ALGORITMA-ALGORITMA PARALLEL RANDOM ACCESS MACHINE (PRAM)

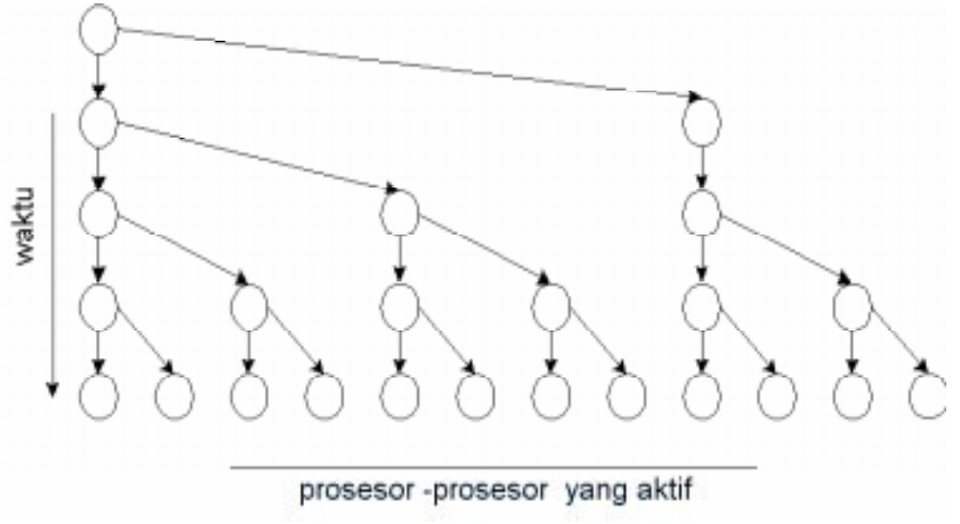
Algoritma yang dibahas :

1. Parallel reduction
2. Prefix sums
3. List ranking
4. Pre-order tree traversal`
5. Merging two sorted lists
6. Graph coloring

See. : M.J. Quinn for detailed

Algoritma-algoritma PRAM memiliki 2 (dua) fase :

1. mengaktifkan sejumlah prosesor
2. prosesor yang sudah diaktifkan (pada fase 1), melaksanakan komputasi secara paralel



Gambar 2.4 Untuk mengubah 1 prosesor yang aktif ke p prosesor dibutuhkan ⌈log p⌉ Langkah

Jumlah prosesor yang aktif merupakan lipat-2 (2^n) dari prosesor tunggal atau logaritma dari basis 2.

Instruksi meta untuk mengaktifkan prosesor yang digunakan (dalam fase 1) :

```
spawn (<nama prosesor>)
```

Instruksi meta untuk melakukan komputasi secara paralel (dalam fase 2) :

```
for all <processor list>
do
   <statement list>
endfor
```

Pohon biner menjadi paradigma yang penting dalam komputasi paralel. Pada beberapa algoritma ada yang menggunakan aliran data top-down (akar –daun). Contoh :

- **broadcast**
  akar mengalirkan (mengirimkan) data yang sama ke setiap daunnya
- **divide-and-conquer**
  pohon menggambarkan adanya perulangan sub divisi suatu masalah ke sub masalah yang lebih kecil.

Algoritma lain yang mengalirkan data secara bottom-up (daun -akar) adalah **operasi reduksi atau “fan-in”.**

DEFINISI

Diberikan sekumpulan n nilai a_1, a_2, … ,a_n dan operator biner asosiatif $\oplus$, **"reduksi"** adalah proses menghitung dari :

$$a_1 \oplus a_2 \oplus \ldots \oplus a_n$$

Salah satu contoh dari operasi reduksi adalah penjumlahan paralel (parallel summation).

Parallel Reduction (Reduksi paralel)

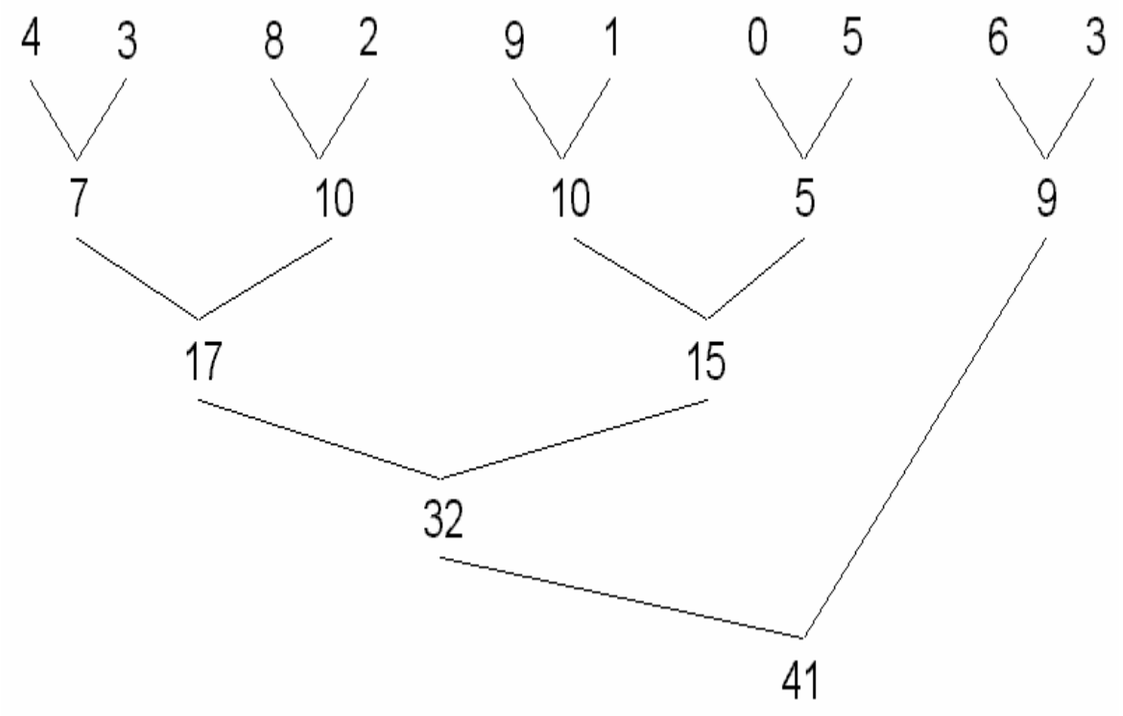
Prosesor PRAM memanipulasi data yang disimpan dalam register global.

