UNIVERSITÀ DEL SALENTO

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

ALGORITMI PARALLELI

Docente: Massimo Cafaro

Deterministic parallel selection algorithm on coarse-grained multicomputers

Autore: Emanuele Mele

Introduzione

Il problema della selezione della i-esima statistica d'ordine consiste nel selezionare l'i-esimo elemento più piccolo all'interno di un'array. Più formalmente possiamo dire che, dato un set A di n elementi distinti, il *rank* di un elemento $x \in A$ viene espresso come $rank(x, A)$ e rappresenta il numero di elementi minori o uguali a x in A. L'i-esima statistica d'ordine sarà data dall'elemento $x \in A$ tale per cui $rank(x, A) = i$. Il problema può essere risolto in modo banale effettuando un sorting in tempo $O(nlg(n))$ sfruttando l'algoritmo Merge Sort o Heap Sort [1] e successivamente accedendo alla cella i-esima in tempo $O(1)$. Tuttavia esiste un algoritmo sequenziale [2], che prende il nome di Selection, che nel caso peggiore impiega tempo lineare $O(n)$. In questo lavoro si è implementato un algoritmo parallelo che sfrutta la controparte sequenziale Selection e il concetto di mediana pesata per produrre per il modello *coarse-grained multicomputers* (CGM) — dove il numero di processori p è molto minore rispetto alla dimensione del problema n — una risoluzione al problema della selezione in tempo $O(log(p)(log(p) + p + n/p))$. L'algoritmo è pensato principalmente per funzionale in modo general purpose (GP), ma viene anche implementata una piccola variante con Radix Sort (RS) che, nelle giuste condizioni, effettua un ordinamento in tempo lineare $O(n)$, non modificando la complessità dell'algoritmo di selezione. Il medesimo elaborato riproduce ciò che è stato fatto in [3].

1 Background

Prima di procedere con l'analisi dell'algoritmo parallelo evidenziamo tutti i sotto algoritmi principali necessari al funzionamento del codice concorrente.

1.1 Selection

Nell'algoritmo parallelo di selezione andiamo a utilizzare anche la sua controparte sequenziale per svolgere delle fasi intermedie. L'algoritmo prende il nome di *Selection*. Verrà fornito in input il vettore degli elementi X e l'i-esima statistica d'ordine i desiderata. In output riceveremo il numero corrispondente all'i-esima statistica d'ordine cercata [Alg. 1].

Algorithm 1: Selection

Data: i, X
Result: i-th order statistic of X
$n \longleftarrow$ size of X
if $n = 1$ **then**
  return $X[0]$
Divide X into subarrays of 5 elements each
for *each subarray* **do**
  Sort the subarray with insertion sort and find the median
Create an array M of the found medians
$x \longleftarrow$ median of M
$X_{\text{left}}, X_{\text{right}} \longleftarrow$ Partition around x
$k = \text{rank}(x)$
if $i = k$ **then**
  return x
else if $i < k$ **then**
  return Selection(X_{left}, i)
else
  return Selection$(X_{\text{right}}, i - k)$

L'equazione di ricorrenza associata a questo algoritmo è $T(n) = T(n/5) + T(7n/10) + O(n)$ la cui soluzione è $O(n)$.

1.2 Weighted 3-Median

Per il corretto funzionamento dell'algoritmo parallelo di selezione dobbiamo introdurre il concetto relativo alla mediana pesata, questo ci sarà utile per un espansione del concetto di mediana utilizzata nell'algoritmo di selezione sequenziale. Questa modifica ci permette di scartare a ogni iterazione un minimo di 1/3 degli elementi. La dimostrazione realtiva a questa osservazione è discussa in [3]. Per n distinti elementi $x_1, x_2, ..., x_n$ con pesi positivi $w_1, w_2, ..., w_n$ tali per cui $\sum_{i=1}^{n} w_i = W$, la *lower weighted 3-median* è data dall'elemento x_k che soddisfa:

$$\sum_{i, x_i < x_k} w_i \leq \frac{W}{3} \tag{1}$$

$$\sum_{i, x_i > x_k} w_i \leq \frac{2W}{3} \tag{2}$$

Invece per n distinti elementi $x_1, x_2, ..., x_n$ con pesi positivi $w_1, w_2, ..., w_n$ tali per cui $sum_{i=1}^{n} w_i = W$, la *upper weighted 3-median* è data dall'elemento x_k che soddisfa:

$$\sum_{i, x_i < x_k} w_i \leq \frac{2W}{3} \tag{3}$$

$$\sum_{i, x_i > x_k} w_i \leq \frac{W}{3} \tag{4}$$

Sia n il numero totale di elementi in A, p il numero di processori e i la statistica d'ordine desiderata; all'inizio di ogni iterazione n_i sarà il numero totale di elementi rimanenti gestiti dal processo p_i, e vale $\sum_{i=0}^{p-1} n_i = N$. Lo pseudocodice per la ricerca della *lower weighted 3-median* il seguente è 2, prende in input il vettore di elementi X, il vettore dei pesi associati agli elementi Y, S rappresenta la somma dei pesi di tutti gli elementi con rango minore di s e E la somma dei pesi degli elementi maggiori di e, dove $[s, e]$ è un intervallo che inizialmente rappresenta tutto l'array. Resituisce in output la mediana pesata.

