglia, ogni blocco j della quale calcola C_j sottraendo $B_{j+1}, \ldots, B_{m-1}$ a C, per un totale di $\mathcal{O}(km^2)$ sottrazioni; B è un prodotto di scarto del calcolo.

Possiamo fare di meglio lanciando $\mathcal{O}(\log m)$ griglie seriali: la prima di dimensione intera (m blocchi) e le altre di dimensione sempre minore ($m - 2^0, \ldots, m - 2^{\lceil \log_2 m \rceil - 1}$ blocchi) secondo lo schema di riduzione naïve per il calcolo della prefix-sum, a meno dell'ultima, di dimensione intera (m blocchi), di nuovo, separate da un punto di sincronizzazione con lo host. Infatti, $C = P_B = \overline{P}_B + B$, quindi

$$C_j = P_B - \sum_{j'>j} B_{j'} = \overline{P}_B + B - \sum_{j'>j} B_{j'} = \overline{P}_B + \sum_{j'} B_{j'} - \sum_{j'>j} B_{j'} = \overline{P}_B + \sum_{j' \le j} B_{j'}$$

dove $\overline{P}_B$ denota la prefix-sum esclusiva di B. Ne consegue che se vediamo i B_j come "macro-elementi" di un array $\mathfrak{B}$ di lunghezza m tale che per ogni $j = 0, \ldots, m-1$, $\mathfrak{B}[j] = B_j$, allora

$$C_j = \overline{P}_B + P_{\mathfrak{B}}[j]$$

dove $P_{\mathfrak{B}}$ denota la prefix-sum inclusiva di $\mathfrak{B}$. Di nuovo, al termine della prima griglia, di dimensione intera, tutti i B_j sono calcolati e inoltre si può assumere che B e C siano calcolati, e inizia una serie di $\mathcal{O}(\log m)$ griglie di dimensione sempre minore. Ogni blocco di ogni griglia della serie addiziona due B_j secondo lo schema di riduzione naïve per il calcolo della prefix-sum, per un totale di $\mathcal{O}(km \log m)$ addizioni. Al termine dell'ultima griglia della serie, $P_{\mathfrak{B}}$ è calcolato e inizia un'ultima griglia di dimensione intera, ogni blocco j della quale calcola C_j addizionando $P_{\mathfrak{B}}[j]$ a $\overline{P}_B = C - B$. Lo schema di riduzione in due passate per il calcolo della prefix-sum necessiterebbe del doppio delle griglie con conseguente overhead, quindi lo schema naïve parrebbe un buon compromesso.

Si noti che il calcolo di C a partire da B potrebbe essere parallelizzato rispetto al calcolo di $P_{\mathfrak{B}}$ a partire dai B_j.

Variante "verticale". Ci proponiamo di risolvere il problema della co-dipendenza dei blocchi nel calcolo dei C_j. Per ogni $i = 0, \ldots, k-1$, vediamo la sequenza di elementi

$$(B_0[i], \ldots, B_{m-1}[i])$$

come un array B^i di lunghezza m, i.e. incolonniamo i B_j e consideriamo la colonna i-ma B^i della matrice così ottenuta, e analogamente, vediamo la sequenza di elementi

$$(C_0[i], \ldots, C_{m-1}[i])$$

come un array C^i di lunghezza m. Osserviamo che per ogni $i = 0, \ldots, k-1$, $j = 0, \ldots, m-1$,

$$\begin{aligned} C^i[j] = C_j[i] = \overline{P}_B[i] + \sum_{j' \le j} B_{j'}[i] &= \sum_{i' < i} B[i'] + \sum_{j' \le j} B_{j'}[i] \\ &= \sum_{i' < i} \sum_{j'} B_{j'}[i'] + \sum_{j' \le j} B_{j'}[i] = \sum_{i' < i} \sum_{j'} B^{i'}[j'] + \sum_{j' \le j} B^i[j'] \end{aligned}$$

Ne consegue che se vediamo i B^i come segmenti consecutivi di un array $\mathcal{B}$ di lunghezza km tale che per ogni $i = 0, \ldots, k-1$, $j = 0, \ldots, m-1$, $\mathcal{B}[im+j] = B^i[j]$, e analogamente, vediamo i C^i come segmenti consecutivi di un array $\mathcal{C}$ di lunghezza km, allora

$$\mathcal{C} = P_{\mathcal{B}}$$

dove $P_{\mathcal{B}}$ denota la prefix-sum inclusiva di $\mathcal{B}$.

	B^0	B^1	B^2
B_0	3	1	1
B_1	1	3	1
B_2	0	3	2
B_3	1	2	2

	B^0				B^1				B^2			
$\mathcal{B}$	3	1	0	1	1	3	3	2	1	1	2	2
					↓							
$\mathcal{C}$	3	4	4	5	6	9	12	14	15	16	18	20
	C^0				C^1				C^2			

	C^0	C^1	C^2
C_0	3	6	15
C_1	4	9	16
C_2	4	12	18
C_3	5	14	20

Lanciamo quindi tre griglie seriali: una di dimensione intera (blocchi da K thread) e le altre due di dimensione dimezzata (blocchi da $K/2$ thread), separate da un punto di sincronizzazione con lo host. Al termine della prima griglia, tutti i B_j sono calcolati e iniziano la seconda e terza griglia, ogni blocco delle quali addiziona una serie di elementi secondo lo schema di riduzione in due passate per il calcolo della prefix-sum, per un totale di $\mathcal{O}(km)$ addizioni; B e C non sono mai calcolati. Al termine della seconda e terza griglia, tutti i C_j sono calcolati. (Nella pratica, potremmo dover lanciare più di tre griglie.) Lo svantaggio di questa variante è che nella costruzione di $\mathcal{B}$ a partire dai B_j, e viceversa, nella costruzione dei C_j a partire da $\mathcal{C}$, gli accessi alla memoria globale non sono coalescenti.

Fase 3

Ogni blocco j inserisce gli elementi di A_j nelle posizioni indicate da C_j.

Riguardo all'inserimento, analogamente alla fase 1, individuiamo tre soluzioni (di seguito semplificate):

(a) *Ogni thread $i = 0, \ldots, k-1$ di ogni blocco j decrementa $B_j[i]$ e inserisce gli $A_j[p]$ di classe i in un ciclo* `for`*:*

```
1  int i = threadIdx.x;
2  for (int p = 0; p < blockDim.x; p++) {
3      if (A_j[p] == i) {
4          insert A_j[p] at position C_j[i] - B_j[i]
5          B_j[i]--;
6      }
7  }
```

(b) *Per $i = 0, \ldots, k-1$, ogni blocco j estrae e inserisce gli $A_j[p]$ di classe i:*

```
1  __shared__ extern int e[]; // |e| = blockDim.x

2  int p = threadIdx.x;
3  for (int i = 0; i < k; i++) {
4      if (B_j[i] > 0) {
5          if (A_j[p] == i)
6              e[p] = 1;
7          else
8              e[p] = 0;
9          compute the exclusive prefix-sum of e
10         if (A_j[p] == i)
11             insert A_j[p] at position C_j[i] - B_j[i] + e[p]
12     }
13 }
```

Linea 4 controlla che vi siano elementi di classe i, altrimenti è inutile eseguire l'estrazione. Linee 5-8 calcolano una bitmap degli elementi di classe i. Linea 9 calcola la prefix-sum esclusiva della bitmap. Linee 10-11 eseguono l'estrazione degli elementi di classe i usando la prefix-sum della bitmap.