DEFINISI

Penjumlahan secara paralel merupakan salah satu contoh dari operasi reduksi.

CONTOH

Reduksi secara paralel dapat digambarkan dengan pohon biner. Sekelompok n nilai ditambahkan dalam $\lceil \log p \rceil$ langkah penambahan paralel.



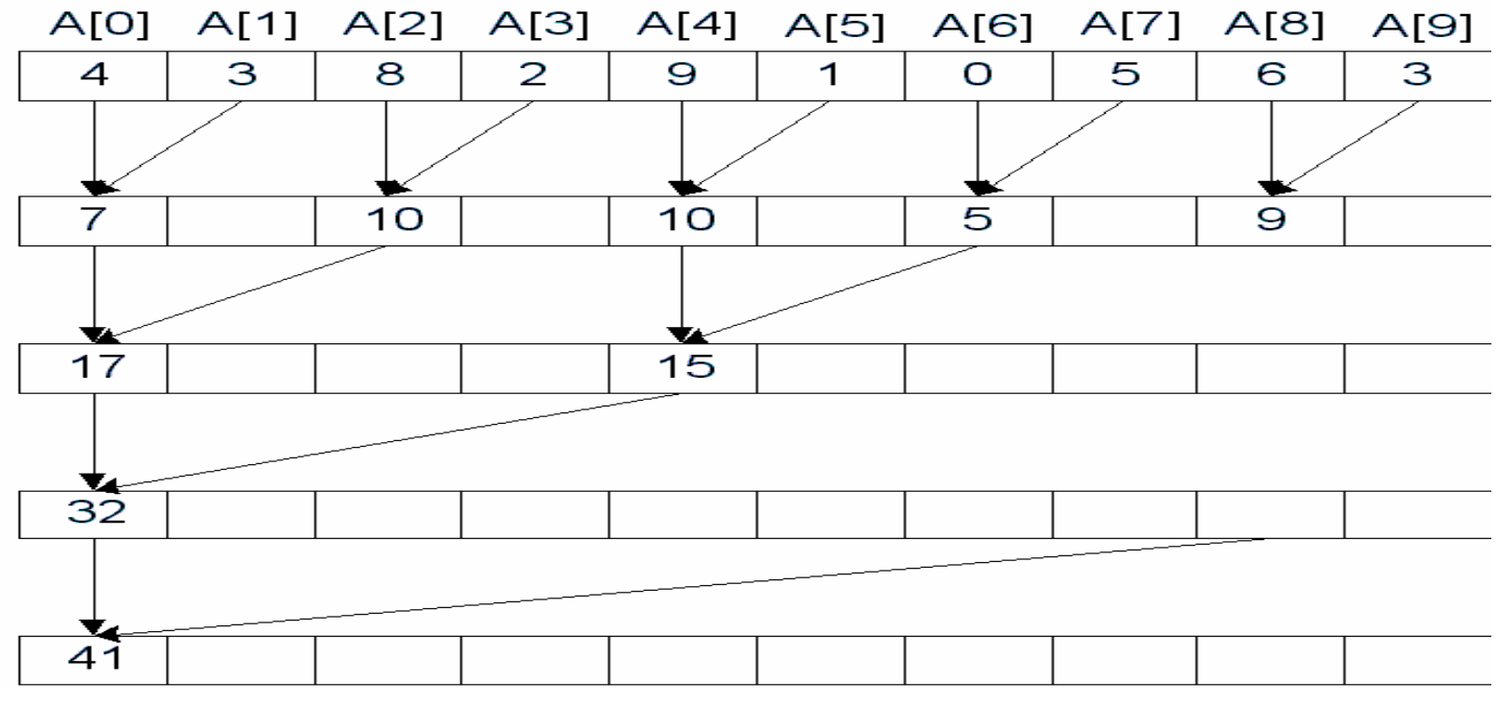
Gambar 2.5 *Implementasi algoritma penjumlahan, setiap node dari pohon merupakan elemen dalam array*

PSEUDOCODE

```
SUM(EREW PRAM)
Initial condition : List of n ≥ 1 elements stored in
A[0 … (n - 1)]
Final condition   : Sum of elements stored in A[0]
Global variables : n, A[0 … (n -1)], j
begin
  spawn (P0, P1, P2, … , P⌊ n/2 ⌋ - 1)
  for all Pi where 0 ≤  i  ≤  ⌊n/2⌋ –1 do
     for j  ←  0 to ⌈log n⌉  - 1 do
      if i modulo 2^j = 0 and 2i + 2^j < n then
         A[2i] ←   A[2i] + A[2i + 2^j]
      endif
     endfor
  endfor
end
```

Gambar 2.7 ***Algoritma PRAM EREW untuk menjumlah n elemen dengan ⌊n/2⌋ prosesor***

GAMBARAN PSEUDOCODE



Gambar 2.6 Menjumlahkan 10 nilai

KOMPLEKSITAS

Rutin spawn : $\lceil \lfloor n/2 \rfloor \rceil$ doubling steps
Perulangan for yang sekuensial : $\lceil \log n \rceil$ kali

Waktu kompleksitas algoritma : $\Theta(\log n)$,
dengan $\lfloor n/2 \rfloor$ prosesor.

PREFIX SUMS (sering disebut parallel prefixes, scan)

DEFINISI

Diberikan sekumpulan n nilai a_1, a_2, …, a_n dan operasi asosiatif $\oplus$, prefix sum adalah menghitung :

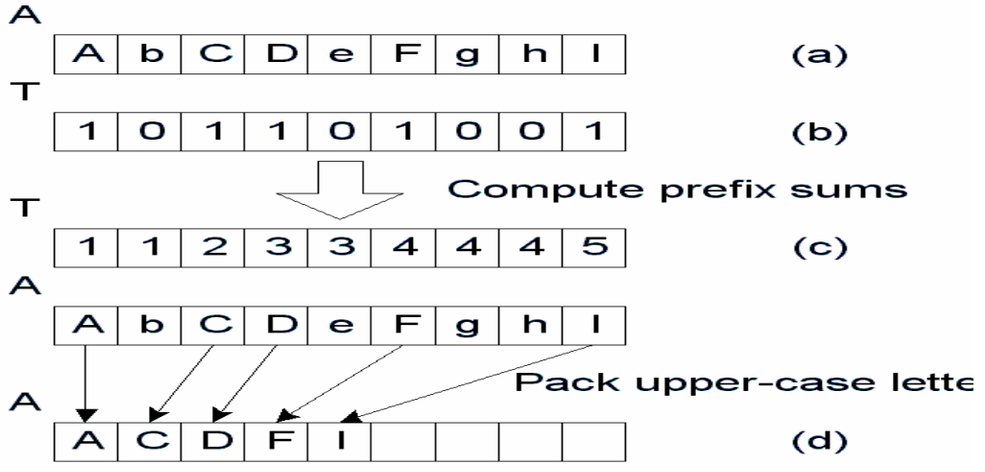
a_1
$a_1 \oplus a_2$
$a_1 \oplus a_2 \oplus a_3$
…
$a_1 \oplus a_2 \oplus a_3 \oplus \ldots \oplus a_n$

Misal : diberikan operasi + dan array integer {3, 1, 0, 4, 2}, hasil prefix sum adalah array dari {3, 4, 4, 8, 10}.

