CAPITOLO 27: *Algoritmi PRAM*

Vettori

Sommatoria di n el.ti in a[n..2n-1]. P=O(n) e T=O(log n). EREW.
```
SOMMATORIA(a:vect; n:int):int
  for k=log(n)-1 downto 0 do
    forall j where 2^k≤j≤2^(k+1)-1 pardo
      a[j]=SUM(a[2j], a[2j+1]);
  SOMMATORIA=a[1];
```

Sommatoria2 di n el.ti in a[n..2n-1]. EREW. P=O(n²/log n) e T=O(log n).
```
SOMMATORIA2(a:vect; n:int):int
  h=n/log(n);
  forall j where 1≤j≤h pardo
    m=n+(j-1)·log(n);
    a[h+j-1]=a[m];
    for k=1 to log(n)-1 do
      a[h+j-1]=SUM(a[h+j-1], a[m+k]);
  SOMMATORIA2=SOMMATORIA(a, h);
```

Somme prefisse di n el.ti in a[n..2n-1]. P=O(n) e T=O(log n). EREW.
```
SOMME_PREFISSE(a,b:vect; n:int);
  b[1]=SOMMATORIA(a,n); //PRODUTTORIA(a,n)
  for k=1 to log(n) do
    forall j where 2^k≤j≤2^(k+1)-1 pardo
      if ODD(i)   then b[i]=b[i/2];
                  else b[i]=b[i/2]-a[i+1];
  //in prod. pref.: b[i]=b[i/2]/a[i+1]
```

Somme prefisse2 di n el.ti in a[n..2n-1]. EREW. (Versione 1)
P=O(n/log n) e T=O(log n).
```
SOMME_PREFISSE2(a,b:vect; n:int);
  h=n/log(n);
  b[1]=SOMMATORIA2(a, n); //o PRODUTTORIA2(a,n)
  SOMME_PREFISSE(a, b, h); //o PROD_PREF(a,b,h)
  forall j where 1≤j≤h pardo
    m=n+j·log(n)-1;
    b[m]=b[h+j-1];
    for k=1 to log(n)-1 do
      b[m-k]=b[m-k+1]-a[m-k+1];
  // in prod. pref.: b[m-k-1]=b[m-k]/a[m-k]
```

Somme prefisse2 di n el.ti in a[n..2n-1]. EREW. (Versione 2)
P=O(n/log n) e T=O(log n).
```
SOMME_PREFISSE2(a,b:vect; n:int);
  h=n/log(n);
  forall j where 1≤j≤h pardo
    m=n+(j-1)·log(n);
    a[h+j-1]=a[m];
    for k=1 to log(n)-1 do
      a[h+j-1]+=a[m+k]; //in prod. pref.: *=
  SOMME_PREFISSE(a, b, h); //o PROD_PREF(a,b,h)
  forall j where 1≤j≤h pardo
    m=n+j·log(n)-1;
    b[m]=b[h+j-1];
    for k=1 to log(n)-1 do
      b[m-k]=b[m-k+1]-a[m-k+1];
  // in prod. pref.: b[m-k]=b[m-k+1]/a[m-k]
```

Replica n copie di d in a[1..n]. P=O(n) e T=O(log n). EREW.
```
REPLICA(a:vect; d,n:int);
  a[1]=d;
  for k=0 to log(n)-1 do
    forall j where 1≤j≤2^k pardo
      a[i+2^k]=a[i];
```

Replica2 n copie di d in a[1..n]. P=O(n/log n) e T=O(log n). EREW.
```
REPLICA2(a:vect; d,n:int);
  a[1]=d;
  for k=0 to log(n/log(n))-1 do
    forall j where 1≤j≤2^k pardo
      h=(j-1)·log(n);
      a[h+2^k·log(n)+1]=a[h+1];
  forall j where 0≤j≤n/log(n)-1 pardo
    for k=1 to log(n)-1 do
      h=j·log(n)+k;
      a[h+1]=a[h];
```