(c) *Ogni thread p di ogni blocco j esegue un'operazione atomica di decremento su $B_j[A_j[p]]$ e inserisce $A_j[p]$:*

```
1  int p = threadIdx.x;
2  int b = atomicDec(&B_j[A_j[p]], 0);
3  insert A_j[p] at position C_j[A_j[p]] - b;
```

Il problema di questa soluzione è che gli elementi di A_j di una stessa classe potrebbero essere inseriti in disordine. (Nel contesto dell'algoritmo di Moore, non lavoriamo tanto sugli elementi di A quanto sulle posizioni, ed è richiesto che l'ordinamento risultante da ogni applicazione di CountingSort sia stabile. Ne consegue che ogni due elementi di A di una stessa classe, pur virtualmente indistinguibili, devono giacere dopo l'ordinamento nello stesso ordine relativo in cui giacevano prima dell'ordinamento.) Quindi, modifichiamo la procedura come segue:

```
1  __shared__ extern int min[]; // |min| = k
```

```
2  {
3      int i = threadIdx.x;
4      if (i < k)
5          min[i] = blockDim.x;
6  }

7  __syncthreads();

8  {
9      int p = threadIdx.x;
10     bool done = false;
11     bool done_all = false;
12     while (!done_all) {
13         if (!done)
14             atomicMin(&min[A_j[p]], p);

15         __syncthreads();

16         if (min[A_j[p]] == p) {
17             min[A_j[p]] = blockDim.x;
18             insert A_j[p] at position C_j[A_j[p]] - B_j[A_j[p]]
19             B_j[A_j[p]]--;
20             done = true;
21         }
22         done_all = __syncthreads_and(done);
23     }
24 }
```

Linee 4-5 preparano l'array ausiliario `min` per il primo turno di inserimenti. Linea 12 controlla che vi sia ancora qualche elemento da inserire. Linea 13 controlla che l'elemento p-mo sia ancora da inserire. Linea 14 non inserisce l'elemento p-mo, bensì per ogni i, calcola la posizione minima in cui occorre un elemento restante di classe i. Linea 16 controlla che p coincida con tale posizione minima. Linea 17 prepara l'array ausiliario `min` per il turno di inserimenti successivo. Si noti che a ogni turno, sono inseriti i soli elementi restanti di ogni classe che occorrono in posizione minima, risultando in un inserimento ordinato.

Di nuovo, si può assumere che gli elementi di A_j siano inseriti in tempo

$$\begin{aligned}
&\mathcal{O}(n/m) = \mathcal{O}(K) && \text{nella soluzione (a)}\\
&\mathcal{O}(k \log(n/m)) = \mathcal{O}(k \log K) \overset{(1)}{=} \mathcal{O}(k) && \text{nella soluzione (b)}\\
&\mathcal{O}(\max_i B_j[i]^2) \overset{(1)}{=} \mathcal{O}((n/mk)^2) \overset{(2)}{=} \mathcal{O}(1) && \text{nella soluzione (c)}
\end{aligned}$$

sotto le seguenti ipotesi:

- Nella soluzione (b), che le classi siano distribuite in modo iniquo (1);
- Nella soluzione (c), che le classi siano distribuite in modo equo (1) e $k \approx K$ (2).

Sull'equidistribuzione delle classi

Di seguito, ci concentriamo sulla soluzione (c) delle fasi 1-3. Sia A' un segmento di A; diciamo che le classi di elementi sono *equamente distribuite* lungo A' se in valore atteso, ogni classe conta lo stesso numero di elementi in A'.

Nei frammenti di codice di esempio, abbiamo assunto che la lunghezza degli A_j fosse pari alla dimensione di un blocco. In realtà, è possibile che i blocchi operino sul corrispondente segmento di A in block-strided loop ma, come accennato, l'idea è che la lunghezza degli A_j sia proporzionale alla dimensione di un blocco. Inoltre, abbiamo assunto che tale dimensione fosse pari al numero $K = 2^H$ di classi cui intendiamo ricondurci. Ne consegue che se ci riconduciamo a tale numero di classi e le classi sono equamente distribuite lungo gli A_j, allora in media, ogni classe conta un numero costante di elementi in ogni A_j: questa è la situazione più favorevole nella soluzione (c). (Quanti meno elementi conta ogni classe, tanto più le operazioni atomiche possono essere parallelizzate.) Di seguito, per semplicità, assumeremo che la lunghezza degli A_j sia pari a K.

Denotiamo con $\mathrm{id}(t)$ la sequenza $(0, 1, ..., t-1)$. Se partiamo da $k > K$ e le classi sono equamente distribuite lungo gli A_j, allora possiamo assumere che lo siano anche nel ricondurci a $k \leq K$ come abbiamo descritto in precedenza. Di seguito, descriveremo come ricondurci a $k \approx K$ a partire da $k \leq K$ cercando di preservare l'eventuale equidistribuzione delle classi. Interpretiamo ancora ogni classe di elementi di A come un numero binario di h bit. La prima idea è di passare da A a un array $\overline{A}$ per composizione di $H - h$ bit fittizi tratti da una sequenza $\mathrm{id}(n)$ di $\lceil \log_2 n \rceil =: \ell$ bit (fig. 4.3). Non è difficile vedere che ordinare (in modo stabile) A è equivalente a ordinare (in modo stabile) $\overline{A}$. A scopo dimostrativo, consideriamo il caso particolare $k = 1$, i.e. $h = 0$, e quindi

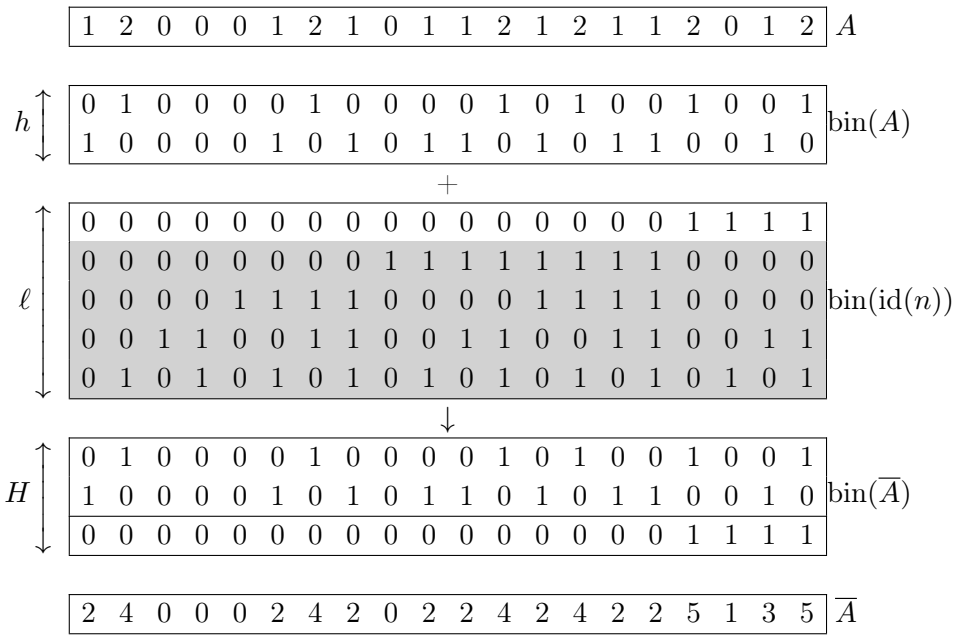
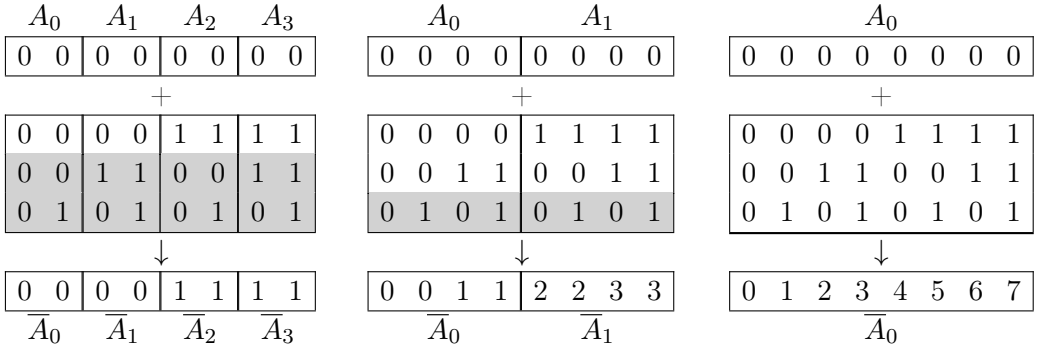


Figura 4.3: Composizione della codifica binaria a $h = 2$ bit di A col primo $H - h = 1$ bit della codifica binaria a $\ell = 5$ bit di $\mathrm{id}(n)$; il risultato è la codifica binaria a $H = 3$ bit di $\overline{A}$.