CONTOH

Diberikan array A dari n huruf. Huruf-huruf besarnya akan diurut. (lihat gambar 2.8)

a) Array A berisi huruf besar maupun huruf kecil dan ada array tambahan T berukuran n. Huruf-huruf besar akan diletakkan di awal dari A secara terurut.
b) Array T berisi 1 untuk merepresentasikan huruf besar dan 0 untuk merepresentasikan huruf kecil
c) Array T dikomputasi dengan prefix sum, menggunakan operasi tambah. Setiap huruf besar L diletakkan di A[i], nilai dari T[i] adalah indeks dari L.
d) Array A setelah "*packing*".



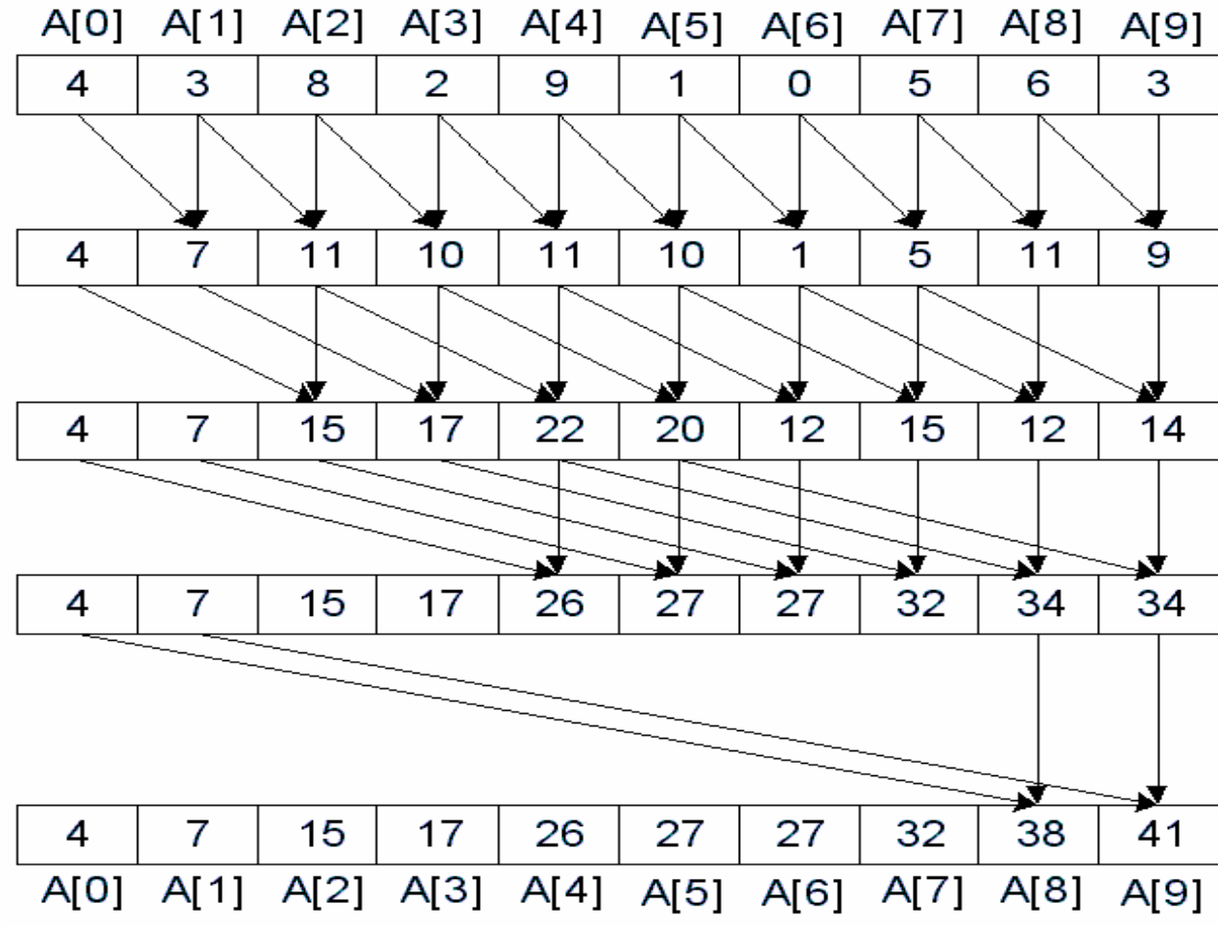
Gambar 2.8 "Packing" elemen dengan aplikasi prefix sum.

PSEUDOCODE

```
PREFIX.SUMS (CREW PRAM):
Initial condition : List of n ≥ 1 elements stored in A[0
… (n -1)]
Final condition : Each element a[i] contains A[0]  ⊕ … ⊕
A[i]
Global variables : n, A[0 … (n-1)], j
begin
 spawn (P1, P2, …, Pn - 1)
 for all Pi where 1 ≤  i  ≤ n – 1 do
       for j ← 0 to ⌈log n⌉ – 1 do
if i – 2^j ≥ 0 then
        A[i]  ← A[i] + A[i – 2^j]
    endif
    endfor
 endfor
end
```

Gambar 2.9 *Algoritma PRAM untuk menemukan prefix sum dari n elemen dengan n-1 prosesor*

GAMBARAN PSEUDOCODE



Gambar 2.10 Algoritma Prefix sum dari 10 nilai

KOMPLEKSITAS

Rutin spawn : $\lceil \log n - 1 \rceil$ instruksi

Perulangan for yang sekuensial : $\lceil \log n \rceil$ kali

Waktu kompleksitas algoritma : $\Theta(\log n)$,
dengan n - 1 prosesor.