Torneo per ordinare n el.ti distinti in a[1..n]. EREW.
P=O(n²) e T=O(log n).
```
TORNEO(a:vect; n:int);
  for i=1 to n do
    REPLICA(b[i,*], a[i], n);
    REPLICA(c[*,i], a[i], n);
  forall i,j where 1≤i,j≤n pardo
    if b[i,j]≤c[i,j]   then v[i,j]=1
                       else v[i,j]=0;
  forall j where 1≤j≤n pardo
    p[j]=SOMMATORIA(v[*,j], n);
  forall j where 1≤j≤n pardo
    a[ p[j] ]=b[j,j];
```

Torneo2 per ordinare n el.ti distinti in a[1..n]. EREW.
P=O(n²/log n) e T=O(log n).
```
TORNEO2(a:vect; n:int);
  forall j where 1≤j≤n pardo
    REPLICA2(r[j,*], a[j], n);
    REPLICA2(s[*,j], a[j], n);
  forall i,j where 1≤i≤n,0≤j≤n/log(n)-1 pardo
    for k=1 to log(n) do
      h=j·log(n)+k;
      if r[i,h]≤s[i,h]   then v[i,h]=1;
                         else v[i,h]=0;
  forall j where 1≤j≤n pardo
    p[j]=SOMMATORIA2(v[*,j], n);
  forall j where 1≤j≤n pardo
    a[ p[j] ]=s[1,j]
```

Torneo per ordinare n el.ti non distinti in a[1..n]. EREW.
P=O(n²) e T=O(log n).
```
TORNEO_NON_DISTINTI(a:vect; n:int);
  forall i where 1≤i≤n pardo
    REPLICA(b[i,*], a[i], n);
    REPLICA(c[*,i], a[i], n);
  forall i,j where 1≤i,j≤n pardo
    if b[i,j]<c[i,j]   then v[i,j]=1
                       else v[i,j]=0;
  forall j where 1≤j≤n pardo
    p[j]=SOMMATORIA(v[*,j], n);
  forall i,j where 1≤i,j≤n pardo
    if b[i,j]==c[i,j] AND i<j   then q[i,j]=1
                                else q[i,j]=0;
  forall j where 1≤j≤n pardo
    r[j]=SOMMATORIA(n[*,j], n);
  forall j where 1≤j≤n pardo
    a[ p[j]-r[j] ]=b[j,j];
```

Valutazione nel punto q di un polinomio. EREW.
P=O(n/log n) e T=O(log n).
```
VALUTA_POLINOMIO(a:vect; n:num):num;
  REPLICA2(x[n..2n-1], q, n-1);
  x[n]=1;
  PRODOTTI_PREFISSI2(x, b, n);
  forall j where 0≤j≤n/log(n)-1 pardo
    for k=0 to log(n)-1 do
      h=j·n/log(n)+k;
      r[n+h]=a[h]+b[n+h];
  SOMMATORIA2(r, n);
  VALUTA_POLINOMIO=r[1];
```

Valutazione del massimo in un vettore a[1..n]. CRCW.
P=O(n²) e T=O(1).
```
MAX_CRCW(a:vect; n:int):int;
  forall i,j where 1≤i,j≤n pardo
    if a[i]≤a[j]   then v[i,j]=1;
                   else v[i,j]=0;
  forall j where 1≤j≤n pardo
    r[j]=AND_CRCW(v[*,j], n);
    if r[j]=1   then result=a[j];
  MAX_CRCW=result;
```

Valutazione della posizione del minimo in un vettore a[1..n].
P=O(n) e T=O(log n). EREW.
```
POSIZIONE_MIN(a:vect; n:int):int;
  min=MINIMO2(a, n);
  k=1;
  for i=1 to log(n) do
    if a[2k]==min   then k=2k;
                    else k=2k+1;
  POSIZIONE_MIN=k;
```

Valutazione della posizione del minimo in un vettore a[1..2n-1] con lavoro ottimo. P=O(n/log n) e T=O(log n). EREW.
```
POSIZIONE_MIN2(a:vect; n:int):int;
  min=MINIMO2(a, n);
  k=1;
  for i=1 to log( log(n) ) do
    if a[2k]==min   then k=2k;
                    else k=2k+1;
  g=n+(k-n/log(n))·log(n);
  for i=1 to log(n) do
    if a[g+i]==min   then ret=g+i;
  POSIZIONE_MIN2=ret;
```