Algorithm 2: LowerWeighted3Median

Data: X,Y,S,E

Result: The lower weighted 3-median of X

find the median x_i of X

partition X around x_i

for $j = 0$ *to lenght[X]-1* **do**

  if $x_j < x_i$ **then**

    $A \longleftarrow A + w_j$

  else

    if $x_j > x_i$ **then**

      $B \longleftarrow B + w_j$

if $A + S \leq W/3$ *and* $B + E \leq 2W/3$ **then**

  return x_i

if $A + S > W/3$ **then**

  $E \longleftarrow B + E + w_i$

  $X' \longleftarrow x_k \in X : x_k < x_i$

  return LowerWeighted3Median(X',Y,S,E)

if $B + E > 2W/3$ **then**

  $S \longleftarrow A + S + w_i$

  $X' \longleftarrow x_k \in X : x_k > x_i$

  return LowerWeighted3Median(X',Y,S,E)

Lo pseudocodice della *upper weighted 3-median* è speculare a 2, ma tiene in considerazione la definizione [Eq. 3 4]. L'equazione di ricorrenza associata a questo algoritmo è la seguente $T(n) = T(n/2) + O(n)$ che ha soluzione nel caso peggiore $O(n)$. La chiamata iniziale per questo algoritmo sarà LowerWeighted3Median$(X, Y, 0, 0)$.

1.3 Partitioning

Dopo aver determinato le due weighted 3-median, x e y, dobbiamo analizzare due casi:

- le mediane corrispondono $x = y$
- le mediane non corrispondono $x \neq y$

a seconda del caso in cui ci si trova sarà necessario effettuare delle partizioni degli elementi per ogni processore in modo differente, questo allo scopo di scartare almeno 1/3 degli elementi per ogni iterazione.

1.3.1 Five Way Partition

Se le due mediane differiscono, quindi $x \neq y$, necessitiamo di effettuare una *Five Way Partition*. Nell'algoritmo GP si partizionano gli elementi attorno le due weighted 3-median x e y. La partizione può essere svolta da ogni processore in tempo lineare $O(n/p)$. Ogni processore determina lx_i, ex_i, bxy_i, ey_i e gy_i. Queste rappresentano la grandezza delle cinque partizioni locali e sono rispettivamente gli elementi minori di x, uguali a x, compresi tra x e y, uguali a y e più grandi di y. Il successivo step è una comunicazione ALL-REDUCE dove ogni processore determina LX, EX, BXY, EY, GY e sono rispettivamente gli elementi minori di x, uguali a x, compresi tra x e y, uguali a y e più grandi di y considerando tutti i processori. La comunicazione ha complessità $O(log(p))$. A questo punto vengono effettuati in cascata una serie di confronti per determinare le nuove partizioni o, nel caso migliore, per restituire l'i-esima statistica d'ordine. I confronti vengono fatti in $O(1)$. Quindi la complessità dell'algoritmo è $O((n/p) + log(p))$ [Alg. 3]. Nella reale implementazione dell'algoritmo la funzione prenderà a parametro i valori LX, EX, BXY, EY, GY precedentemente ottenuti tramite una comunicazione ALL-REDUCE.

1.3.2 Three Way Partition

Se le due mediane sono uguali, quindi $x = y$, necessitiamo di effettuare una *Three Way Partition*. Consideriamo il caso GP. Ogni processore inizia partizionando gli elementi attorno la weighted 3-median x, questo necessita tempo $O(n/p)$. Ogni processore calcola il numero di elementi che sono $< x$ o $> x$, che indico con l e g. A questo punto si effettua una comunicazione ALL-REDUCE per trasmettere $L \longleftarrow \sum_i l_i$ e $G \longleftarrow \sum_i g_i$, che rappresentano rispettivamente il numero di elementi $< x$ o $> x$ considerando tutti i processori. I successi passi effettuano dei confronti per determinare le nuove partizioni o, nel caso migliore, per restituire l'i-esima statistica d'ordine. La complessità dell'algoritmo è quindi $O((n/p)+log(p))$ [Alg. 4]. Nella reale implementazione dell'algoritmo la funzione prenderà a parametro i valori L, G precedentemente ottenuti tramite una comunicazione ALL-REDUCE.

2 Parallel Selection

Dopo aver discusso i principali sotto-algoritmi necessari per l'esecuzione dell'applicazione, possiamo introduttore l'algoritmo parallelo. All'inizio si assegnano ad ogni processore un quantitativo di elementi pari a n/p. Nel caso in cui n non sia divisibile per p allora il processo 0 manterrà i valori in eccesso. Nel ciclo while non necessitiamo di bilanciare il carico di lavoro ad ogni iterazioni, questo perchè l'utilizzo della weighted 3-median garantisce di scartare almeno 1/3 degli elementi. Questo potrebbe causare uno sbilanciamento del carico che però viene compensato attraverso un notevole risparmio sul tempo di comunicazione. A causa di quanto appena detto, possiamo determinare il numero di iterazioni svolte dall'algoritmo.