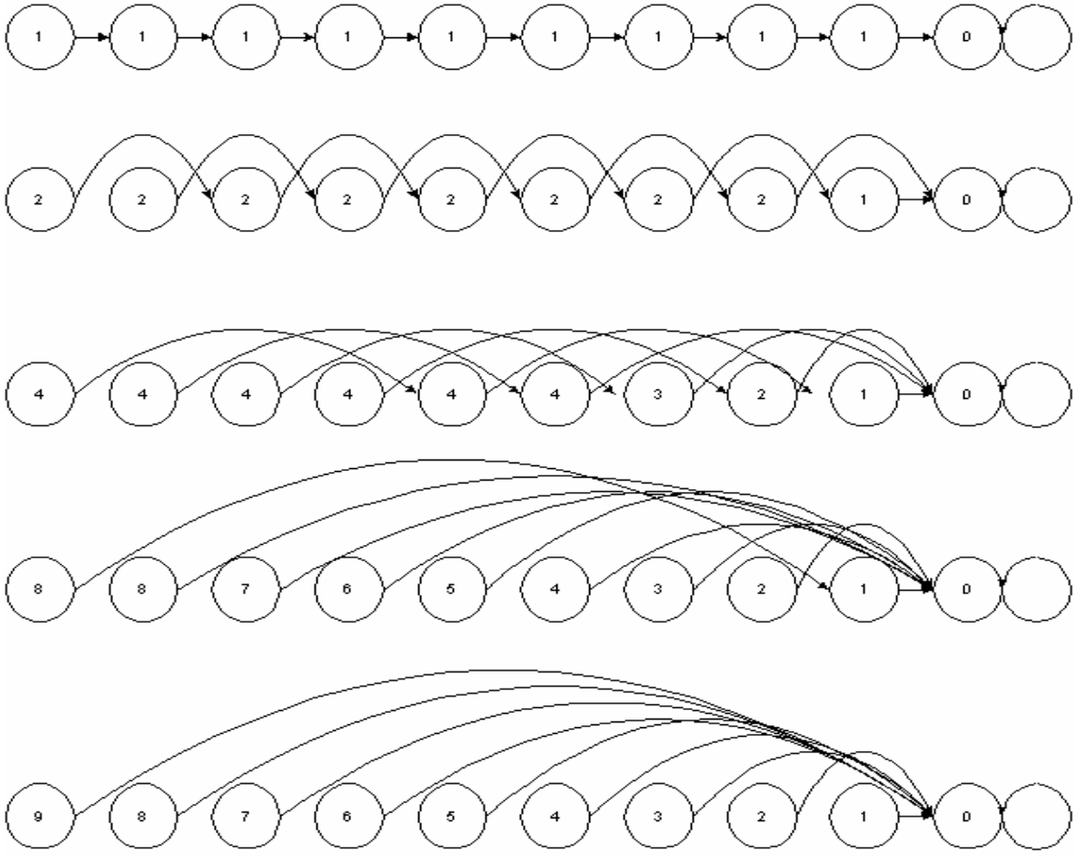
List Ranking

Suffix sum adalah "*variant*" dari prefix sum, dimana elemen array digantikan dengan linked list, dan penjumlahan dihitung dari belakang (Karp & Ramachandran 1990).

DEFINISI

Jika elemen-elemen dari list berisi 0 atau 1 dan operasi asosiatif $\oplus$ merupakan penambahan, maka masalah ini biasa disebut list ranking.

CONTOH



Gambar 2.11 *Posisi setiap item pada linked-list n elemen dicapai dalam ⌈log n⌉ langkah pointer-jumping*

PSEUDOCODE

```
LIST.RANKING (CREW PRAM):
Initial condition : Values in array next represent a linked list
Final condition : Values in array position contain original
                  distance of each element from end of list
Global variables : n, position[0 … (n - 1)], next[0 … (n - 1)], j
begin
    spawn (P0, P1, P2, …, Pn-1)
    for all Pi where 0 ≤ i ≤ n – 1 do
       if next[i] = i then position[i] ← 0
        else position[i] ←1
      endif
      for j ←1 to ⌈log n⌉ do
        position[i] ← position[i] + position[next[i]]
        next[i] ← next[next[i]]
      endfor
    endfor
end
```

Gambar 2.12 Algoritma PRAM untuk menghitung jarak dari belakang/ akhir list untuk setiap elemen singly-linked list

GAMBARAN PSEUDOCODE

Untuk menunjukkan posisi list adalah dengan menghitung jumlah penelusuran antara elemen list dan akhir list. Hanya ada (n-1) pointer antara elemen list awal dan akhir list.

Jika satu prosesor diasosiasikan dengan setiap elemen list dan pointer lompatan secara paralel, jarak dari akhir list hanya ½ bagian melalui instruksi *next*[i] ← *next*[*next*[i]]. Jika sebuah prosesor menambah hitungan ke link-traversalnya sendiri, *position*[i], hitungan link-traversal sekarang dari successornya dapat dicapai.

KOMPLEKSITAS

Rutin spawn : $\Theta(\log n)$,
Perulangan for : maksimal $\lceil \log n \rceil$ kali

Waktu kompleksitas algoritma : $\Theta(\log n)$,
dengan n prosesor.

Preorder Tree Traversal

DEFINISI

Secara sekuensial

```
PREORDER.TRAVERSAL(nodeptr):
begin
  if nodeptr null then
     nodecount  codecount + 1
     nodeptr.label  nodecount
     PREORDER.TRAVERSAL(nodeptr.left)
     PREORDER.TRAVERSAL(nodeptr.right)
  endif
end
```

Dimana paralelnya ?

Operasi dasarnya adalah pelabelan pada node. Label pada verteks sub pohon kanan tidak dapat diberikan sampai diketahui berapa banyak verteks yang ada di sub pohon kirinya, begitu sebaliknya.