Operatori logici

Calcolo del AND logico in un vettore a[1..n] di booleani. CRCW.
P=O(n) e T=O(1).
```
AND_CRCW(a:vect; n:int):bool;
  x=1;
  forall j where 1≤j≤n pardo
    if b[i]==0   then x=false
  AND_CRCW=x;
```

Calcolo degli op. logici prefissi in un vettore a[n..2n-1] booleani.
P=O(n/log n) e T=O(log n). EREW.
```
OP_LOGICI_PREFISSI(a,b:vect; n:int)
  SOMMEPREFISSE2(a, b, n);
  forall j where 0≤j≤n/log(n)-1 pardo
    for k=0 to log(n)-1 do
      h=j·log(n)+k;
      //if b[n+h]>0       // OR  prefissi
      //if ODD(b[n+h])==1 // XOR prefissi
      if b[n+h]==h+1 then b[n+h]=1; // AND pre
      else b[n+h]=0;
```

Liste

Replica n copie di d in lista lunga n. EREW. P=O(n) e T=O(log n).
```
REPLICA_LISTA(a:list; d,n:int);
  a[1]=d;
  for k=1 to log(n) do
    forall j where 1≤j≤n pardo
      if succA[j]<>null   then
        a[ succA[j] ]=a[j];
        succA[j]=succA[ succA[j] ];
```

Somme prefisse in lista di n elementi. EREW. P=O(n) e T=O(log n).
```
SOMME_PREF_LISTA(a:list; n:int);
  for k=1 to log(n) do
    forall j where 1≤j≤n pardo
      if succA[j]<>null   then
        a[ succA[j] ]+=b[j];
        succA[j]=succA[ succA[j] ];
```

Calcolo del num. di el.ti che seguono ciascun el.to (rango)
in una lista lunga n. P=O(n) e T=O(log n). EREW.
```
RANGO_LISTA(a,b:list; n:int);
  forall j where 1≤j≤n pardo
    b[j]=1;
    succB[j]=succA[n-j];
  SOMME_PREFISSE_LISTA(b, succB, n);
  forall j where 1≤j≤n pardo
    b[j]=b[j]-1;
```

Partizione di una lista di n el.ti booleani in 2 liste veri e falsi.
EREW. P=O(n) e T=O(log n).
```
LISTE_VERI_FALSI(a,b,t,f:list; n:int);
  TRASFORMA_LISTA_BOOL_INT(a, b, n);
  sum=SOMMATORIA_LISTA(b, succB, n);
  forall j where 1≤j≤sum do
    t[i]=true;
    if j<sum   then succT[j]=j+1;
               else succT[j]=null;
  forall j where sum+1≤j≤n do
    f[i]=false;
    if j<n   then succF[j]=j+1;
             else succF[j]=null;
```

Alberi

Calcolo del ciclo euleriano su un albero di n nodi, per ottenere una lista lunga 3n. P=O(n) e T=O(1). EREW.
```
CICLO_EULERIANO(a:tree; b:list; x,y,z,n:int)
  forall j where 1≤j≤n pardo
    b[j]=x;  b[j+n]=y;  b[j+2n]=z;
    if sxA[j]<>null
      then succB[j]=sxA[j];
      else succB[j]=j+n;
    if dxA[j]<>null
      then succB[j+n]=dxA[j];
      else succB[j+n]=j+2n;
    if j<>dxA[ topA[j] ]
      then succB[j+2n]=topA[j]+n;
      else succB[j+2n]=topA[j]+2n;
```