Il numero di iterazioni è determinato da questa relazione: dato il numero di elementi che l'algoritmo deve considerare N, che inizialmente è n — la grandezza del problema —, si ha che quando N diventerà $n/cp = O(n/p)$ che è al massimo $log_{3/2}(cp) = O(log(p))$ allora il ciclo while termina. Il parametro c è un valore intero che si determina sperimentalmente. Durante ogni iterazione il singolo processore determina il numero di elementi locali n_i e successivamente determina la mediana locale m_1 in tempo $O(n_i)$ grazie alla procedura di Selection [Alg. 1]. Successivamente si effettua una comunicazione ALL-GATHER dove ogni processore verrà a conoscenza di tutti gli m_i e n_i, questa ha complessità $O(log(p))$ grazie all'utilizzo di un albero binomiale. Ogni processore in modo indipendente calcolerà le due weighted 3-median x e y [Alg. 2] in tempo $O(p)$. Se $x \neq y$ si applica la procedura di Five Way Partition [Alg. 3] altrimenti si utilizzerà la Three Way Partition [Alg. 4]. Dopo aver effettuato una di queste due procedure si controlla se il problema è stato risolto. Se è stata determinata l'i-esima statistica d'ordine si restituisce il risultato e si termina la computazione, altrimenti si verifica la condizione $N \leq n/(cp)$, se questa non è verificata allora si itera nuovamente riducendo la dimensione del problema, altrimenti si necessita una risoluzione sequenziale

Algorithm 3: Five Way Partition

Data: A, an array; i, the statistics to be determined; x, a pivot; y, another pivot; *sorted*, boolean value indicating whether or not A has been sorted

Result: the value of the i-th statistics of A or ∞, a sentinel value indicating that additional work must be performed, i the new index of the statistics searched for, ni the new local number of elements and N, the new total number of elements; note that when the i-th statistics of A is found, the values of i, ni, and N are not relevant and we return ∞ as their value

if $sorted \neq true$ **then**
- partition the elements around x and y;
- /* compute the number of elements respectively $< x$, $= x$, $> x$ and $< y$, $= y$, $> y$ */
- compute lxi, exi, $bxyi$, eyi and gyi;
- /* compute the total number of elements $< x$, $= x$, $> x$ and $< y$, $= y$, $> y$ */
- /* An ALL-REDUCE exchange */
- compute LX, EX, BXY, EY, GY;

if $i \leq LX$ **then**
- discard all of the elements $\geq x$;
- $ni \longleftarrow lxi$;
- $N \longleftarrow LX$;
- return ∞, i, ni, N;

else if $LX < i \leq LX + EX$ **then**
- return x, ∞, ∞, ∞;

else if $LX + EX < i \leq LX + EX + BXY$ **then**
- discard all of the elements $\leq x$ and $\geq y$;
- $i \longleftarrow i - (LX + EX)$;
- $ni \longleftarrow bxyi$;
- $N \longleftarrow BXY$;
- return ∞, i, ni, N;

else if $LX + EX + BXY < i \leq LX + EX + BXY + EY$ **then**
- return y, ∞, ∞, ∞;

else
- discard all of the elements $\leq y$;
- $i \longleftarrow i - (LX + EX + BXY + EY)$;
- $ni \longleftarrow gyi$;
- $N \longleftarrow GY$;
- return ∞, i, ni, N;

Algorithm 4: Three Way Partition

Data: A, an array; i, the statistics to be determined; x, a pivot; *sorted*, boolean value indicating whether or not A has been sorted

Result: the value of the i-th statistics of A or ∞, a sentinel value indicating that additional work must be performed, i the new index of the statistics searched for, ni the new local number of elements and N, the new total number of elements; note that when the i-th statistics of A is found, the values of i, ni, and N are not relevant and we return ∞ as their value

if $sorted \neq true$ **then**
- partition the elements around x;
- compute li, the number of elements $< x$;
- compute gi, the number of elements $> x$;
- /* An ALL-REDUCE exchange */
- compute $L \longleftarrow \sum li$ and $G \longleftarrow \sum gi$;

if $i \leq L$ **then**
- discard all of the elements $\geq x$;
- $ni \longleftarrow li$;
- $N \longleftarrow L$;
- return ∞, i, ni, N;

else if $L < i \leq N - G$ **then**
- return x, ∞, ∞, ∞;

else
- discard all of the elements $\leq x$;
- $i \longleftarrow i - (N - G)$;
- $ni \longleftarrow gi$;
- $N \longleftarrow G$;
- return ∞, i, ni, N;

del problema rimanente. Quando il problema è sufficientemente piccolo attraverso una comunicazione GATHER si inviano al processo 0 tutti i valori restanti. Questo si occuperà di identificare l'i-esima statistica d'ordine attraverso una chiamata alla funzione Selection [Alg. 1]. L'analisi nel caso peggiore dell'algoritmo prevede una complessità dovuta alla comunicazione pari a $O((log(p))^2)$, questo poichè svolgere una ALL-GATHER o una ALL-REDUCE (effettuate nella fase di partitioning) costa $O(log(p))$, però noi svolgiamo $O(log(p))$ iterazioni e quindi $O((log(p))^2)$. La computazione costa $O(n/p + p)$ ad ogni iterazione. Se l'algoritmo necessita una risoluzione sequenziale allora necessito di un ulteriore step che ci impiega $O(n/p)$, quindi $O(log(p)(n/p + p) + n/p) = O(log(p)(n/p + p))$. In conclusione tenendo conto del tempo di calcolo e del tempo di comunicazione l'algoritmo impiega $O(log(p)(log(p) + p + n/p))$ [Alg. 5].