$A[p] = 0$ per ogni $p = 0, \ldots, n-1$. Banalmente, le classi sono equamente distribuite

sia lungo A, sia lungo ogni A_j; tuttavia, è anche la situazione più sfavorevole nella soluzione (c), siccome l'unica classe (0) conta un numero di elementi in ogni A_j pari alla lunghezza degli A_j stessi. Nel ricondurci a $k \approx K$ come abbiamo appena descritto, osserviamo che le classi sono ancora equamente distribuite lungo $\overline{A}$, ma a seconda del valore di K, n, non più necessariamente lungo ogni $\overline{A}_j$ (fig. 4.4).



(a) $n = 8, K = 2$ (b) $n = 8, K = 4$ (c) $n = 8, K = 8$

Figura 4.4

Più in generale, se partiamo da $k \leq K$ e le classi sono equamente distribuite lungo gli A_j, allora quanto più $K \approx n$, i.e. quanto più $H \approx \ell$, tanto più possiamo assumere che lo siano ancora nel ricondurci a $k \approx K$ come abbiamo appena descritto. Ricordando che, per semplicità, abbiamo assunto che la lunghezza degli A_j fosse pari a K, la seconda idea è di passare non da A a $\overline{A}$, bensì da ogni A_j a ogni $\overline{A}_j$ per composizione di $H - h$ bit fittizi tratti da una sequenza $\mathrm{id}(n/m)$ di $\lceil \log_2(n/m) \rceil = \log_2 K = H$ bit (fig. 4.5). Di nuovo, a scopo dimostrativo, consideriamo il caso particolare $k = 1$, i.e. $h = 0$, e

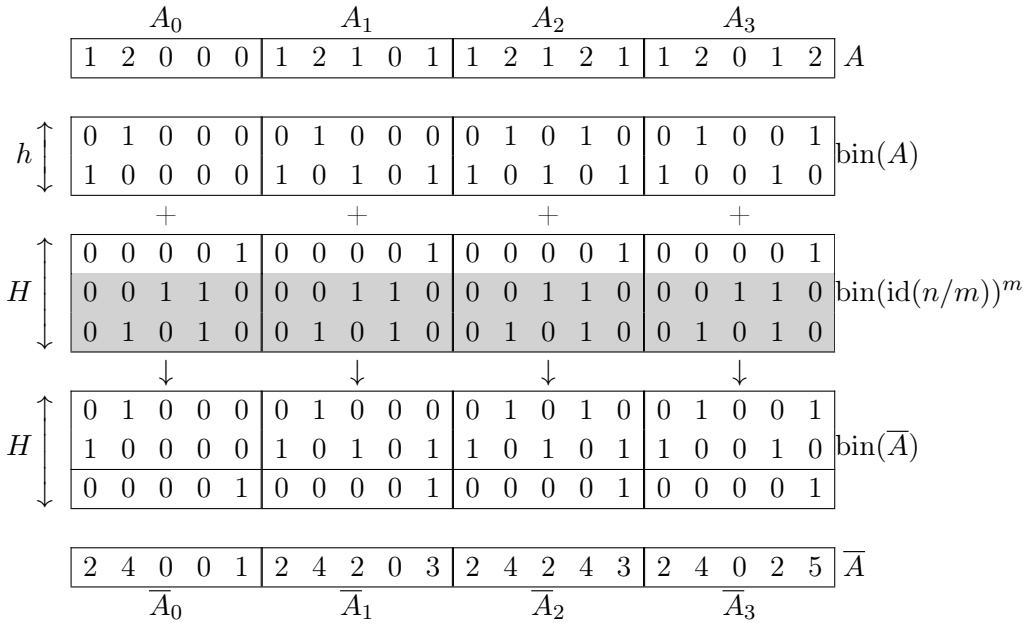
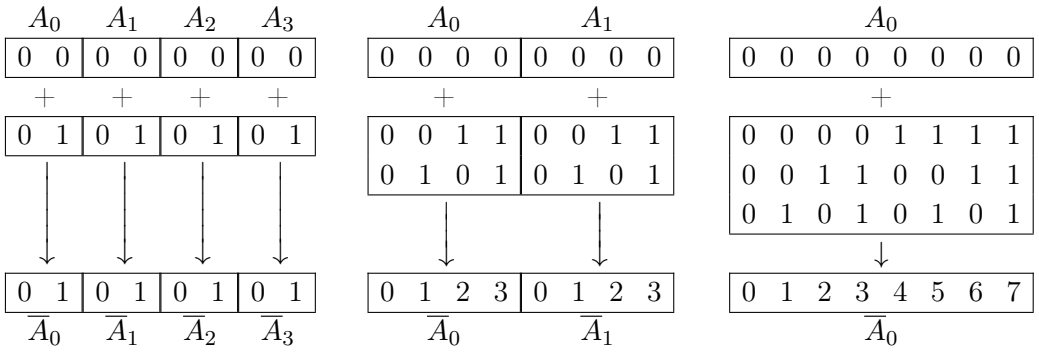


Figura 4.5: Composizione della codifica binaria a $h = 2$ bit di ogni A_j col primo $H - h = 1$ bit della codifica binaria a $H = 3$ bit di $\mathrm{id}(n/m)$; il risultato è la codifica binaria a $H = 3$ bit di ogni $\overline{A}_j$.

quindi $A[p] = 0$ per ogni $p = 0, \ldots, n-1$. Nel ricondurci a $k \approx K$ come abbiamo appena descritto, osserviamo che le classi sono ancora equamente distribuite sia lungo $\overline{A}$, sia lungo ogni $\overline{A}_j$, a prescindere dal valore di K, n (fig. 4.6). Il problema è che ordinare (in



(a) $n = 8, K = 2$ (b) $n = 8, K = 4$ (c) $n = 8, K = 8$

Figura 4.6

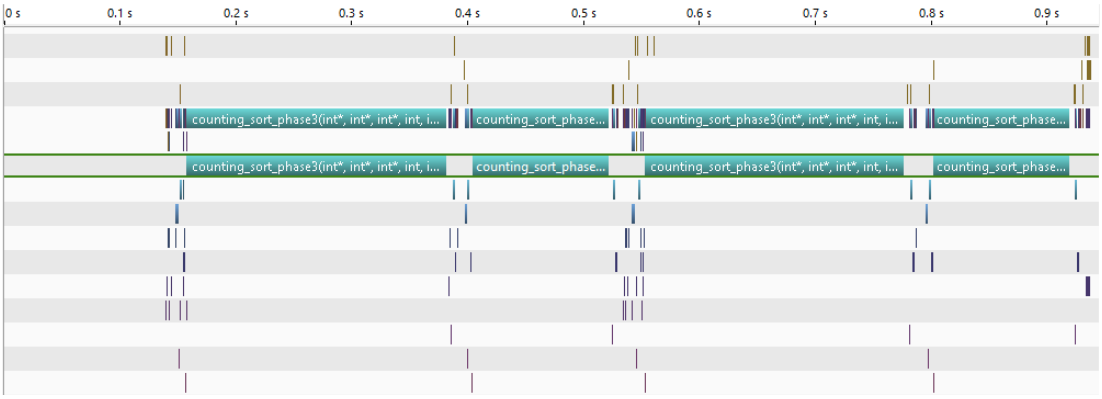
modo stabile) A potrebbe non essere più equivalente a ordinare (in modo stabile) $\overline{A}$. Non ci soffermeremo sulla questione, ma allo scopo di ripristinare tale equivalenza, una soluzione è di suddividere A in segmenti A_j più complessi. Lo svantaggio è che in una tale suddivisione, gli accessi alla memoria globale non sono più coalescenti, quindi per valori non critici di K, n, si osserva una leggera perdita temporale; tuttavia, per valori critici di K, n, si osserva un guadagno temporale di almeno un ordine di grandezza (cfr. fig. 4.7a e fig. 4.7b).