Calcoli vari sugli alberi binari usando il ciclo euleriano. Ritorna un albero strutturato come quello in input ma con i valori del risultato nei singoli nodi. P=O(n) e T=O(log n). EREW.
Livello nodi: x=1, y=0, z=-1. FUNC=b[j+2n];
Dim. sottoalberi: x=0, y=0, z=1. FUNC=b[j+2n]-b[j];
Visita preorder: x=1, y=0, z=0. FUNC=b[j];
Visita inorder: x=0, y=1, z=0. FUNC=b[j+n];
Visita postorder: x=0, y=0, z=1. FUNC=b[j+2n];
```
ALB_BIN_POLIMORFA
(a,res:tree; x,y,z,n:int; FUNC:function);
  CICLO_EULERIANO(a, res, x, y, z, n);
  SOMME_PREF_LISTA(b, succB, 3n);
  forall j where 1≤j≤n pardo
    res[j]=FUNC(b, j, n);
    sxRes[j] =sxA[j];
    dxRes[j] =dxA[j];
    topRes[j]=topA[j];
```

Calcolo della radice dei nodi in una foresta di alberi (lista dei padri).
P=O(n) e T=O(log n). CREW.
```
RADICI_ALB_BIN(a:list; n:int);
  for k=1 to log(n) do
    forall j where 1≤j≤n pardo
      if topA[j]<>null   then
        topA[j]=topA[ topA[j] ];
```

CAPITOLO 28: *Reti a grado limitato*

Ipercubo

Sommatoria su ipercubo, di grado k, di n=2^k el.ti in $A_0..A_{n-1}$.
Risultato nel primo processore. P=O(n) e T=O(log n).

```
SOMMATORIA_IPERCUBO(A0..An-1, n:int);
  for k=log(n)-1 downto 0 do
    h=2^k;
    forall j where 0≤j≤h-1 pardo
      Bj←Aj+h;
      Aj=SUM(Aj, Bj);
```

Sommatoria su ipercubo, di grado k, di n=2^k el.ti in $A_0..A_{n-1}$.
Risultato in tutti i processori. P=O(n) e T=O(log n).

```
SOMMATORIA_IPERCUBO_DIFF(A0..An-1, n:int);
  for k=0 to log(n)-1 do
    forall j where 0≤j≤n-1 pardo
      TMPj←ACOMPL(j,k);
      Aj=SUM(Aj, TMPj);
```

Sommatoria2 su ipercubo di n el.ti in $A_0..A_{n-1}$.
Risultato in tutti i processori. P=O(n/log n) e T=O(log n).

```
SOMMATORIA2_IPERCUBO_DIFF
(A0[1..log(n)]..An/log(n)-1[1..log(n)]:vect; n:int);
  forall j where 0≤j≤n/log(n)-1 pardo
    for k=1 to log(n)-1 do
      Aj[i+1]=SUM(Aj[i], Aj[i+1]);
  for k=0 to log(n/log(n))-1 do
    forall j where 0≤j≤n/log(n)-1 pardo
      TMPj←ACOMPL(j,i)[log(n)];
      Aj[log(n)]=SUM(Aj[log(n)], TMPj);
```

Verifica di appartenenza di un el.to d (da diffondere) ad un insieme $A_0..A_{n-1}$, su ipercubo. P=O(n/log n) e T=O(n).

```
RICERCA_IPERCUBO
(A0[1..log(n)]..An/log(n)-1[1..log(n)]:vect;
 B0..Bn/log(n)-1:bool; d,n:int);
  V0=d;
  DIFFUSIONE_IPERCUBO(V, n);
  forall j where 0≤j≤n/log(n)-1 pardo
    Bj=false;
    for k=1 to log(n) do
      if Vj==Aj[k]   then Bj=true;
  for i=0 to log(n)-1 do
    forall j where 0≤j≤n/log(n)-1 pardo
      TMPj←BCOMPL(j,i);
      Bj=Bj OR TMPj;
```

Diffusione di un el.to d in tutto il vettore $A_0..A_{n-1}$ su ipercubo.
P=O(n) e T=O(log n).

```
DIFFUSIONE_IPERCUBO(A0..An-1, d,n:int);
  for k=0 to log(n)-1 do
    forall j where BIT(j,k)==1 pardo
      Aj←ACOMPL(j,i);
```