Algorithm 5: ParallelSelection

Data: A, an array; i, the statistics to be determined; *sort*, boolean value indicating whether or not A must be sorted using RadixSort; c, a constant influencing the number of iterations, to be determined experimentally

Result: the i-th statistics of A

/* The n elements of the input array A are distributed to the p processors so that each one is responsible for either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ elements; let $left$ and $right$ be respectively the indices of the first and last element of the sub-array handled by the process with rank id */

$left \longleftarrow \lfloor id \cdot n/p \rfloor$;

$right \longleftarrow \lfloor (id + 1) \cdot n/p \rfloor - 1$;

if *sort* **then**

  RadixSort$(A, left, right)$;

while $N > n/(c \cdot p)$ **do**

  $ni \longleftarrow right - left + 1$;

  $mi \longleftarrow Select(i, left, right)$;

  /* An ALL-GATHER exchange */

  Send to the other processes mi and ni;

  Compute x and y, the two weighted 3-medians of the medians;

  if $x \neq y$ **then**

    $m, i, ni, N \longleftarrow FiveWayPartition(A, i, x, y, sort)$;

  else

    $m, i, ni, N \longleftarrow ThreeWayPartition(A, i, x, sort)$;

  if $m \neq \infty$ **then**

    return m;

/* the problem has not been solved in parallel */

/* A GATHER exchange */

send to the processor with rank 0 the remaining ni elements;

if $id = 0$ **then**

  determine new $left$ and $right$ indices;

  return Select$(i, left, right)$;

2.1 Variante con Radix Sort

In questa variante ogni processore ordinerà i propri elementi utilizzando l'algoritmo RS. Questo può ordinare in tempo lineare, quindi la complessità è $O(n/p)$ per ogni processore. In questo caso il calcolo della mediana m_i è immediato poichè non necessitiamo più di eseguire 1, ma possiamo direttamente accedere all'elemento $\lfloor m_i \rfloor$ in $O(1)$. L'analisi per le procedure di partitioning non cambia. Possiamo analizzare la complessità: il numero di iterazioni risulta lo stesso dell'algoritmo GP quindi $O(log(p))$, ogni processo ordina i suoi elementi in $O(n/p)$ con RS, determinare m_i costa $O(1)$, il calcolo della weighted 3-median costa $O(p)$ e le procedure di partitioning costano n/p. Se il problema deve essere risolto in sequenziale l'ultima Selection costa $O(n/p)$. Quindi il costo totale è dato da $O(log(p)(log(p) + p + n/p) + n/p)) = O(log(p)(log(p) + p + n/p))$. Dunque l'algoritmo GP e la variante che utilizza RS presentano la stessa complessità.

2.1.1 Analisi Radix Sort

Radix Sort utilizza come sotto routine Counting Sort (CS). La complessità di RS è $O(d(n+k))$ dove:

- d è il numero di passate che deve effettuare
- n è la dimensione del problema
- k è il valore più grande tra gli elementi del problema

in generale d è una costante e quindi risulta trascurabile, portando a una complessità di $O(n+k)$, ovvero la complessità di CS. Mettiamoci nel caso in cui $n = k$, facendo diventare la complessità di CS $O(n)$. RS impiega nel caso peggiore $O(d(n))$. Posso ricavare $d = log_b(M) + 1$ dove b è la base in cui stiamo lavorando e M è il valore massimo che vogliamo rappresentare. Per esempio se io volessi rappresentare in base 10 una serie di numeri che vanno da 0 a 1000 allora $d = log(1000) + 1 = 4$, quindi necessiterei di effettuare 4 passate con CS. Ovviamente è facile vedere che se $n \sim M$ allora la complessità diverrebbe $O(n(log(M))) = O(n(log(n)))$ che non è migliore rispetto all'utilizzo di Merge Sort. Ma possiamo fare un ragionamento più generico per ricavare un upper bound per M. Scelgo una base $t = n$, per cui vale che $d = log_n(M) + 1$, la complessità sarà dunque $O(n(log(M)))$ dato che la base risulta irrilevante asintoticamente. Quindi per ottenere un comportamento lineare in RS deve valere $M \leq n^c$ con c una costante arbitraria. Per esempio se $M = n^c$ allora la complessità è $O(n(log(n^c))) = O(nc) = O(n)$. Se $M = 2^n$ è facile vedere che il comportamento di RS è $O(n^2)$. Nell'implementazione reale si è utilizzato RS con l'utilizzo della base n e con numeri compresi tra $[0, n-1]$, mantenendo la relazione $M \leq n^c$ in modo da garantire comportamento lineare.