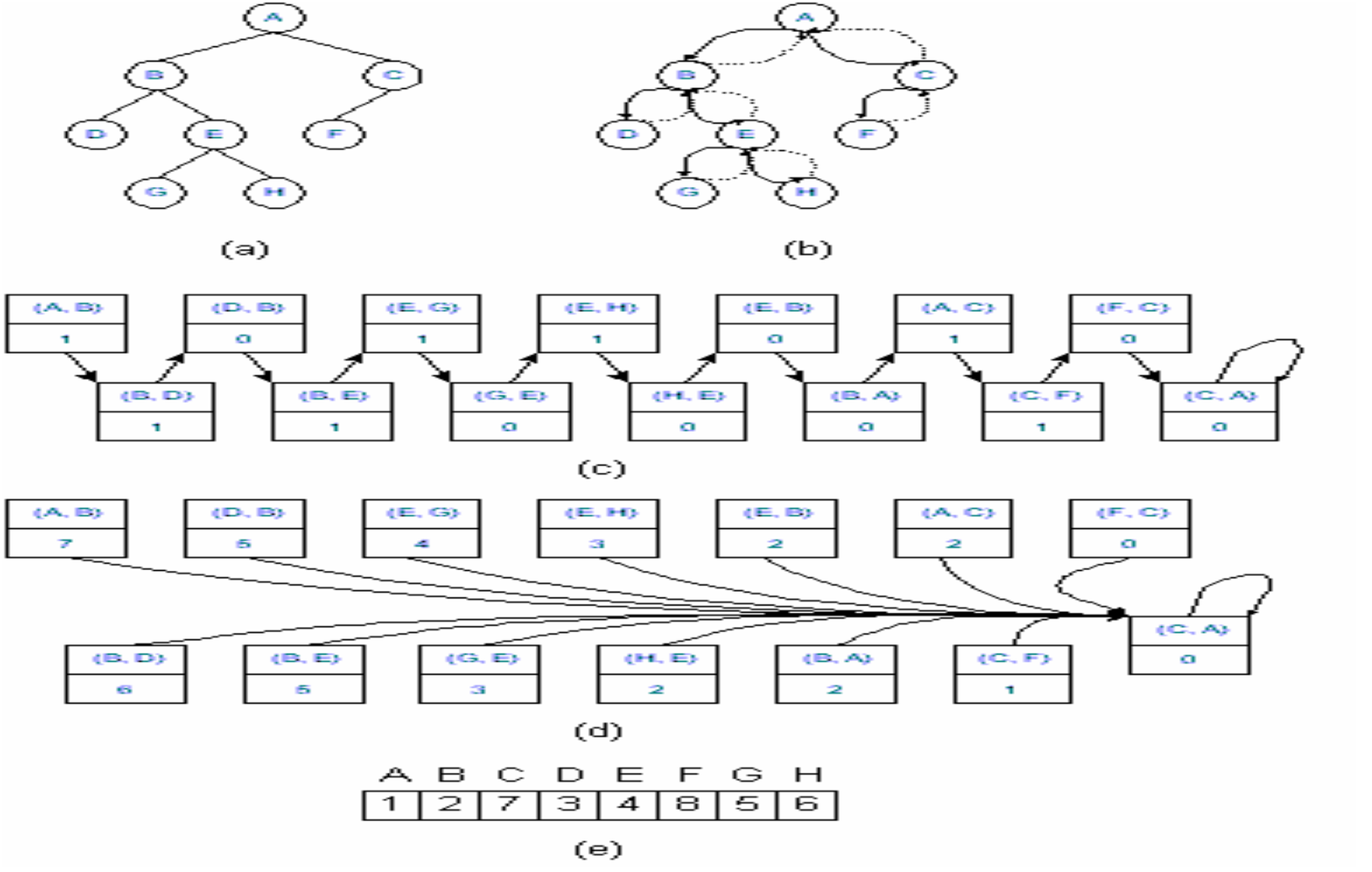
Pelaksanaan penelusuran dari depan (preorder traversal), dikerjakan secara sistematis melalui semua edge pohon. Setiap edge selalu 2 (dua) kali melewati verteks, yang turun dari parent ke child dan kebalikkannya.

Penelusuran pohon berorientasi edge ini merupakan algoritma paralel yang cepat. (**Tarjan & Vishkin, 1984).**

CONTOH (lihat gambar 2.13)

Algoritma ini mempunyai 4 (empat) fase :

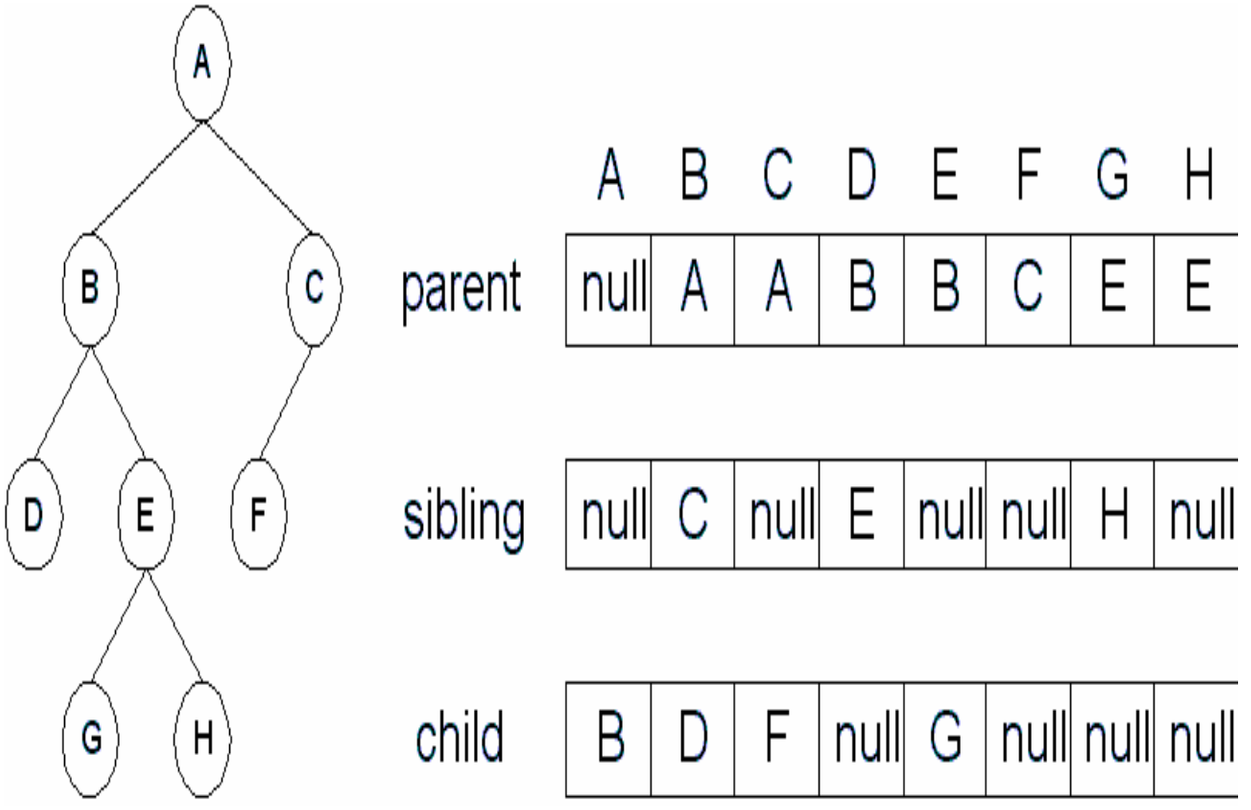
1. Algoritma membentuk singly-linked list. Setiap verteksnya mempunyai penelusuran edge turun maupun naik dari pohon
2. Memberikan bobot ke verteks-verteksnya,
   penelusuran naik (upward) : 0
   penelusuran turun (downward) : 1
3. Setiap elemen singly-linked list menghitung rank-nya dari list secara parallel
4. Prosesor yang diasosiasikan dengan edge yang turun menggunakan rank yang sudah dihitung sebagai nomor dari penelusuran preorder.