*Moltiplicazione matrici di dimensione m=n*n su ipercubo.*
Risultato in tutti i processori. P=O(n^3) e T=O(log n).

```
MOLT_MATR_IPERCUBO_DIFF(A0..An^2-1, B0..Bn^2-1,
                        C0..Cn^2-1, n:int);
  h=log(n);
  for p=2h to 3h-1 do
    forall j where 2^p≤j≤2^p+1-1 pardo
      Aj←ACOMPL(j,p);
      Bj←BCOMPL(j,p);
  for p=0 to h-1 do
    forall j where BIT(j,p)<>BIT(j,2h+p) pardo
      Aj←ACOMPL(j,p);
  for p=h to 2h-1 do
    forall j where BIT(j,p)<>BIT(j,h+p) pardo
      Bj←BCOMPL(j,p);
  forall j where 0≤j≤2^3h-1 pardo
    Cj=Aj·Bj; // Cj=Aj+Bj per QUADRATURA()
  for p=2h to 3h-1 do
    forall j where 0≤j≤2^3h-1 pardo
      Dj←DCOMPL(j,p);
      Cj+=Dj; // Cj=MIN(Cj,Dj) per QUADRATURA()
```

Calcolo della matrice dei cammini minimi su ipercubo.
P=O(n^3) e T=O(log^2 n).

```
CAMMINI_MINIMI_IPERCUBO(A0..An^2-1,
                        B0..Bn^2-1, n:int);
  forall j where 0≤j≤n^2 pardo
    Cj=Aj;
  for k=1 to log(n) do
    QUADRATURA_MATRICI_IPERCUBO(C, C, B, n);
    forall j where 0≤j≤n^2 pardo
      Cj=Bj;
```

Shuffle

Sommatoria su shuffle di n=2^k el.ti in $A_0..A_{n-1}$.
Risultato in tutti i processori. P=O(n) e T=O(log n).

```
SOMMATORIA_SHUFFLE(A0..An-1, n:int);
  for k=1 to log(n) do
    forall j where 0≤j≤n-1 pardo
      MESCOLA(Aj);
      Bj=Aj;
      SCAMBIA(Bj);
      Aj=SUM(Aj, Bj);
```

*Trasposizione di una matrice n*n su shuffle in $A_0..A_{n^2-1}$.*
L'el.to $A_{i,j}$ è memorizzato nel proc. P_k con k=n·(i-1)+(j-1).
P=O(n^2) e T=O(log n).

```
TRASPOSTA_SHUFFLE(A0..An^2-1, B0..Bn^2-1, n:int);
  forall k where 0≤k≤n^2-1 pardo
    Bk=Ak;
  for p=1 to log(n) do
    forall k where 0≤k≤n^2-1 pardo
      MESCOLA(Bk);
```

Verifica di appartenenza di un el.to d (da diffondere) ad un insieme $A_0..A_{n\ log(n)-1}$ su shuffle. P=O(n) e T=O(log n).

```
RICERCA_SHUFFLE(A0..An-1, V0..Vn-1, d,n:int);
  X0=Y0=d;
  for p=1 to log(n) do
    forall j where 0≤j≤n-1 pardo
      MESCOLA(Yj);
      Xj=Yj;
      SCAMBIA(Yj);
      Xj=Yj;
  forall j where 0≤j≤n-1 pardo
    if Aj==Xj   then Vj=1;
                else Vj=0;
  for p=1 to log(n) do
    forall j where 0≤j≤n-1 pardo
      MESCOLA(Vj);
      Bj=Vj;
      SCAMBIA(Bj);
      Vj=Vj OR Bj;
```

Mesh

Sommatoria su mesh di n=p^2 el.ti in $A_{1,1},..,A_{1,p},..,A_{p,p}$.
Risultato solo nel primo processore $P_{1,1}$. P=O(n) e T=O($\sqrt{n}$).

```
SOMMATORIA_MESH(A=[ai,j], n:int);
  for p=√n -1 downto 1 do
    forall i,j where i=p,1≤j≤√n  pardo
      Bi,j←Ai+1,j;
      Ai,j=SUM(Ai,j, Bi,j);
  for p=√n -1 downto 1 do
    forall i,j where i=1,j=p pardo
      Bi,j←Ai,j+1;
      Ai,j=SUM(Ai,j, Bi,j);
```