È possibile effettuare una trasformazione dell'input che permette di eseguire RS in tempo $O(n)$ sempre. Immaginiamo di lavorare con n *computer words* costituite da r bits ciascuna per un totale di b bits, la cui base è 2^r. Il numero di passate che RS può effettuare su queste words è $d = \lceil b/r \rceil$. Possiamo ottenere comportamento lineare scegliendo $r = log_2(n)$, poichè la complessità di RS si può esprire come $O(d(n+k)) = O(d(n+2^r)) = O(\frac{b}{r}(n+2^r)) = O(\frac{b}{lg(n)}(n+2^{lg(n)})) = O(\frac{b}{lg(n)}(n))$. Vale che $d = \frac{b}{r} = \frac{b}{log_2(n)}$ da cui $d(log_2(n)) = b$ ottenendo una complessità $O(\frac{b}{r}(n+2^r)) = O(\frac{d(lg(n))}{lg(n)}(n)) = O(dn) = O(n)$. Inoltre scegliendo $r = log_2(n)$ posso rappresentare numeri nel range $n^d - 1$. Per esempio per ordinare $n = 2^{16}$ numeri da $b = 32$ bits, scegliendo $r = log_2(2^{16}) = 16$ mi bastano con RS $d = b/r = 32/16 = 2$ passate.

3 Benchmark

In questa sezione si discutono i risultati ottenuti sperimentalmente. I test sono stati eseguiti su una MacBook M1 Pro equipaggiato con un processore M1 Pro da 10 Core (di cui solo 8 accessibili all'utente) e 16 GB di RAM.

Sono stati condotti dei test su diverse taglie dell'input n, in particolare la dimensione dell'input parte da un minimo di $n = 10^4$ elementi fino ad un massimo di $n = 10^9$ numeri interi. Per ogni n è stato fatto girare l'algoritmo con un numero di processori che varia da $p = 1, .., 8$. Siano T_1 il tempo necessario per eseguire l'algoritmo su un singolo processore e T_p il tempo necessario per eseguire l'algoritmo su p processori, le metriche riportate nei grafici seguenti sono:

- Speedup: $\psi(n,p) = T_1/T_p$
- Efficienza: $E = T_1/pT_p$

inoltre è presente un grafico che relaziona il tempo di esecuzione rispetto al numero dei processori. Il parametro c sperimentale è stato fissato a $c = 2$ per tutti gli esperimenti. Nel grafico [3.1] sono riportate le differenti metriche nel caso GP, quindi quando non si utilizza l'algoritmo di ordinamento RS. Il secondo grafico [3.2] rappresenta le performance relative all'esecuzione dell'algoritmo con l'utilizzo di RS. In ultimo [3.3] è presente una rappresentazione che unisce le due casistiche.
Si può notare immediatamente che nonostante l'algoritmo GP e la sua variante che sfrutta RS presentino la stessa complessità computazione, il comportamento sperimentale è in netto vantaggio per la variante con ordinamento. Inoltre è possibile osservare un fenomeno di saturazione: all'aumentare della dimensione del problema n lo speedup e il tempo di calcolo tendono a stabilizzarsi. Tale comportamento non è visibile invece quando n è piccolo. Questo andamento è un'evidenza tipica del regime di Gustafson, infatti, si nota come la dimensione del problema n sia funzione crescente di p in modo da tenere stabile T_p. Durante

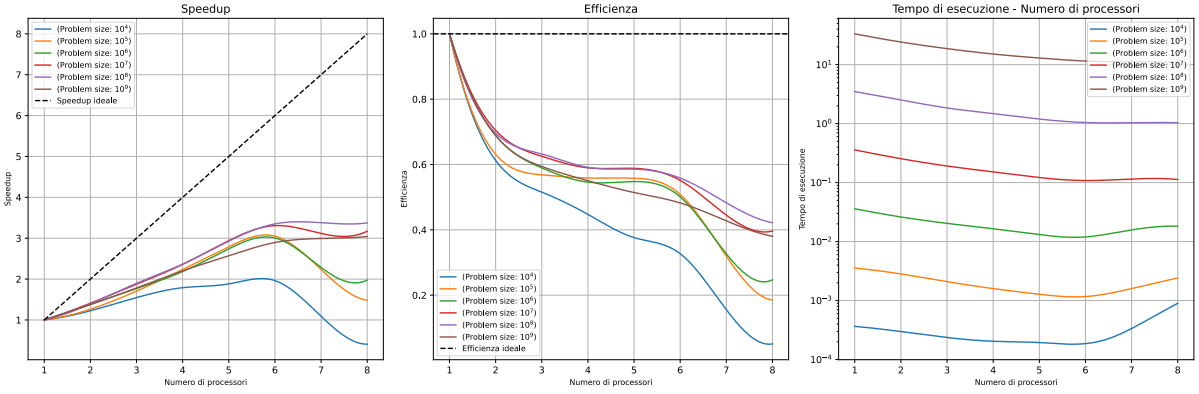

Figura 3.1: Curve di speedup, efficienza e tempo/processori relative all'esecuzione dell'algoritmo GP su differenti taglie dell'input n.