Ora, il problema è che non possiamo sapere come siano distribuite le classi di elementi, quantomeno non prima di aver calcolato la fase 1. La terza idea è di fissare $\overline{K} = 2^{\overline{H}}$ sufficientemente piccolo affinché l'eventuale distribuzione iniqua dei bit effettivi sia compensata dalla più equa distribuzione dei bit fittizi, e ricondurci a $k \leq \overline{K}$ come abbiamo descritto in precedenza; quindi, partendo da $k \leq K$, ci riconduciamo a $k \approx K$ come abbiamo appena descritto. Lo svantaggio è che potremmo dover risolvere un maggior numero di istanze di CountingSort; nondimeno, in genere, si osserva un ulteriore guadagno temporale (cfr. fig. 4.7b e fig. 4.7c).

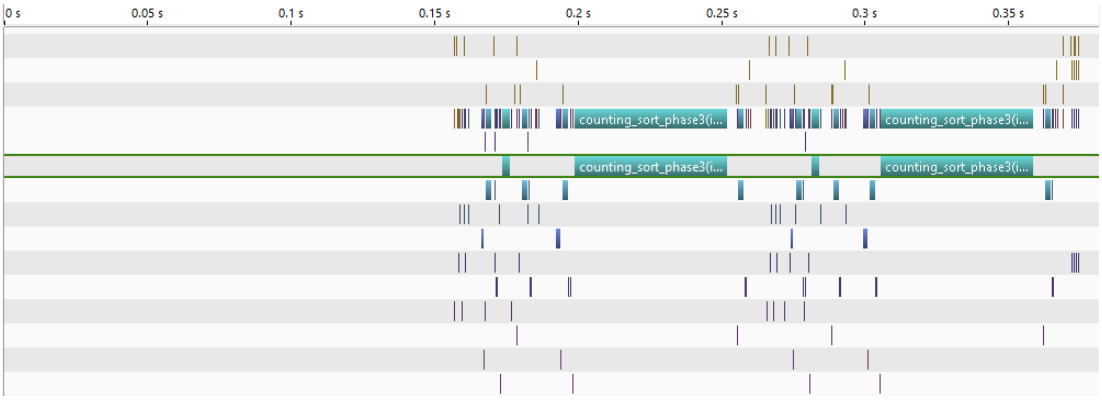
4.1.2 Implementazione alternativa: decomposizione "bottom-up"

Abbiamo già accennato alle decomposizioni "top-down" e "bottom-up". Con riferimento alle figure 4.1 e 4.2, la decomposizione "bottom-up" appare più semplice da implementare di quella "top-down". Infatti, nella decomposizione "top-down", il lavoro dei segmenti ottenuti dev'essere ridistribuito in qualche modo. I segmenti possono avere lunghezza molto variabile, come testimonia l'esempio, e non necessariamente multipla della dimensione di un blocco. Quindi, a ogni passo, ridistribuire il lavoro di tutti i segmenti su una stessa griglia potrebbe introdurre frammentazione interna (divergenza), mentre ridistribuire il lavoro di ogni segmento su una griglia diversa potrebbe introdurre frammentazione esterna (overhead). Viceversa, nella decomposizione "bottom-up", non si ottengono segmenti e il lavoro non dev'essere in alcun modo ridistribuito.

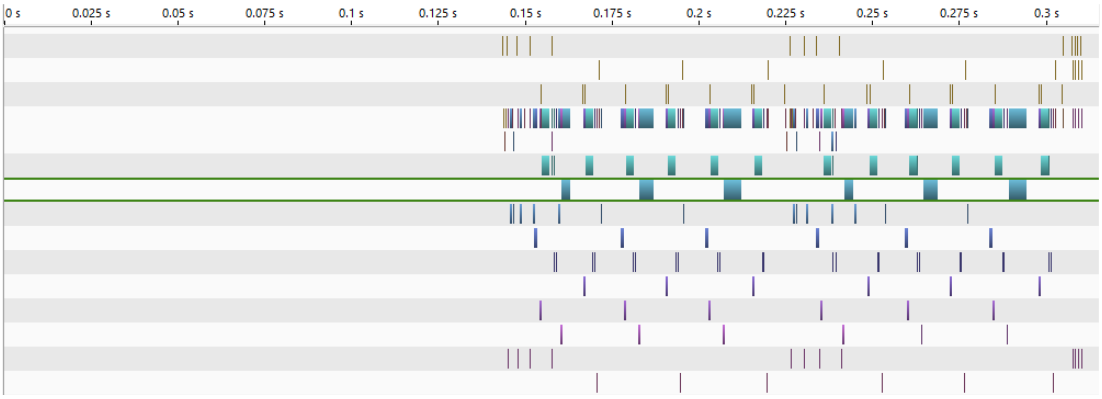
Di seguito, ci concentriamo sulla decomposizione "bottom-up". Abbiamo detto che ogni passo della decomposizione può essere calcolato come coppia di estrazioni complementari. Siano ancora A l'array e n la sua lunghezza. Di nuovo, immaginiamo di



(a) $k \leq K$



(b) $k \approx K$



(c) $k \leq \overline{K} \rightarrow k \approx K$

Figura 4.7: Profilo parziale dell'algoritmo di Moore eseguito sull'automa di $3 \cdot 10^5$ stati e 3 simboli che rappresenta il caso migliore, con segmenti di 128 elementi e blocchi di 128 thread, ottenuto con `nvvp`. In evidenza è la fase 3 di CountingSort nella soluzione (c).

suddividere A in m segmenti $A_0, \ldots, A_{m-1}$ della stessa lunghezza n/m, e lanciamo una griglia unidimensionale di m blocchi unidimensionali, per semplicità, di n/m thread ciascuno, dove s'intende che ogni blocco j opera sul segmento A_j. Sappiamo che entrambe le estrazioni di ogni coppia possono essere calcolate come segue:

1. Si calcola un array M di lunghezza n tale che per ogni $p = 0, \ldots, n-1$,
$$M[p] = \begin{cases} 1 & \text{se } A[p] \text{ soddisfa le condizioni di estrazione} \\ 0 & \text{altrimenti} \end{cases}$$
2. Si calcola la prefix-sum *esclusiva* P di M;
3. Si estrae ogni elemento $A[p]$ tale che $M[p] = 1$ e lo si inserisce in posizione $P[p]$:

```
1  int p = threadIdx.x;
2  int q = blockIdx.x * blockDim.x + p;

3  if (M[q] == 1)
4       insert A_j[p] at position P[q]
```

Possiamo fare di meglio calcolando ogni passo come una singola estrazione modificata:

1. Si calcola un array M di lunghezza n tale che per ogni $p = 0, \ldots, n-1$,
$$M[p] = \begin{cases} +1 & \text{se } A[p] \text{ soddisfa le condizioni di estrazione} \\ -1 & \text{altrimenti} \end{cases}$$
2. Si calcola la prefix-sum *inclusiva* P di M;
3. Si estrae ogni elemento $A[p]$: se $M[p] = +1$, allora lo si inserisce in certuna posizione i_p^+, altrimenti $M[p] = -1$ e lo si inserisce in cert'altra posizione i_p^-, dove i_p^+, i_p^- sono funzioni di $P[p]$.