*Moltiplicazione matrici su mesh ciclica di n*n el.ti ($a_{n,n}$*$b_{n,n}$).*
Risultato C nel primo processore ($C_{i,j}$ iniz. a 0). P=O(n^2) e T=O(n).

```
MOLT_MATR_MESH_CICLICA(A=[ai,j], B=[bi,j],
                       C=[ci,j], n:int);
  for p=1 to n-1 do
    forall i,j where 1≤i,j≤n pardo
      if i>p   then Ai,j←Ai,j+1;
      if j>p   then Bi,j←Bi+1,j;
  for q=1 to n do
    forall i,j where 1≤i,j≤n pardo
      Ci,j+=Ai,j·Bi,j;
      Ai,j←Ai,j+1;
      Bi,j←Bi+1,j;
```

*Trasposizione di una matrice su mesh di n*n el.ti.*
P=O(n^2) e T=O(n).

```
TRASPOSTA_MESH(A=[ai,j], n:int);
  forall i,j where 1≤i,j≤n pardo
    Bi,j←Ai,j;
    Ci,j←Ai,j;
  for k=1 to n-1 do
    forall i,j where 1≤i,j≤n pardo
      if j>1   then Bi,j=Bi,j-1; // → di B
      if i>1   then Bi,j=Bi+1,j; // ↑ di B
      if j>n   then Ci,j=Ci,j+1; // ← di C
      if i>n   then Ci,j=Ci-1,j; // ↓ di C
      if k=|i-j|   then Ai,j=Bi,j;
                   else Ai,j=Ci,j;
```

Butterfly

Sommatoria di O(n log^2 n) el.ti in A=[$a_{i,j}$[1..log(n)]] su butterfly di rango k, con n=2^k colonne e log(n)+1=k+1 righe. Ogni proc. Ha log(n) el.ti, lavoro ottimo. Risultato in tutti i processori in S=[$s_{i,j}$]. P=n·(log(n)+1) e T=O(log n).

```
SOMMATORIA_BUTTERFLY(A=[ai,j[1..log(n)]],
                     S=[si,j], n:int);
  // se el.ti==(n·log^2 n):   0≤i≤log(n)-1
  forall i,j where 0≤i≤log(n),0≤j≤n-1 pardo
    Si,j=Ai,j[1];
    for p=2 to log(n) do
      Si,j+=Ai,j[p];
  // se el.ti==(n·log^2 n):   p=1 to log(n) do
  for p=1 to log(n)+1 do
    forall i,j where i=p,0≤j≤n-1 pardo
      Bi,j←Si-1,j;
      Si,j+=Bi,j;
  // se el.ti==(n·log^2 n):   p=log(n)-1 downto
  for p=log(n) downto 0 do
    forall i,j where i=p,0≤j≤n-1 pardo
      Bi,j←Si+1,j;
      Ci,j←Si+1,COMPL(j,log(n)-i);
      Si,j=Bi,j+Ci,j;
  for p=1 to log(n)+1 do
    forall i,j where i=p,0≤j≤n-1 pardo
      Si,j←Si-1,j;
```