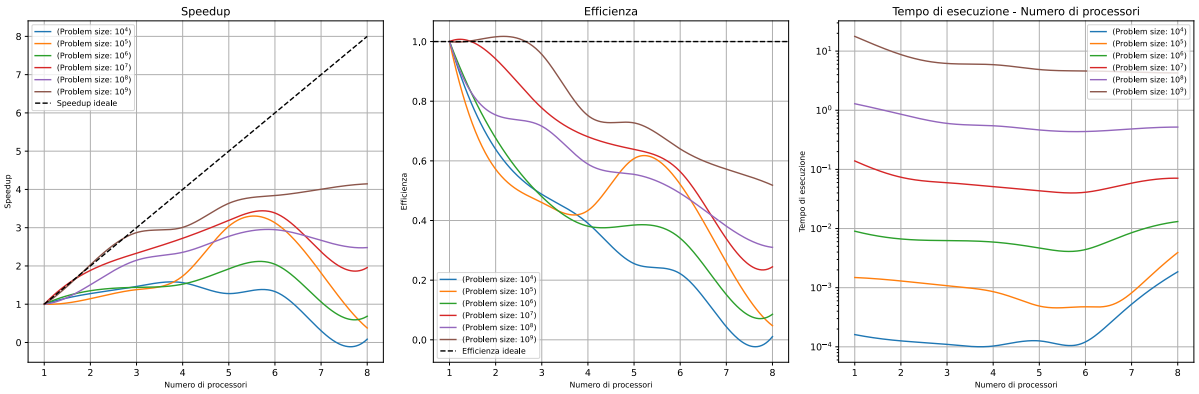

Figura 3.2: Curve di speedup, efficienza e tempo/processori relative all'esecuzione dell'algoritmo con Radix Sort su differenti taglie dell'input n.

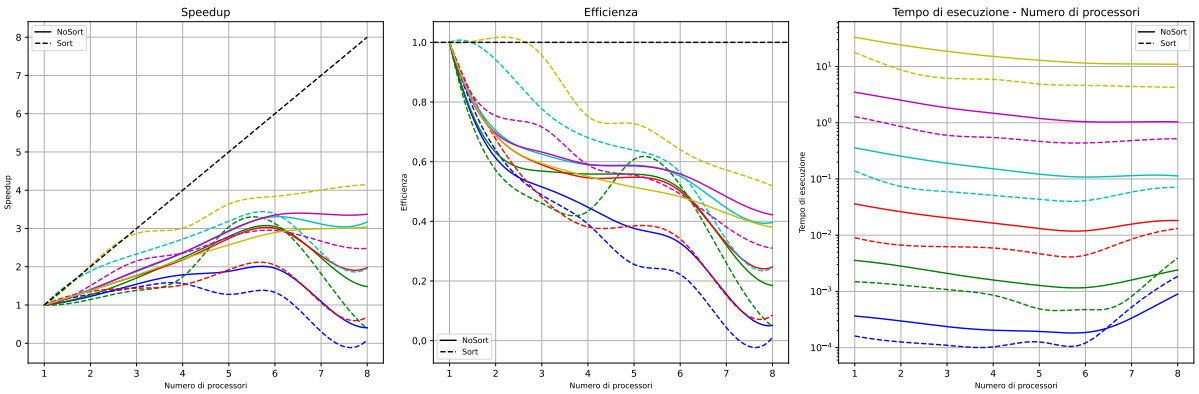

Figura 3.3: Curve di speedup, efficienza e tempo/processori relative all'esecuzione dell'algoritmo GP e la sua variante con Radix Sort su differenti taglie dell'input n.

il test con il maggior numero di elementi, ovvero $n = 10^9$, si può osservare come l'efficienza rimanga sopra il 70% fino a $p = 5$. Molto probabilmente aumentando la dimensione del problema l'efficienza dovrebbe essere $\geq 70\%$ anche per $p \geq 5$. Questa assunzione è lecita osservando l'andamento dell'efficienza nelle varie dimensioni del problema.

Il tempo di esecuzione non tiene conto della lettura del file di l'input, poichè questo avviene in sequenziale. Inoltre si ignora il tempo impiegato — in fase di inizializzazione — per la distribuzione dell'input ai differenti processi da parte del processo 0 tramite una comunicazione SCATTER.

Tutti i risultati numerici riportati nei grafici sono visibili in tabella 3.1.

Le immagini sono state inserite in formato vettoriale in modo da non perdere di risoluzione in caso di zoom, permettendo un migliore confronto tra le curve.