Vorremmo che gli elementi di A che soddisfano la condizione di estrazione fossero inseriti nella prima parte dell'array, quelli che non la soddisfano nella seconda. Osserviamo che per ogni $p = 0, \ldots, n-1$, $P[p]$ è la differenza fra il numero di elementi di $A[0, p]$ che soddisfano la condizione di estrazione e il numero di quelli che non la soddisfano. Siano quindi x^+, risp. x^- il numero di elementi di A che soddisfano la condizione di estrazione, risp. che non la soddisfano; evidentemente,

$$\begin{aligned} x^+ + x^- &= n \\ x^+ - x^- &= P[n-1] \end{aligned}$$

Ne consegue che

$$x^+ = \frac{n + P[n-1]}{2} \qquad x^- = \frac{n - P[n-1]}{2}$$

Vorremmo che gli elementi di A che soddisfano la condizione di estrazione fossero inseriti a partire dalla posizione $y^+ = 0$, quelli che non la soddisfano a partire dalla posizione $y^- = x^+$. Non è difficile vedere che

$$\begin{aligned} i_p^+ &= y^+ + (p + P[p] + 1)/2 - 1 \\ i_p^- &= y^- + (p - P[p] + 1)/2 - 1 \end{aligned}$$

Quindi, modifichiamo la procedura come segue:

```
int p = threadIdx.x;
int q = blockIdx.x * blockDim.x + p;
int y_pos = 0;
int y_neg = (n + P[n - 1])/2;

if (M[q] == 1)
    insert A_j[p] at position y_pos + (q + P[q] + 1)/2 - 1
else
    insert A_j[p] at position y_neg + (q - P[q] + 1)/2 - 1
```

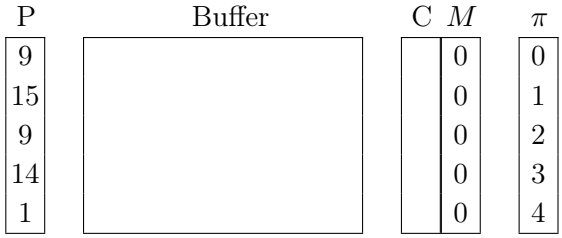
4.2 RadixSort parallelo

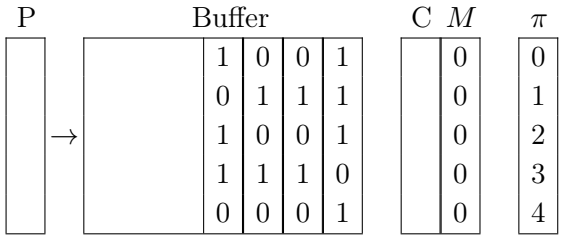
Siano ancora $h = \lceil \log_2 k \rceil$ e $H = \lceil \log_2 K \rceil$, e precisamente $K = 2^H$. In precedenza, abbiamo ragionato che ogni istanza di CountingSort su classi "a grana fine" $(0, \ldots, k-1)$ può essere decomposta in una sequenza di istanze di CountingSort su classi "a grana grossa" $(0, 1)$, e quindi ricomposta in una sequenza di $\lceil h/H \rceil$ istanze di CountingSort su classi "a grana media" $(0, \ldots, K-1)$. Per un'estensione di tale ragionamento, ogni istanza di RadixSort su classi "a grana fine" può essere decomposta, e quindi ricomposta in una sequenza di $\lceil mh/H \rceil$ istanze di CountingSort su classi "a grana media", dove m è la lunghezza degli array su cui è definita l'istanza di RadixSort.

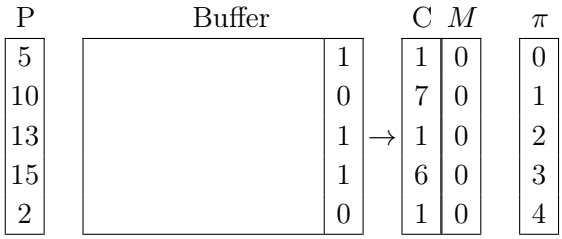
7	5	9	0	1	1	1	0	1	0	1	1	0	1	1
12	10	7	1	1	0	0	1	0	1	0	0	1	1	1
4	13	9	= 0	1	0	0	1	1	0	1	1	0	0	1
14	15	14	1	1	1	0	1	1	1	1	1	1	1	0
9	2	1	1	0	0	1	0	0	1	0	0	0	0	1

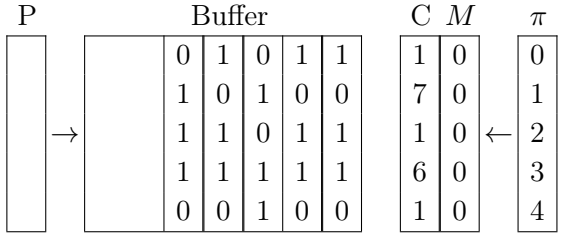
0	1	1	1	0	1	0	1	1	0	0	1	3	5	3	1
1	1	0	0	1	0	1	0	0	1	1	1	6	2	4	7
0	1	0	0	1	1	0	1	1	0	0	1	= 2	3	3	1
1	1	1	0	1	1	1	1	1	1	1	0	7	3	7	6
1	0	0	1	0	0	1	0	0	0	0	1	4	4	4	1

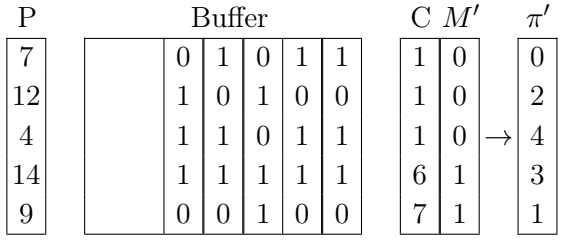
Per natura dell'algoritmo di Moore, possiamo vedere l'istanza di RadixSort contestuale a tale algoritmo come un processo del tipo produttore-consumatore, in cui il produttore (P) genera una sequenza di m array di lunghezza n, mentre il consumatore (C) li ordina risolvendo una sequenza di istanze di CountingSort. Precisamente, il produttore decompone ogni array generato su classi "a grana fine" in una sequenza di h array su classi "a grana grossa" e li scrive nel buffer, mentre il consumatore legge dal buffer ogni sequenza di H array su classi "a grana grossa" e li ricompone in un array su classi "a grana media". Inoltre, prima di ordinare ogni tale array, il consumatore vi applica la permutazione corrente π, e la aggiorna dopo averlo ordinato. Infine, nel contesto dell'algoritmo di Moore, è utile mantenere un array binario M della stessa lunghezza degli array generati tale che per ogni $j = 0, \ldots, n-1$, $M[j] = 0$ se la porzione dell'array j-mo finora consumata è uguale a quella dell'array $(j+1)$-mo su cui è definita l'istanza di RadixSort, $M[j] = 1$ altrimenti.

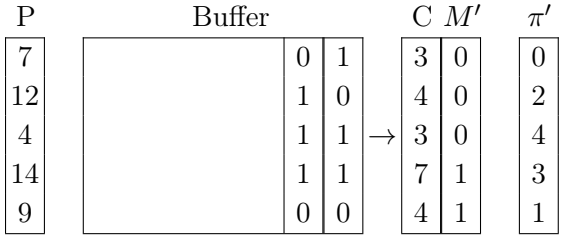

P
Buffer
C
M
π
9
15
9
14
1
0
0
0
0
0
0
1
2
3
4

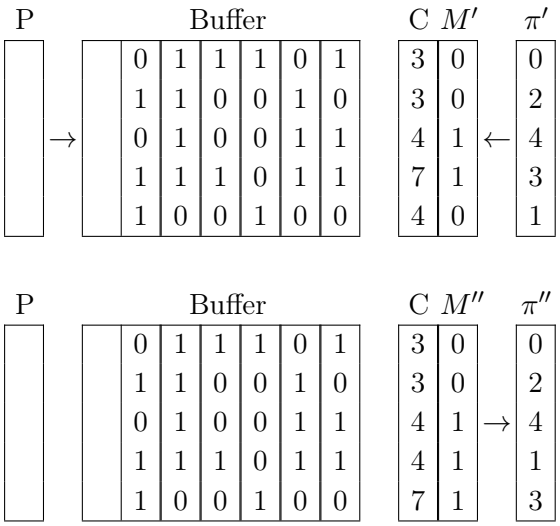

P
Buffer
C
M
π
1 0 0 1
0 1 1 1
1 0 0 1
1 1 1 0
0 0 0 1
0
0
0
0
0
0
1
2
3
4


P
Buffer
C
M
π
5
10
13
15
2
1
0
1
1
0
1 0
7 0
1 0
6 0
1 0
0
1
2
3
4


P
Buffer
C
M
π
0 1 0 1 1
1 0 1 0 0
1 1 0 1 1
1 1 1 1 1
0 0 1 0 0
1 0
7 0
1 0
6 0
1 0
0
1
2
3
4