Conta degli 0 e 1 in A=[$a_{i,j}$[1..log(n)]] di O(n log^2 n) booleani su butterfly. Ogni proc. ha log(n) el.ti, lavoro ottimo. Risultato in tutti i processori in D=[$d_{i,j}$]. P=n·(log(n)+1) e T=O(log n).

```
CONTA_0_1_BUTTERFLY(A=[ai,j[1..log(n)]],
                    D=[di,j], n:int);
  // se el.ti==(n·log^2 n):   0≤i≤log(n)-1
  forall i,j where 0≤i≤log(n),0≤j≤n-1 pardo
    Bi,j=0;
    for p=0 to log(n)-1 do
      if Ai,j[p]==1   then Bi,j+=1;
  // se el.ti==(n·log^2 n):   p=1 to log(n) do
  for p=1 to log(n)+1 do
    forall i,j where i=p,0≤j≤n-1 pardo
      Ci,j←Bi-1,j;
      Bi,j+=Ci,j;
  // se el.ti==(n·log^2 n):   p=log(n)-1 downto
  for p=log(n) downto 0 do
    forall i,j where i=p,0≤j≤n-1 pardo
      Ci,j←Bi+1,j;
      Di,j←Bi+1,COMPL(j,log(n)-i);
      Bi,j=Ci,j+Di,j;
  forall i,j where i=0,0≤j≤n-1 pardo
    Ci,j=n-Bi-1,j;
    Di,j=Bi,j-Ci,j;
  for p=1 to k+1 do
    forall i,j where i=p,0≤j≤n-1 pardo
      Di,j←Di-1,j;
```

Fourier

FFT sequenziale su un polinomio di n coeff. $A_0..A_{n-1}$.
P=1 e T=O(n log n).

```
FFT_SEQUENZIALE(A,B:vect; n,isign:int);
  w=cos(isign·2π/n)-i·sen(isign·2π/n);
  w1=1;
  for i=0 to n/2-1 do
    pariA[i]=A[2i];
    dispariA[i]=A[2i+1];
  FFT_SEQUENZIALE(pariA, pariB, n/2);
  FFT_SEQUENZIALE(dispariA, dispariB, n/2);
  for i=0 to n/2-1 do
    B[i]=pariB[i]+w1·dispariB[i];
    B[i+n/2]=pariB[i]-w1·dispariB[i];
```

FFT su butterfly. P=n·(log(n)+1) e T=O(log n).

```
FFT_BUTTERFLY(A0..An-1,B0..Bn-1:num; n,isign:int);
  forall i,j where i=0,0≤j≤n-1 pardo
    Bi,j=Ai,j;
  for p=0 to log(n)-1 do
    forall i,j where i=p+1,0≤j≤n-1 pardo
      k=log(n)-p;
      Xi,j=Bi-1,j;
      Yi,j=Bi-1,COMPL(j,k);
      if BIT(j,k)==1   then SWAP(Xi,j, Yi,j);
      Bi,j=Xi,j+OMEGA(p,j,log(n),isign)*Yi,j;
```

FFT su ipercubo. P=O(n) e T=O(log n).

```
FFT_IPERCUBO(A0..An-1,B0..Bn-1:num; n,isign:int);
  forall j where 0≤j≤n-1 pardo
    Bj=Aj;
  for p=0 to log(n)-1 do
    forall j where 0≤j≤n-1 pardo
      k=log(n)-p;
      Xj=Bj;
      Yj=BCOMPL(j,k);
      if BIT(j,k)==1   then SWAP(Xj,Yj);
      Bj=Xj+OMEGA(i,j,log(n),isign)*Yj;
```

COMPITI D'ESAME: *PRAM*

Es. 1 – 18 Gennaio 2006

Dato un insieme I di n intervalli chiusi della retta calcolare il numero massimo di intervalli che si intersecano tra loro (estremi tutti distinti).
P=O(n²) e T=O(log n). EREW.

```
MAX_INTERSEZ_SU_RETTA(s,d:vect; n:int):int;
  forall i where 1≤i≤n pardo
    REPLICA( sb[i,*], s[i], n);
    REPLICA( db[i,*], d[i], n);
    REPLICA( sc[*,i], s[i], n);
    REPLICA( dc[*,i], d[i], n);
  forall i,j where 1≤i,j≤n pardo
    if ( (sb[i,j]<sc[i,j] and sc[i,j]<db[i,j]) or
         (sb[i,j]<dc[i,j] and dc[i,j]<db[i,j]) or
         (sc[i,j]<sb[i,j] and sb[i,j]<dc[i,j]) or
         (sc[i,j]<db[i,j] and db[i,j]<dc[i,j]) )
      then v[i,j]=1;
    else v[i,j]=0;
  forall j where 1≤j≤n pardo
    p[j]=SOMMATORIA(v[*,j], n);
  MAX_INTERSEZ_SU_RETTA=MAX(p, n);
```