4 Conclusioni

In questo elaborato si è illustrato come poter eseguire un algoritmo parallelo in grado di determinare l'i-esima statistica d'ordine in tempo $O(log(p)(log(p) + p + n/p))$. Si è visto come applicare le tecniche basate sulla *weighted 3-median* permetta a ogni iterazione di eliminare almeno $1/3$ di N garantendo un numero di esecuzioni pari $O(log(p))$. Inoltre, si è implementata anche una variante che sfrutta l'algoritmo Radix Sort. Formalmente si è dimostrato che la complessità dell'algoritmo che sfrutta l'ordinamento è la stessa di quello GP, ma sperimentalmente si ottengono risultati nettamente migliori utilizzando come sotto routine RS. In ultimo è stato osservato come l'algoritmo agisca in regime di Gustafson, infatti, all'aumentare di n e di p — rispettivamente: dimensione del problema e numero di processori — il tempo parallelo T_p tende a stabilizzarsi, tenendo l'efficienza $\geq 70\%$ per un numero di processori p sempre maggiore.

A Esecuzione dell'algoritmo

Il file contenente il codice dell'algoritmo è *main.c*. Per il corretto funzionamento dell'applicazione assicurarsi che:

- La macro BENCHMARK sia impostata al valore 0
- La macro CORRECTNESS sia impostata al valore 0
- La macro C sia impostata ad un valore ≥ 2

dopo aver effettuato i controlli sarà possibile compilare i file necessari all'esecuzione tramite il comando `make`. Il comando da eseguire per il corretto funzionamento dell'algoritmo è il seguente:

```
mpirun -np <p> ./main <NF> <ith> <S>
```

dove:

- p è il numero di processori
- NF è il nome del file contenente numeri interi
- ith è l'iesima statistica d'ordine desiderata
- S vale 0 se si vuole l'algoritmo GP altrimenti vale 1 se si desidera utilizzare la variante con Radix Sort.

Un esempio di esecuzione per determinare la 10^{a} statistica d'ordine nel file *array.txt* utilizzando l'algoritmo GP su 6 processori è questa:

```
mpirun -np 6 ./main "array.txt" 10 0
```

Per come è stato implementato Radix Sort, questo può agire anche nel caso GP e non necessariamente per valori che presentano tutti lo stesso numero di cifre. I valori che dispongono di un numero di cifre inferiore rispetto al numero di cifre massimo presenteranno degli zeri davanti. Per esempio se il numero massimo di cifre è 3 allora il numero 12 sarà rappresentato come 012.

B Benchmark

Prima di vedere come replicare i benchmark vediamo alcuni file ausiliari che sono utili ai fini dell'esecuzione dell'algoritmo. Il codice *generator.py* serve per produrre un file dal nome *array.txt* che conterrà dei numeri casuali in un certo range di valori. Per eseguire questo codice dovremmo lanciare il comando:

```
python generator.py <maxValue>
```

per esempio:

```
python generator.py 1000
```

il parametro *maxValue* rappresenta il numero di valori che si vuole generare. Il range di valori sarà compreso tra 1 e $maxValue - 1$. Quindi nell'esempio stiamo generando 1000 valori compresi tra 1 e 999.

Si possono effettuare due tipi di banchmark, uno relativo alle performance e uno di correttezza.

B.1 Performance

Il benchmark seguente può essere svolto utilizzando lo script *benchmark.py*. Questo codice effettua una serie di esecuzioni dell'algoritmo parallelo su un input di taglia n — si fa riferimento ai dati di input presenti nel file *array.txt* — in tutte le combinazioni possibili, dunque valutando p da uno ad otto processori nella configurazione di default nel caso GP e successivamente con l'utilizzo Radix Sort. Per garantire il corretto funzionamento dello script python è necessario che all'interno del file *main.c* le macro siano impostate in questo modo:

- La macro BENCHMARK sia impostata al valore 1
- La macro CORRECTNESS sia impostata al valore 0

- La macro C sia impostata ad un valore ≥ 2

Dopo averle configurate correttamente ed aver eseguito una compilazione tramite comando `make` possiamo lanciare il seguente comando per avviare il benchmark:

```
python benchmark.py
```

Alla fine dell'esecuzione verrà creato un file dal nome *resoconto-benchmark.txt* che conterrà i tempi medi di esecuzione per le differenti configurazioni testate.

B.2 Correttezza

Per verificare i risultati dell'algoritmo parallelo sono stati effettuati più benchmark di correttezza confrontando i risultati forniti dalla Selection sequenziale con quelli ricavati della Parallel Selection. La procedura viene svolta tramite uno script chiamato *checkResult.py* che si occuperà di trovare un certo numero di statistiche d'ordine sia sulla Parallel Selection che sulla Selection per poi effettuarne il confronto. Lo script agisce sul file di input *array.txt* che conterrà il problema. Per garantire il corretto funzionamento dello script python è necessario che all'interno del file *main.c* le macro siano impostate in questo modo:

- La macro BENCHMARK sia impostata al valore 0
- La macro CORRECTNESS sia impostata al valore 1
- La macro C sia impostata ad un valore ≥ 2

Inoltre anche la macro CORRECTNESS del file *select.c* deve essere impostata a 1. Dopo aver configurato tutto correttamente e ed aver eseguito una compilazione tramite comando `make` possiamo lanciare il seguente comando per avviare il benchmark:

```
python checkResult.py
```

In questo caso lo script non produrrà nessun file di output ma scriverà sul terminale i risultati ottenuti ogni volta che verrà effettuato un controllo di correttezza nelle differenti configurazioni. Se lo script termina allora tutti i controlli di correttezza saranno stati eseguiti con successo, altrimenti trovata la prima statistica d'ordine non coincidente con quella corretta lo script si fermerà restituendo un messaggio d'errore.