Gambar 2.13 Penelusuran dari depan (preorder traversal) dari akar pohon

(a) pohon
(b) edge-edge pohon, yang turun dan yang naik
(c) membuat linked list berdasarkan edge berarah pohon.
edge turun berbobot 1; edge naik berbobot 0
(d) jumping pointer digunakan untuk menghitung total bobot setiap verteks dari akhir list. Elemen-elemen (E, G), (E, H), (A, C) merupakan edge turun. Prosesor mengatur elemen untuk nilai preorder-nya. Misalnya elemen (E,G) berbobot 4 yang artinya node pohon G merupakan node ke-4 dari akhir preorder traversal list. Pohon memiliki 8 node sehingga node pohon G berlabel 5 pada preorder traversal
(e) nilai-nilai penelusuran dari depan.

Implementasi dari algoritma paralel preorder traversal menggunakan struktur data yang tidak biasa untuk merepresentasikan pohon.



Gambar 2.14 *Pohon berakar yang direpresentasikan dengan struktur data*
Parent : akar dari node yang ada di atasnya
Sibling : node yang merupakan tetangga sebelah kanan dari parent yang sama
Child : node paling kiri

PSEUDOCODE

```
PREORDER.TREE.TRAVERSAL (CREW PRAM):
Global          n                   {Number of vertices in tree}
                parent[1 … n]       {Vertex number of parent node}
                child[1 … n]        {Vertex number of firts child}
                sibling[1 … n]      {Vertex number of edge}
                succ[1 … (n -1)]    {Index of successor edge}
                position[1 … (n -1)] {Edge rank}
                preorder[1 … n]     {Preorder traversal number}
begin
      spawn (set of all P(i,j) where (i,j) is an edge)
      for all P(i,j) where (i,j) is an edge do
        {Put the edges into a linked list}
        if parent[i] = j then
            if sibling[i] ≠ null then
              succ[(i,j)] ← (j, sibling[i])
            else if parent[j] ≠ null then
              succ[(i,j)] ← (j, parent[j])
            else
              succ[(i,j)] ← (i,j)
              preorder[j] ← 1 {j is root of tree}
            endif
        else
```

```
            if child[j] ≠ null then succ[(i,j)] ← (j, child[j])
            else succ[(i,j)] ← (j,i)
            endif
        endif
        {Number of edges of the successor list}
        if parent[i] = j then position[(i,j)] ←  0
        else position[(i,j)] ← 1
        endif
        {Perform suffix sum on successor list}
        for k  ← 1 to ⌈log(2(n – 1))⌉ do
            position[(i,j)] ← position[(i,j)] + position[succ(i,j)]
            succ[(i,j)] ← succ[succ[(i,j)]]
        endfor
        {Assign preorder values}
        if i = parent[j] then preorder[j]  ← n + 1 - position[(i,j)]
        endif
    endfor
end
```

Gambar 2.15 ***Algoritma PRAM untuk label node pohon berdasarkan posisi secara preorder traversal***

GAMBARAN PSEUDOCODE

Sebuah pohon dengan n buah node memiliki n-1 buah edge. Karena setiap edge dibagi ke dalam edge yang “naik” dan “turun”, algoritma membutuhkan 2(n-1) prosesor untuk memanipulasi 2(n-1) elemen dari singly-linked list ke penelusuran edge-nya.

Pada saat prosesor diaktifkan, linked list dibentuk yang berisi elemen-elemen edge dari preorder traversal. Dengan edge (i, j), setiap prosesor harus menghitung successor (pengikut) dari edge dalam traversal.

Jika parent[i] = j maka edge bergerak naik pada pohon, dari node child ke node parent.

Edge-edge yang “naik” mempunyai 3 jenis successor :

- ⇨ jika child memiliki sibling, maka egde successor berasal dari node parent ke node sibling,
- ⇨ jika child memiliki grandparent, maka edge successor berasal dari node parent ke grandparent-nya,
- ⇨ jika kedua kondisi di atas tidak ada, maka edge merupakan akhir dari preorder traversal.

Akar pohon diidentitaskan dan nomor preordernya adalah 1.

Jika parent[I] j, yaitu jika edge bergerak turun dari node parent ke salah satu child-nya, maka ada 2 macam edge successornya :

- ⇨ jika node child memiliki node keturunan, edge successor berasal dari node child ke node grandchild
- ⇨ jika node child merupakan daun, edge successor berasal dari node child itu sendiri ke parent-nya.

Nilai posisi akhir menunjukkan nomor node preorder traversal antara elemen list dan akhir list. Untuk menghitung setiap label dari node, setiap prosesor yang diasosiasikan dengan edge “turun” dikurangkan nilai position dari n+1. Penambahan 1 menyebabkan penomoran preorder traversal dimulai dari 1.

KOMPLEKSITAS

Carilah, berapa kompleksitas algoritma seluruhnya ?