P
Buffer
C
M'
π'
7
12
4
14
9
0 1 0 1 1
1 0 1 0 0
1 1 0 1 1
1 1 1 1 1
0 0 1 0 0
1 0
1 0
1 0
6 1
7 1
0
2
4
3
1


P
Buffer
C
M'
π'
7
12
4
14
9
0 1
1 0
1 1
1 1
0 0
3 0
4 0
3 0
7 1
4 1
0
2
4
3
1



E così via. Questa soluzione non è dissimile dalla seconda e terza versione dell'implementazione OpenMP (sez. 3.2 e 3.3) e consente di risparmiare memoria da $\mathcal{O}(nm)$ a $\mathcal{O}(n)$, al costo di calcolare a ogni passo la prefix-sum inclusiva di M per il mantenimento di M stesso. Si noti che, risolta l'istanza di RadixSort contestuale all'algoritmo di Moore, tale prefix-sum, opportunamente riordinata, è proprio il raffinamento della relazione di equivalenza sugli stati dell'automa che induce le classi di elementi.

4.2.1 Produttore-consumatore

Discutiamo brevemente del processo del tipo produttore-consumatore di cui alla descrizione di RadixSort parallelo. L'implementazione del processo si basa su un'opportuna combinazione di stream, eventi e direttive OpenMP.

Introduciamo due pthread: l'uno per il produttore e l'altro per il consumatore. Immaginiamo di sostituire lo stream di default con due stream privati, associati l'uno al pthread del produttore e l'altro al pthread del consumatore: useremo tali stream per la produzione e consumazione effettive; inoltre, introduciamo un terzo stream condiviso `s`, il quale useremo per la lettura e scrittura sul buffer:

```
1  cudaStream_t s;
2  cudaStreamCreate(&s);
3  volatile int offt = 0;
4  volatile int done = 0;

5  #pragma omp parallel sections shared(offt, done, s)
6  {
7      #pragma omp section
8      {
9          producer
10     }
11     #pragma omp section
12     {
13         consumer
14     }
15 }
```

```
16  cudaStreamDestroy(s);
```

Le variabili intere `offt`, risp. `done` indicano la dimensione attuale (offset) del buffer, risp. la condizione di terminazione del processo. In quanto condivise, tali variabili sono dichiarate volatili, siccome vogliamo che ogni loro modifica da parte del produttore sia immediatamente visibile al consumatore, e viceversa.

Di seguito, vediamo nel dettaglio il produttore; il consumatore è del tutto analogo. Per poterlo scrivere sul buffer, ogni elemento successivo dev'essere prima prodotto. Se è il primo elemento, allora possiamo produrlo senz'altro; altrimenti, dobbiamo attendere che sia terminata la scrittura sul buffer di quello precedente (evento `e_o`):

```
1   cudaEvent_t e_i, e_o;
2   cudaEventCreate(&e_o);
3   #pragma omp critical (producer_consumer)
4       cudaEventRecord(e_o, s);

5   while (true) {
6       cudaEventCreate(&e_i);
7       cudaStreamWaitEvent(0, e_o);
8       produce element X of length h using the default stream
9       cudaEventRecord(e_i, 0);
10      cudaEventDestroy(e_o);
11      ⋮
```

dove `X` è l'elemento prodotto e `h` è la sua lunghezza. Se l'elemento ha lunghezza nulla, allora assumiamo che il processo sia terminato:

```
10      ⋮
11      if (h == 0) {
12          #pragma omp critical (producer_consumer)
13              *done = 1;
14          cudaEventDestroy(e_i);
15          break;
16      }
17      ⋮
```

altrimenti, lo scriviamo nel buffer. Prima di scrivercelo, dobbiamo attendere che vi sia abbastanza spazio:

```
16      ⋮
17      cudaEventCreate(&e_o);
18      while (true) {
19          int temp_offt;
20          #pragma omp critical (producer_consumer)
21              temp_offt = *offt;
22          if (temp_offt + h > BUFFER_SIZE)
23              continue;
24          ⋮
```

inoltre, dobbiamo attendere che sia terminata la sua produzione (evento `e_i`):

```
23              ⋮
24              #pragma omp critical (producer_consumer)
25              {
26                  cudaStreamWaitEvent(s, e_i);
27                  write element X on the buffer using stream s
28                  cudaEventRecord(e_o, s);
29                  *offt += h;
30              }
31          }
32          cudaEventDestroy(e_i);
33      }
```

Le sezioni critiche, combinate con la serializzazione implicita delle letture e scritture sul buffer, garantiscono che le scritture da parte del produttore siano eseguite in un ordine consistente rispetto alle letture da parte del consumatore, oltreché rispetto alle produzioni e consumazioni effettive.

4.3 Algoritmo di Moore parallelo

Di seguito, per completezza, mettiamo assieme quanto discusso in precedenza e diamo una breve descrizione dell'algoritmo di Moore parallelo. La struttura di base della variante parallela dell'algoritmo resta sostanzialmente la stessa di quella seriale, siccome la maggior parte della parallelizzazione avviene all'interno di RadixSort e, in ultima analisi, di CountingSort:

```
1   int k1 = 0;
2   int k2 = 2;
3   cudaMemcpy(Tilde, F, sizeof(int) * n,
        cudaMemcpyHostToDevice);
4   while (k1 < k2) {
5       k1 = k2;
6       k2 = radix_sort(Q, Delta, Tilde, k1, T, M, P);
7       cudaMemcpy(Tilde, P, sizeof(int) * n,
            cudaMemcpyDeviceToDevice);
8   }
```

In particolare, il ciclo `while` è eseguito in seriale.

Sia $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ l'automa in input, dove per convenzione $Q = \{0, \ldots, n-1\}$ e $\Sigma = \{0, .., m-1\}$. Sono definite le variabili host `Q`, `F` di tipo `int *`, e la variabile host `Delta` di tipo `int **`. Il vettore `Q` rappresenta l'insieme Q. Ogni elemento `Delta[s]` del vettore `Delta` è a sua volta un vettore che rappresenta la funzione di transizione parziale $\delta_s(q) := \delta(q, s)$. Il vettore `F` rappresenta l'insieme F in forma di array binario. In conformità con la notazione di cui alla trattazione di OpenMP, denotiamo con $\sim$ la relazione di Myhill-Nerode, rappresentata dalla variabile device vettoriale `Tilde`. Sono inoltre definite le variabili device `T`, `M`, `P` di tipo `int *`. Il vettore `T` rappresenta la permutazione corrispondente all'ordinamento (stabile), risultato dell'applicazione di RadixSort alle cosiddette *firme* degli stati di $\mathcal{A}$. Con riferimento alla trattazione di OpenMP, ricordiamo che la firma di uno stato q di $\mathcal{A}$ è l'applicazione di `Tilde` al q-

mo elemento del vettore `Q`, i.e. q stesso (primo *digit*), oppure al q-mo elemento di un vettore della forma `Delta[s]`, i.e. $\delta(q, s)$ (restanti m *digit*). Il vettore `M` rappresenta l'array binario M di cui alla descrizione di RadixSort parallelo. Il vettore `P` rappresenta la prefix-sum inclusiva di M, riordinata secondo la permutazione inversa di quella rappresentata dal vettore `T`.

