PRAM

SOMMATORIA (EREW)

```
for j=1 to logn
    for k=1 to n/2^j par do
        M[2^jk] = M[2^jk] + M[2^jk-2^(j-1)]
return M[n]
```

P(n) = n-1
T(n,P(n)) = 4logn -> n <= pot.2 <= 2n
4log2n <= 5logn
E(n,P(n)) -> 0

Pw = p
Tw = n/p + 5logp
Ew -> 1/2 se p = n/5logp

SOMME PREFISSE (Pointer Doubling) (EREW)

```
for i=1 to logn
    for k=1 to n-2^j par do
        M[S[k]] = M[k] + M[S[k]]
        S[k] = (S[k]==0? 0 : S[S[k]])
```

P(n) = n-1
T(n,P(n)) = logn
E(n,P(n)) -> 0

Pw = n/logn
Tw = logn
Ew = n-1/(n/logn * logn) -> c

VALUTAZIONE POLINOMI (EREW)

p(x) = a0 + a1x + a2x^2 + … + anx^n
Output: p(alpha)

```
A: A[0]=a0, A[1]=a1, …, A[n]=an

- REPLICA alpha -> a, a, a, … utilizzando
  somme-prefisse su Q= [a,0,0,…]
- PRODOTTO PREFISSO Q ->
  Q[1]=alpha, …, Q[n]=alpha^n
- Prodotto vettori <A,Q>
```

Pw = n/logn
Tw = logn

RICERCA ELEMENTO

1) CRCW

```
for i=1 to n par do
   if (M[i]==alpha)
       M[n] = 1
```

P=n
T=c

2) CREW

```
for i=1 to n par do
   M[i]= (M[i]==alpha?
1:0)
MAX-ITERATO M
```

Pw=n/logn
Tw=logn
Ew=c

3) EREW

```
REPLICA alpha in A
for i=1 to n par do
   M[i]= (M[i]==alpha?
1:0)
```

Pw=n/logn
Tw=logn
Ew=c

REV (EREW)

```
for i=1 to n/2 par do
   SWAP(A[i],A[n-k+1])
```

MINMAX (EREW)

```
for i=1 to n/2 par do
   if A[i]>A[i+(n/2)]
      SWAP(A[i],A[i+(n/
```

ORDINAMENTO

1) COUNTING SORT (CREW)

```
for i,j=1,..n par do
    V[i]= (M[j]<M[i]? 1:0)
for i=1 to n par do
    SOMMATORIA in V[i,n]
for i=1 to n par do
    M[V[i,n]]=M[i]ù
```

Pw=n^2/logn
Tw=logn
Ew=logn/n->0

2) BIT MERGE(A bitonica) (EREW)

```
MINMAX A
if |A|>2 then
   bit-merge(Amin)
   bit-merge(Amax)
return Amin.Amax
```

P=n/2
T=logn
E=c

3) BIT SORT (A generica) (EREW)

```
MINMAX A
if |A|>2 then
   bit-sort(Amin)
   bit-sort(Amax)
   bit-merge(Amin.REV(Amax))
return Amin.Amax
```

P=n/2
T=log^2(n)
E->0
lentamente

CICLO EURELIANO

utilizzato con alberi binari per costruire delle strutture dati che poi risultano efficienti per risolvere problemi di attraversamento dell'albero

```
albero -> ciclo eureliano -> (vs)(vc)(vd)
-> S[vx] se v foglia/nodo interno
```

P=n Pw=n/logn
T=c Tw=logn

1) ATTRAVERSAMENTO PREORDINE (EREW)

```
A[vx]= 1 se x=s oppure 0 se x=c o x=d
ordino A secondo S
SOMME-PREFISSE su A
risultato in A[vs]
```

Pw=n/logn
Tw=logn
Ew=c

2) CALCOLO PROFONDITA (EREW)

```
A[vx] = 1 se x=s
        0 se x=c
        -1 se x=d
poi come prima
```

ARCHITETTURE PARALLELE A MEM. DISTRIBUITA

ARRAY LINEARI

Grado = 2
Diametro = n-1 lower b MAX = n
Amp. bisezione = 1 lower b ORD = n/2B=n/2

1) SHUFFLE

```
|A|=n=2S
S-1 swap ad albero
```

P=2(S-1)
T=3(S-1)
E=c

2) MAX

```
for j=1 to logn
    for k∈{(2^j)t-2^(j-1)}(1<=t<=n/2^j) par do
        SEND(k, k+2^(j-1))
    for k∈{(2^j)t}(1<=t<=n/2^j) par do
        if A[k]<A[k-2^(j-1)]
            A[k]=A[k-2^(j-1)]
```

P=n
T=O(n)
E->0

Pw=$\sqrt{p}$
Tw = O($\sqrt{p}$)
E = c

3) ORDINAMENTO (ODD-EVEN)

```
for i=1 to n
    for k∈{2t-(i%2) con 1<=t<=n/2} par do
        MINMAX(k, k+1)
```

P=n
T=n
E->0

Wyllie -> MINMAX diventa MERGE-SPLIT
Pw=logn
Tw=n
E=c

MESH

m = $\sqrt{n}$
Grado = 4
Diametro = 2$\sqrt{n}$ lower b MAX = O($\sqrt{n}$)
Amp. bisezione = m lower b ORD=n/2B = O($\sqrt{n}$)

1) MAX

```
Algoritmo Righe-Colonna
for i=1 to √n par do
    MAX (Pi1, Pi2, …, Pim)
MAX (P1m, P2m …, Pmm)
```