Merging Two Sorted Lists

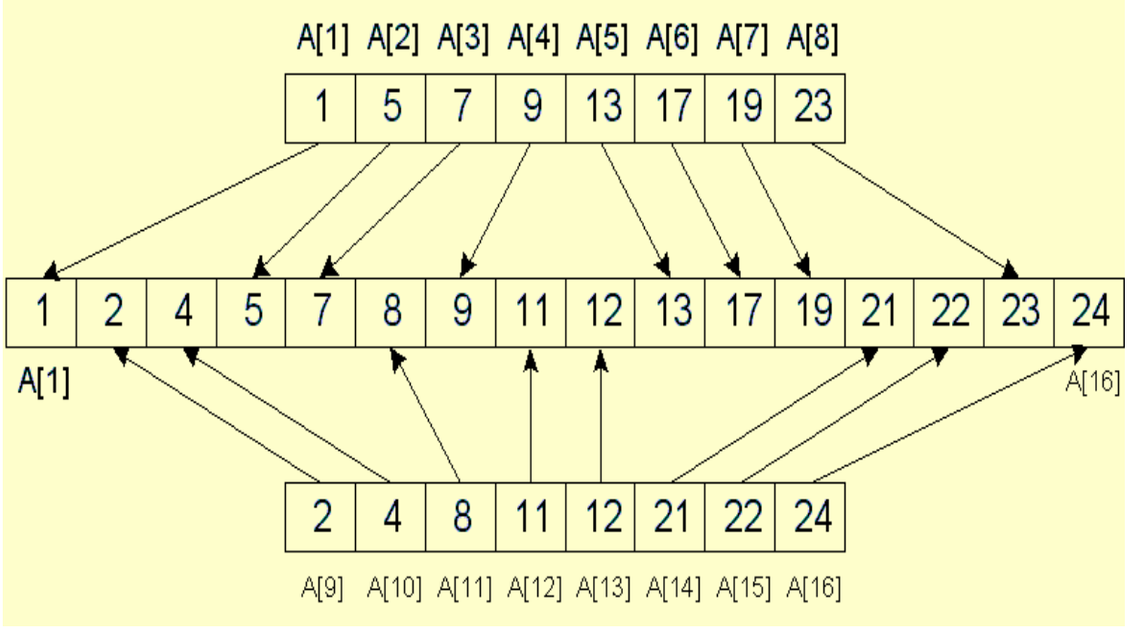
DEFINISI

Algoritma yang optimal adalah penggabungan daftar (list) untuk satu elemen setiap waktu. Untuk menggabungkan dua list secara terurut membutuhkan paling banyak n-1 perbandingan dari n/2 elemen. Waktu kompleksitasnya $\Theta(n)$. (Secara sekuensial)

Dengan menggunakan algoritma PRAM, proses penggabungan dapat dicapai dalam waktu $\Theta(n \log n)$ yaitu setiap elemen list dialokasikan ke prosesornya sendiri. Setiap prosesor menemukan posisi elemen-elemen pada list yang lain dengan pencarian biner (binary search).

Karena setiap indeks elemen pada list diketahui, tempat pada gabungan list dapat dihitung saat indeks pada list lainnya diketahui dan du indeks ditambahkan. Semua n elemen dapat dimasukkan ke gabungan list dengan prosesornya sendiri-sendiri dalam waktu konstan.

CONTOH



Gambar 2.16 *Dua list dengan n/2 elemen digabungkan dalam waktu Θ(log n)*

PSEUDOCODE

MERGE.LISTS (CREW PRAM):
Given : Two sorted lists of n/2 elements each stored in
A[1] … A[n/2] and A[(n/2)+1] … A[n]
The two lists and their unions have disjoint values
Final condition : Merged list in locations A[1] … A[n]
Global A[1 … n]
Local x, low, high, index
begin
spawn(P_1, P_2, …, P_n)
for all P_i where $1 \leq i \leq n$ do
{Each processor sets bounds for binary search}
if $i \leq n/2$ then
low $\leftarrow (n/2) + 1$
high $\leftarrow n$
else
low $\leftarrow 1$
high $\leftarrow n/2$
endif
{Each processor performs binary search}
$x \leftarrow A[i]$

```
        repeat
            index ← ⌊(low + high)/2⌋
            if x  < A[index] then
               high ← index – 1
            else
              low ← index + 1
            endif
        until low > high
        {Put value in correct position on merged list}
        A[high + i – n/2] ← x
      endfor
end
```

Gambar 2.17 *Algoritma PRAM menggabungkan dua list secara terurut.*

GAMBARAN PSEUDOCODE

Prosesor yang dibutuhkan ada n buah, satu untuk setiap elemen dari dua list yang digabungkan. Secara paralel, prosesor ini menentukan indeks yang akan dicari. Prosesor yang diasosiasikan dengan elemen dari ½ array bagian bawah akan melakukan pencarian biner pada elemen dari ½ array bagian atas, begitupula sebaliknya.

Prosesor P_i diasosiasikan dengan array A[i] bagian bawah dari list. Nilai akhir prosesor “high” harus berada antara n/2 dan n. Elemen A[i] > i-1 elemen pada bagian bawah list.
Juga A[i] > high – (n/2) untuk elemen bagian atas list.
Sehingga A[i] diletakkan pada gabungan list setelah i + high – n/2 – 1 elemen lainnya, pada indeks i + high – n/2.

Begitu pula dengan array bagian atas list. Prosesor P_i diasosiasikan dengan array A[i] bagian atas dari list. Nilai akhir prosesor “high” harus berada antara 0 dan n/2. Elemen A[i] > i – (n/2 +1) elemen lainnya pada bagian atas list.
Juga A[i] > elemen high untuk bagian bawah list.
Sehingga A[i] diletakkan pada gabungan list setelah i + high – n/2 – 1 elemen lainnya, pada indeks i + high – n/2.

Karena semua prosesor menggunakan ekspresi yang sama untuk menempatkan elemen-elemennya, setiap prosesor merelokasi elemen-elemennya menggunakan instruksi yang sama di akhir algoritma.

KOMPLEKSITAS

Secara sekuensial : $\Theta(n)$
Secara paralel : $\Theta(n \log n)$
Untuk membangun algoritma pada komputer paralel sebenarnya, “cost” algoritma paralel harus diperhitungkan.