Il q-mo elemento di `Tilde` è la classe di equivalenza dello stato q di $\mathcal{A}$ secondo il raffinamento corrente di $\sim$. Il raffinamento iniziale di $\sim$ coincide con quello più grossolano, tale che discrimina gli stati finali e non-finali di $\mathcal{A}$ (linea 3). A ogni iterazione del ciclo `while`, come accennato discutendo di RadixSort parallelo, il raffinamento di $\sim$ coincide con la prefix-sum inclusiva di M, opportunamente riordinata (linea 7); si noti che tale coincidenza vale in particolare al termine del ciclo. Il raffinamento finale di $\sim$ coincide con quello più fine, i.e. con la relazione $\sim$ stessa. Al termine del ciclo `while`, il valore finale delle variabili `T`, `M`, `P` è usato per estrarre in parallelo i rappresentanti delle classi di equivalenza secondo $\sim$, i quali a loro volta sono usati nella costruzione dell'automa di output, i.e. l'automa quoziente $\mathcal{A}/_{\sim}$; al di là dell'estrazione dei rappresentanti, tali variabili possono essere considerate prodotti di scarto di RadixSort.

Restano da descrivere la produzione e consumazione effettive delle firme. Nella produzione, si genera la sequenza dei *digit* delle firme e li si scrive nel buffer; nella consumazione, si leggono porzioni di firma dal buffer e le si ordina. La generazione di ogni *digit* prevede la copia di un vettore in memoria device—il vettore `Q`, oppure un vettore della forma `Delta[s]`—e quindi l'applicazione di `Tilde` a tale vettore. Affinché l'ordinamento (stabile) sia corretto, le porzioni di firma da ordinare devono essere generate al contrario, dal *digit* $(m + 1)$-mo fino al primo. Ne consegue che `Tilde` è applicata ai vettori `Delta[m - 1]`, ..., `Delta[0]` nei turni di produzione risp. dal primo all'm-mo, e al vettore `Q` nell'$(m + 1)$-mo nonché ultimo turno. L'ordinamento di ogni porzione di firma prevede, oltreché l'ordinamento in sé, il ricalcolo del valore delle variabili `M` e `P` a ogni turno di consumazione successivo, usando in particolare il valore della variabile `P` al turno precedente.

4.4 Note implementative

L'attuale implementazione potrebbe presentare ancora qualche sporadico difetto di sincronizzazione. Con riferimento alla descrizione di RadixSort parallelo, il corretto funzionamento del buffer si basa sull'assunto che un intero sia rappresentato in complemento a due.

La struttura dei file è la seguente:

```
CUDA/
└── source/
    ├── automaton.cu
    ├── automaton.cuh
    ├── common.cu
    ├── common.cuh
    ├── counting_sort.cu
    └── counting_sort.cuh
```

```
├── cuda.cu
├── Makefile
├── moore.cu
├── moore.cuh
├── prefix_sum.cu
├── prefix_sum.cuh
├── producer_consumer.cu
├── producer_consumer.cuh
├── radix_sort.cu
└── radix_sort.cuh
```

L'eseguibile risultante dalla compilazione è denominato `cuda` e accetta le seguenti opzioni da linea di comando:

`--help`	Stampa un messaggio di aiuto
`--verbose`	Stampa informazioni sull'esecuzione

- Selezione della modalità di input (I):

`-I decode`	Legge un automa dall'input e lo decodifica (default)
`-I best-case <n> <m>`	Genera l'automa di $3n$ stati e m simboli che rappresenta il caso migliore ($\mathcal{O}(mn)$)
`-I worst-case <n> <m>`	Genera l'automa di $3n$ stati e m simboli che rappresenta il caso peggiore ($\mathcal{O}(mn^2)$)
`-I random <n> <m>`	Genera un automa completo accessibile casuale di n stati e m simboli ($\mathcal{O}(mn \log n)$)

- Selezione della modalità di azione (A):

`-A pass`	Non fa nulla (default)
`-A minimize <m> <t> <r>`	Esegue l'algoritmo di Moore sull'automa in input con t thread e r stride per blocco; per semplicità implementativa, t dev'essere una potenza di 2 compresa fra 64 e 1024. A ogni passo dell'istanza di RadixSort, usa:
`<m>=horizontal-a`	CountingSort, fase 2 orizzontale, fasi 1-3 soluzione (a)
`<m>=horizontal-b`	CountingSort, fase 2 orizzontale, fasi 1-3 soluzione (b)
`<m>=horizontal-c`	CountingSort, fase 2 orizzontale, fasi 1-3 soluzione (c)
`<m>=vertical-a`	CountingSort, fase 2 verticale, fasi 1-3 soluzione (a)
`<m>=vertical-b`	CountingSort, fase 2 verticale, fasi 1-3 soluzione (b)
`<m>=vertical-c`	CountingSort, fase 2 verticale, fasi 1-3 soluzione (c)
`<m>=decompose`	Decomposizione "bottom-up"
	consumando $\log_2 K = \log_2 t$ bit se è usato CountingSort, un numero di bit pari alla dimensione del buffer se è usata la decomposizione "bottom-up".

- Selezione della modalità di output (O):

`-O encode`	Codifica l'automa risultante e lo scrive nell'output (default)
`-O pretty-print`	Stampa l'automa risultante nell'output
`-O pass`	Non fa nulla

Un automa letto dall'input dev'essere codificato come segue:

`n=<n>`	Numero di stati
`m=<m>`	Numero di simboli
`<p_1> <s_1> <q_1>`	
⋮	
`<p_t> <s_t> <q_t>`	Relazione di transizione $\Delta = \{(p_i, s_i, q_i)\}_{i=1}^{t}$
`initial <q_0>`	Stato iniziale q_0
`final <q_1>`	
⋮	
`final <q_f>`	Insieme degli stati finali $F = \{q_i\}_{i=1}^{f}$

Gli stati devono essere specificati nella forma $0, 1, 2, \ldots, n-1$, mentre i simboli devono essere specificati nella forma $a, b, c, \ldots, \mathrm{ch}(m-1)$, dove $\mathrm{ch}(i)$ è l'i-mo carattere dell'alfabeto. Si suppone che l'automa sia accessibile, i.e. tale che ogni stato è raggiungibile dallo stato iniziale. Se l'automa non è completo, allora è completato con transizioni della forma $q \xrightarrow{s} q$.

4.5 Risultati e discussione

I tempi sono stati calcolati su una macchina equipaggiata con una CPU *Intel Core i5-3317U @ 1.70Ghz* e una GPU *NVIDIA GeForce GT 635M* (compute capability 2.1). Denotiamo con t, risp. con r il numero di thread, risp. di stride per blocco. La GPU su cui sono stati calcolati i tempi possiede 2 SM, ciascuno dei quali può schedulare al più 8 blocchi e 1536 thread. Ne consegue che per saturare gli SM, converrebbe scegliere $t = 1536/8 = 192$. Siccome per semplicità implementativa, il numero di thread per blocco dev'essere una potenza di 2, scegliamo $t = 128$. Con riferimento alla descrizione di RadixSort parallelo, denotiamo inoltre con $\overline{H}$ il numero di bit consumati a ogni passo dell'istanza di RadixSort. Se si usa CountingSort, allora dobbiamo scegliere $\overline{H} \leq H$, e in genere sceglieremo $\overline{H} = H$; tuttavia, alcune soluzioni implementative traggono vantaggio dal consumo di un minor numero di bit a ogni passo (v. sez. 4.1.1), quindi per completezza, scegliamo valori diversi di $\overline{H}$. Se invece si usa la decomposizione "bottom-up", allora possiamo scegliere $\overline{H} > H$, e in genere sceglieremo $\overline{H}$ massimo, i.e. pari alla dimensione del buffer.

Ricordiamo che $K = t = 128$, da cui segue che $H = \log_2 K = 7$; inoltre, sulla macchina su cui sono stati calcolati i tempi, il buffer ha una dimensione pari a 31 bit. I tempi sono riportati in tabella 4.6: le righe dalla prima alla penultima riportano i tempi

(in secondi) usando CountingSort nelle diverse soluzioni implementative, mentre l'ultima riga riporta i tempi (in secondi) usando la decomposizione "bottom-up" (indichiamo con ∞ i tempi che superano i 1000 secondi).