P=n
T=O($\sqrt{n}$)
E->0

Pw=n^(2/3)
Tw=n/p+O($\sqrt{n}$) = O($\sqrt[3]{n}$)
Ew=1/2

2) ORDINAMENTO

```
Algoritmo LS3sort(M)
    if |M|==1
        return M
    else
        M1 = LS3sort(M1)
        M2 = LS3sort(M2)
        M3 = LS3sort(M3)
        M4 = LS3sort(M4)
        LS3merge(M1, M2, M3, M4)

Procedura LS3merge(M1, M2, M3, M4)
for i=1 to √n par do
    SHUFFLE M[i]
for i=1 to (√n)/2 par do
    ODD-EVEN (M[2i-1], M[2i])
esegui i primi 2√n passi di ODD-EVEN su M
```

P=n
T=O($\sqrt{n}$)
E->0

ARCHITETTURE DISTRIBUITE

BROADCAST (RI)

SOLUZIONE 1

```
Sinit = {iniziatore, inattivo}
Sterm = {inattivo}

iniziatore X impulso sp -> {
        send(M) to N(x);
        become inattivo;
}
inattivo X ricezione(M) -> {
        processa(M)
        send(M) to N(x);
}
```

SOLUZIONE 2

```
Sinit = {iniziatore, inattivo}
Sstart = {iniziatore}
Sterm = Sfinal = {finito}

iniziatore X impulso sp -> {
        send(M) to N(x);
        become finito;
}
inattivo X ricezione(M) -> {
        processa(M)
        send(M) to N(x)-sender;
        become finito;
}
```

M = 2m-n+1 LB: M>=m

T <= d (diametro) LB: T>=d

WAKE UP

```
Sinit = {dormiente}
Sstart = {attivo}
Sterm = Sfinal = {attivo}

dormiente X impulso sp -> {
        send(M) to N(x);
        become attivo;
}
dormiente X ricezione(M) -> {
        processa(M)
        send(M) to N(x)-sender;
        become attivo;
}
2m-n+1 <= M <= 2m
T <= d
```

TRAVERSAL (RI)

SOLUZIONE 1

```
Sinit = {initiator, idle}
Sterm = {inattivo}

initiator X impulso sp -> {
        initiator = true;
        unvisited = N(x);
        VISIT;
}
idle X ricezione(T) -> {
        initiator = false;
        unvisited = N(x)-sender;
        VISIT;
}

Procedure VISIT{
   if unvisited <> ∅ then
      next <- unvisited;
      send(T) to next;
      become visited;
   else
      if initiator=false then
          send(R) to sender
      become done;
}

visited X receiving(R/B) ->{
        VISIT;
}
visited X receiving(T) -> {
   unvisited = unvisited-this_sender;
   send(B) to this_sender;
}
```

T = M = 2m LB: M >= m T >= n-1

SOLUZIONE 2

un nodo non visitato che riceve T manda ai suoi vicini un messaggio VISITED e loro si aggiornano

M = O(m) T = 2(n-1)

SPANNING TREE (RI)

PROTOCOLLO SHOUT

```
Sinit = {iniziatore, inattivo}
Sterm = {finito}

iniziatore X impulso sp -> {
        root=true;
        counter=0;
        Tree_N(x)=∅;
        send(Q) to N(x);
        become attivo;
}
inattivo X ricezione(Q) -> {
        root=false;
        parent=sender;
        counter=1;
        send "yes" to sender;
        Tree_N(x)={sender};
        if counter=|N(x)| then
           become finito;
        else
           send(Q) to N(x)\sender;
           become attivo;
}
attivo X ricezione("yes") ->{
        counter++;
        Tree_N(x)=Tree_N(x)U{sender};
        if counter=|N(x)| then
           become finito;
}
attivo X ricezione("no") -> {
        counter++;
        if counter=|N(x)| then
           become finito;
}
attivo X ricezione(Q) -> {
        send "no" to sender;
}
M = 4m  LB: M>=m
T <= d+1        LB: T>=d
```

PROTOCOLLO SHOUT+

eliminazione del "no" quando un attivo riceve Q. Un'entità attiva o riceve "yes" o riceve un Q, che ignora

M = 2m

ELECTION (IR)

ELECT MINIMUM - ring

```
Sinit = {Asleep}
Sterm = {leader, follower}

Asleep X impulso sp -> {
   INITIALIZE;
   become awake;
}
Asleep X
ricezione("Elect",value, counter) -> {
   INITIALIZE;
   send("Elect", value, counter++) to other
   min = Min{min, value}
   count++;
   become awake;
}
Procedure INITIALIZE ->{
   count=0;
   size=1;
   know=false;
   send("Elect", id(x), size) to right
   min=id(x)
}
awake X
receiving("Elect", value, counter) -> {
   if value<>id(x) then
     send("Elect", value, counter++) to other
     min=Min(min, value)
     count++
     if know=true then CHECK
   else
     size=counter
     know=true
     CHECK
}
Procedure CHECK{
   if count=size then
      if min=id(x) then become leader
      else become follower
}
```

M = n^2

ROUTING (IR + FRT)

Dest	S.path	Cost
h	(s, h)	1
k	(s, h) (h, k)	4
…		

PROTOCOLLO GOSSIPING

- costruzione albero T da G
- ogni entità manda ai vicini (id+costo link)
- broadcast queste info da parte di tutti
- ogni entità alla fine avrà la matrice di incidenze del grafo, dalla quale si può costruire la FRT

PROTOCOLLO ITERATED-CONSTRUCTION

- ogni entità invia la sua Distance Vector
- se un entità capisce di pagare di meno per arrivare ad una dest allora aggiorna la sua FRT

w[z] = Min y∈N(X){ d(x, y)+ Vyi[z] }

if w[z] < Vxi[z] => update FRT