Graph Coloring

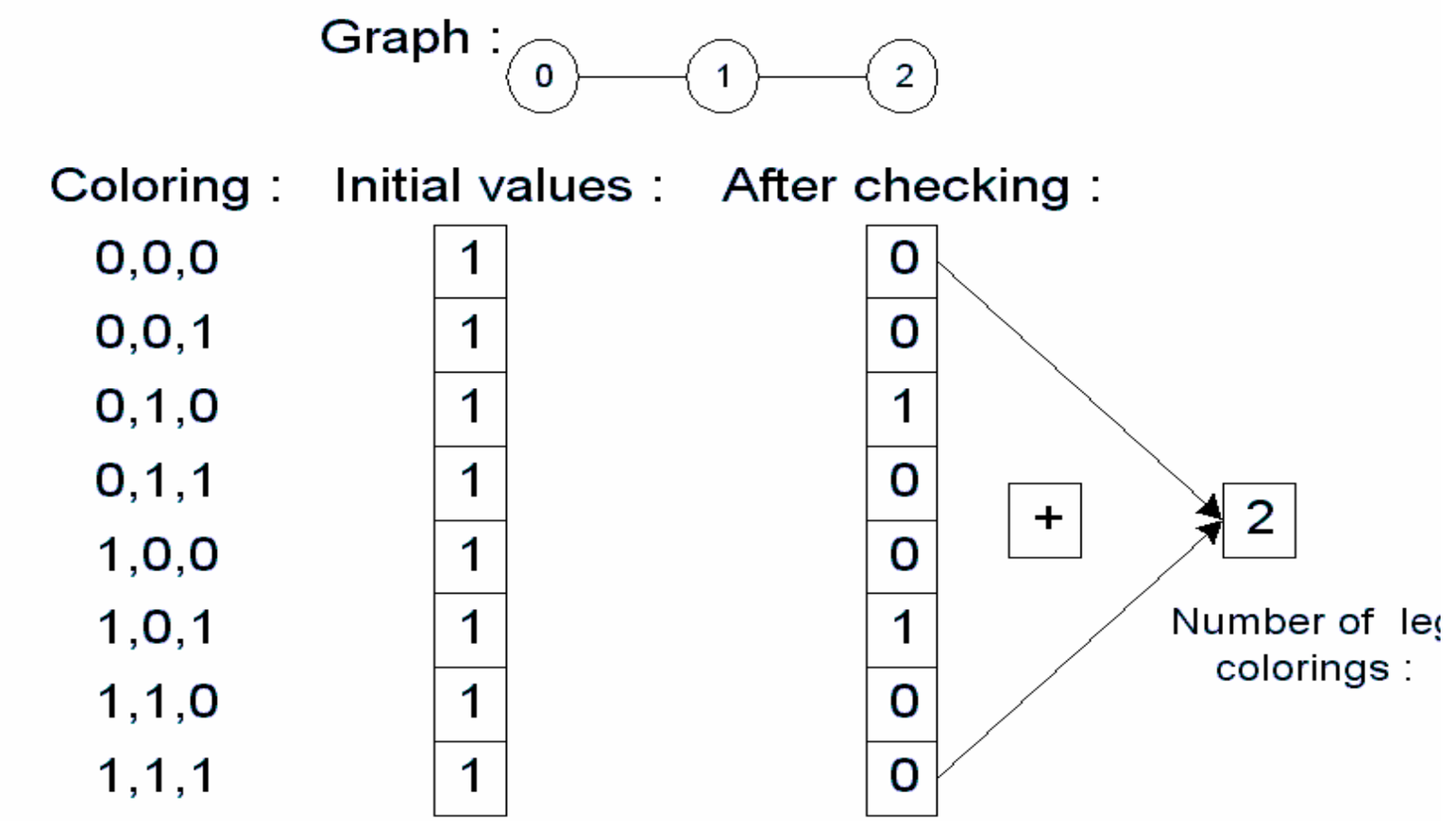
DEFINISI

Pewarnaan graf merupakan graf dimana verteks-verteks dapat diwarnai dengan c warna sehingga tidak ada dua verteks yang berdekatan (bertetangga/ ajasensi) memiliki warna yang sama.

CONTOH

Diasumsikan graf dengan n buah verteks. Diberikan matriks ajasensi (bertetangga) mxn dan konstanta positif c, sebuah prosesor dibuat untuk setiap pewarnaan graf yang mungkin.

Prosesor P(i_0, i_1, i_2, …, i_{n-1}) mengilustrasikan pewarnaan verteks 0 dengan warna i_0, verteks 1 dengan warna i_1 hingga verteks n-1 dengan warna i_{n-1}.



Gambar 2.18 *Contoh algoritma pewarnaan graf CREW PRAM Algoritma mendapatkan 2 warna untuk 3 buah verteks*

PSEUDOCODE

```
GRAPH.COLORING (CREW PRAM):
Global   n                      {Number of vertices}
         c                      {Number of colors}
         A[1…n][1…n]            {Adjacency matrix}
         candidate[1…c][1…c] … [1…c] {n-dimensional
                                boolean matrix}
         valid                  {Number of valid colorings}
         j, k
begin
    spawn(P(i0, i1, i2, …, in-1)) where 0 ≤ iv < c for 0 ≤ v < n
    for all P(i0, i1, i2, …, in-1)  where 0 ≤ iv < c for 0 ≤  v < n do
      candidate[i0, i1, i2, …, in-1] ←1
      for j ← 0 to n-1 do
          for k ← 0 to n-1 do
            if a[j][k] and ij = ik then
               candidate[i0, i1, i2, …, in] ← 0
            endif
          endfor
      endfor
      valid ← Σ candidate {Sum of all elements of candidate}
```

```
    endfor
    if valid > 0 then print “Valid coloring exists”
    else print “Valid coloring does not exist”
    endif
end
```

Gambar 2.19 *Algoritma CREW PRAM untuk menunjukkan jika graf dengan n verteks diwarnai dengan c warna*

GAMBARAN PSEUDOCODE

Setiap prosesor memulai nilainya pada array “candidate” berdimensi-n dengan 1. Waktu yang dipakai $\Theta(n^2)$ untuk mewarnai verteks yang diwakili 2 verteks yang berajasensi diberikan warna yang sama.

Jika A[j,k] = 1 dan $i_j = i_k$ maka pewarnaan salah karena A[j,k] = 1 berarti verteks j dan k bertetangga (ajasensi) dan $i_j = i_k$ berarti verteks j dan k berwarna sama. Jika hal ini terjadi, array “candidate” di-set 0.

Setelah n^2 perbandingan, jika elemen lainnya pada array “candidate” masih 1, pewarnaan benar.

Dengan menjumlah semua elemen c^n pada array “candidate”, dapat digambarkan bahwa pewarnaan benar (valid).

KOMPLEKSITAS

Rutin spawn : $\Theta(\log c^n)$,
Perulangan loop for ganda : $\Theta(n^2)$,
Menjumlah semua elemen c^n : $\Theta(\log c^n)$

Waktu kompleksitas keseluruhan :

$\Theta(\log c^n + n^2) = \Theta(n^2 + n \log c)$

Karena $c < n$,

kompleksitas berkurang menjadi $\Theta(n^2)$.