5 Riferimenti Bibliografici

[1] J. Lobo e S. Kuwelkar, ≪Performance Analysis of Merge Sort Algorithms,≫ pp. 110–115, 2020. DOI: 10.1109/ICESC48915.2020.9155623.

[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest e R. E. Tarjan, ≪Time bounds for selection,≫ *Journal of Computer and System Sciences*, vol. 7, n. 4, pp. 448–461, 1973, ISSN: 0022-0000. DOI: https://doi.org/10.1016/S0022-0000(73)80033-9. indirizzo: https://www.sciencedirect.com/science/article/pii/S0022000073800339.

[3] M. Cafaro, V. Bene e G. Aloisio, ≪Deterministic parallel selection algorithms on coarse-grained multicomputers,≫ *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 2336–2354, dic. 2009. DOI: 10.1002/cpe.1453.

n	p	GP			Radix Sort		
		Tempo	Speedup	Efficienza	Tempo	Speedup	Efficienza
10^4	1	0.000365	1.00	1.00	0.000161	1.00	1.00
	2	0.000298	1.23	0.62	0.000126	1.28	0.64
	3	0.000236	1.55	0.52	0.000110	1.46	0.49
	4	0.000204	1.79	0.45	0.000103	1.56	0.39
	5	0.000194	1.88	0.38	0.000126	1.28	0.26
	6	0.000186	1.96	0.33	0.000121	1.33	0.22
	7	0.000335	1.09	0.16	0.000525	0.31	0.04
	8	0.000898	0.41	0.05	0.001856	0.09	0.01
10^5	1	0.003564	1.00	1.00	0.001486	1.00	1.00
	2	0.002823	1.26	0.63	0.001299	1.14	0.57
	3	0.002090	1.71	0.57	0.001076	1.38	0.46
	4	0.001595	2.23	0.56	0.000857	1.73	0.43
	5	0.001277	2.79	0.56	0.000489	3.04	0.61
	6	0.001168	3.05	0.51	0.000475	3.13	0.52
	7	0.001595	2.23	0.32	0.000817	1.82	0.26
	8	0.002406	1.48	0.18	0.003928	0.38	0.05
10^6	1	0.035986	1.00	1.00	0.009013	1.00	1.00
	2	0.026112	1.38	0.69	0.006663	1.35	0.68
	3	0.020344	1.77	0.59	0.006258	1.44	0.48
	4	0.016481	2.18	0.55	0.005914	1.52	0.38
	5	0.013136	2.74	0.55	0.004684	1.92	0.38
	6	0.011972	3.01	0.50	0.004410	2.04	0.34
	7	0.015744	2.29	0.33	0.008495	1.06	0.15
	8	0.018214	1.98	0.25	0.013135	0.69	0.09
10^7	1	0.358161	1.00	1.00	0.138889	1.00	1.00
	2	0.254193	1.41	0.70	0.073733	1.88	0.94
	3	0.190988	1.88	0.63	0.059555	2.33	0.78
	4	0.151802	2.36	0.59	0.051007	2.72	0.68
	5	0.121768	2.94	0.59	0.043509	3.19	0.64
	6	0.108266	3.31	0.55	0.041047	3.38	0.56
	7	0.114953	3.12	0.45	0.058564	2.37	0.34
	8	0.112982	3.17	0.40	0.070927	1.96	0.25
10^8	1	3.497114	1.00	1.00	1.288136	1.00	1.00
	2	2.513229	1.39	0.70	0.854330	1.51	0.75
	3	1.843699	1.90	0.63	0.599473	2.15	0.72
	4	1.478477	2.37	0.59	0.546694	2.36	0.59
	5	1.193963	2.93	0.59	0.464436	2.77	0.55
	6	1.043614	3.35	0.56	0.436940	2.95	0.49
	7	1.034724	3.38	0.48	0.482981	2.67	0.38
	8	1.036359	3.37	0.42	0.519402	2.48	0.31
10^9	1	33.281954	1.00	1.00	17.746692	1.00	1.00
	2	24.164159	1.38	0.69	8.734381	2.03	1.02
	3	18.678370	1.78	0.59	6.181075	2.87	0.96
	4	15.120630	2.20	0.55	5.899794	3.01	0.75
	5	12.942816	2.57	0.51	4.880016	3.64	0.73
	6	11.499078	2.89	0.48	4.621592	3.84	0.64
	7	11.121658	2.99	0.43	4.430671	4.00	0.57
	8	10.945536	3.04	0.38	4.280570	4.15	0.52

Tabella 3.1: Dati relativi a speedup, efficienza e tempo per le varie configurazioni di processori per l'algoritmo GP e la sua variante con Radix Sort.