$\overline{H}$	Fasi		ist1B	ist2B	ist3B	ist1S	ist2S	ist3S
	2	1-3	$\mathcal{A}_{2000000,30}$	$\mathcal{B}_{5000,20}$	$\mathcal{C}_{2000000,30}$	$\mathcal{A}_{20000000,2}$	$\mathcal{B}_{15000,2}$	$\mathcal{C}_{20000000,2}$
7	O	a	4.13	250.76	21.43	-*	341.31	39.58
	O	b	2.63	309.76	157.72	-*	∞	231.99
	O	c	65.26	∞	18.73	-*	∞	36.04
	V	a	4.08	241.10	23.04	-*	354.80	40.08
	V	b	2.60	301.61	161.29	-*	∞	232.89
	V	c	67.51	∞	18.69	-*	∞	37.96
5	O	a	5.41	334.53	29.34	-*	456.07	53.95
	O	b	3.28	260.53	90.55	-*	658.21	143.47
	O	c	12.00	801.80	25.25	-*	∞	45.82
	V	a	5.40	333.62	30.20	-*	465.79	53.40
	V	b	3.17	257.90	92.52	-*	668.47	142.26
	V	c	11.99	792.99	26.23	-*	∞	44.58
3	O	a	8.30	534.49	51.67	-*	711.45	85.62
	O	b	4.60	330.17	66.08	-*	567.69	105.04
	O	c	5.07	341.05	41.99	-*	463.44	68.32
	V	a	8.05	539.99	51.17	-*	726.96	84.52
	V	b	4.57	341.18	65.72	-*	578.09	103.31
	V	c	4.99	344.45	42.08	-*	470.78	68.31
31	-	-	5.48	346.66	32.95	-*	527.79	57.59
Iterazioni			2	10000	3	2	30000	6

*VRAM insufficiente

Tabella 4.6: $t = 128, r = 4$

Riguardo a CountingSort, facciamo le seguenti osservazioni:

- La soluzione (a) delle fasi 1-3 sembra quella più efficiente su quasi tutte le istanze considerate, con qualche eccezione. Ipotizziamo che scegliendo opportunamente $t, r, \overline{H}$, convenga la soluzione (b) nel caso in cui le classi dell'automa da minimizzare siano distribuite in modo particolarmente iniquo, come sulle istanze ist1B e ist1S, mentre convenga la soluzione (c) nel caso in cui siano distribuite in modo particolarmente equo, come sulle istanze ist3B e ist3S; in tutti gli altri casi, oppure quando la distribuzione delle classi dell'automa da minimizzare non sia nota, conviene la soluzione (a).
- In linea con le aspettative, se $\overline{H} < H$, allora la soluzione (a) delle fasi 1-3 risulta svantaggiata, mentre le soluzioni (b) e (c) traggono vantaggio dal consumo di un minor numero di bit a ogni passo, con l'eccezione della soluzione (b) nel caso in cui le classi dell'automa da minimizzare siano distribuite in modo particolarmente iniquo, come sulle istanze ist1B e ist1S, e della soluzione (c) nel caso in cui siano distribuite in modo particolarmente equo, come sulle istanze ist3B e ist3S (v. sez. 4.1.1).

- Le varianti “orizzontale” e “verticale” della fase 2 sembrano egualmente efficienti su quasi tutte le istanze considerate, di nuovo, con qualche eccezione. Ipotizziamo che la variante “orizzontale” convenga rispetto a quella “verticale” nel caso in cui il numero di simboli dell’automa da minimizzare sia sufficientemente basso, e viceversa, la variante “verticale” convenga rispetto a quella “orizzontale” nel caso in cui sia sufficientemente alto; ipotizziamo inoltre che una variante convenga tanto più sensibilmente rispetto all’altra quanto più è alto il numero di iterazioni dell’algoritmo di Moore.

Riguardo alla decomposizione “bottom-up”, osserviamo che non sembra altrettanto efficiente di CountingSort pur ottenendo comunque risultati soddisfacenti, in particolare rispetto alle soluzioni (b) e (c) delle fasi 1-3.

Capitolo 5

Conclusioni

		ist1B	ist2B	ist3B	ist1S	ist2S	ist3S
programma seriale	tempo (s)	27.2	91.3	39.6	19.7	158.74	109.9
miglior risultato OpenMP	programma	3	3	3	3	5	4
	tempo (s)	6.8	47.7	10.7	9.0	63.9	40.8
	risparmio (%)	75 %	48 %	73 %	54 %	60 %	63 %
miglior risultato CUDA	programma	7Vb	7Va	7Vc	7Ob	7Oa	7Oc
	tempo (s)	2.6	241.10	18.69	/	341.31	36.0
	risparmio (%)	90 %	- 164 %	52 %	/	- 115 %	67 %

Tabella 5.1: Programmi migliori e confronto fra architetture.

	ist1B	ist2B	ist3B	ist1S	ist2S	ist3S
programma	7Vb	7Va	7Vc	7Ob	7Oa	7Oc
tempo (s)	0.3	41.51	1.87	0.51	28.88	3.63
risparmio (%)	99 %	55 %	95 %	97 %	81 %	97 %

Tabella 5.2: Risultati di esecuzione del programma CUDA su una GPU più performante.

Risultati finali

In tabella 5.1 mettiamo a confronto i tempi del programma seriale, i tempi dei migliori programmi OpenMP così come discusso nella sezione 3.7 e i tempi delle migliori configurazioni del programma CUDA così come discusso in sezione 4.5. I risultati sono relativi alle macchine presentate nelle rispettive sezioni. Nei `Makefile` (sia del codice OpenMP sia del codice CUDA), abbiamo incluso sei comandi (uno per ogni istanza test) della forma:

```
make <nome_istanza>
```

Ognuno di questi comandi esegue la corrispondente istanza di test, con la configurazione della tabella 5.1. Ad esempio, nella cartella `OPENMP`, il comando `make ist3B` esegue il programma 3 sull'istanza ist3B. Nella cartella `CUDA/source`, il comando `make ist1B`

esegue il programma con fase 2 nella variante "verticale" e fasi 1-3 nella soluzione (b) sull'istanza ist1B. In tabella 5.1, rispetto al programma OpenMP, il programma CUDA sembra ottenere risultati peggiori in termini di prestazioni, con particolare riferimento agli automi che rappresentano il caso peggiore. Ipotizziamo che ciò sia dovuto in parte al programma in sé e in parte alla macchina su cui sono stati eseguiti i test. Abbiamo avuto modo di testare lo stesso programma su una macchina equipaggiata con una CPU *Intel® Xeon® E5-2609v4 1.7 GHz* e una GPU *NVIDIA GeForce® GTX1080 Ti*; i dati sono riportati in tabella 5.2. Come si evince dal confronto delle due tabelle, i risultati ottenuti sono nettamente migliori, a dimostrazione della scalabilità del programma.

Lavori futuri

Per quanto riguarda l'implementazione OpenMP, rimane da studiare la scalabilità del programma basato su RadixSort, nonchè testare versioni più efficienti di RadixSort come quelle proposte nel capitolo 4. Sempre su OpenMP, non abbiamo testato la possibilità di creare programmi che vengano eseguiti sul device.

Per quanto riguarda CUDA, se si dispone di due device, allora un possibile miglioramento dell'implementazione consiste nel suddividere ogni istanza di CountingSort in due sotto-istanze per esecuzione preliminare di un singolo passo della decomposizione "top-down"; si noti che un tale miglioramento non sarebbe dissimile dalla quarta versione dell'implementazione OpenMP (sez. 3.4). Si potrebbe inoltre testare la versione di RadixSort presentata nel capitolo 4 con librerie già note e scritte in CUDA.

Per quanto riguarda i risultati teorici, rimangono da investigare vari modi di calcolare $\sim_i$ usando meno *digit* possibile. I lemmi 1 e 2 rappresentano solo l'inizio di uno studio in questa direzione, ma per ora, la loro utilità pratica è limitata al caso in cui $m = 2$.