Es. 1 – 8 Febbraio 2006

Ordinare una sequenza S di interi distinti mettendo prima tutti i dispari e poi tutti i pari mantenendo l'ordine iniziale.
P=O(n/log n) e T=O(log n). EREW.

```
ORDINAMENTO_ODD_EVEN(s:vect; n:int);
  forall j where 1≤j≤n/log(n) pardo
    for k=1 to log(n) do
      h=j·log(n)+k;
      tmp[h]=s[h];
      b[n+h-1]=p[h]=EVEN(s[h]);
  SOMME_PREFISSE2(b, c, n);
  REPLICA(numOdd, n-c[2n-1], n/log(n));
  forall j where 1≤j≤n/log(n) pardo
    for k=1 to log(n) do
      h=j·log(n)+k;
      if p[h]==1
        then s[ c[n+h-1]+numOdd[j] ]=tmp[h];
        else s[ n-c[n+h-1] ]=tmp[h];
```

Es. 1 – 12 Luglio 2005

Dati n punti distinti del piano, dire quanti distano al più k dal punto p1.
P=O(n/log n) e T=O(log n). EREW.

```
KNN(x,y:vect; k,n:int):int;
  forall i where 1≤i≤n pardo
    REPLICA( px, x[1], n/log(n));
    REPLICA( py, y[1], n/log(n));
    REPLICA( vk, k,    n/log(n));
  forall j where 1≤j≤n/log(n)-1 pardo
    t=j+n/log(n)-1;
    sum[t]=0;
    for k=1 to log(n) do
      h=j·log(n)+k;
      if ( (SQRT((x[h]-px[j])² +
                 (y[h]-py[j])² ) < vk[j]) and
                                   h!=1
        then   sum[t]=sum[t]+1;
  KNN=SOMMATORIA(sum, n/log(n));
```

Es. 1 – 14 Giugno 2005

Dati n punti del piano trovare la coppia di intervalli più distanti tra loro.
P=O(n²) e T=O(log n). EREW.

```
MAX_DIST_PUNTI(x,y:vect; res1,res2,n:int);
  forall i where 1≤i≤n pardo
    REPLICA( bx[i,*], x[i], n);
    REPLICA( cx[*,i], x[i], n);
    REPLICA( by[i,*], y[i], n);
    REPLICA( cy[*,i], y[i], n);
  forall i,j where 1≤i,j≤n pardo
    d[i,j]=SQRT( (bx[i,j]-cx[i,j])² +
                 (by[i,j]-cy[i,j])² ) );
  forall j where 1≤j≤n pardo
    MAX_POS(d[*,j], colMax[j], rowsIdx[j], n);
  MAX_POS(colMax, tmp, colIdx, n);
  res1=colIdx;
  res2=rowsIdx[colIdx];
```

Calcolo della posizione del max e del max in un vettore a[n..2n-1].
P=O(n) e T=O(log n). EREW.

```
MAX_POS(a:vect; max,pos,n:int);
  forall i where 1≤j≤n pardo
    b[n+j-1]=j;
  for k=log(n)-1 downto 0 do
    forall j where 2^k≤j≤2^(k+1)-1 pardo
      if a[2j]>a[2j+1]
        then b[j]=b[2j];
             a[j]=a[2j];
      else b[j]=b[2j+1];
           a[j]=a[2j+1];
  max=a[1];
  pos=b[1];
```

Es. 1 – 1 Giugno 2004

Dati n intervalli della retta trovare la coppia più distante tra loro.
P=O(n²) e T=O(log n). EREW.

```
MAX_DIST_INTERV(s,d:vect; res1,res2,n:int);
  forall i where 1≤i≤n pardo
    REPLICA( sb[i,*], s[i], n);
    REPLICA( db[i,*], d[i], n);
    REPLICA( sc[*,i], s[i], n);
    REPLICA( dc[*,i], d[i], n);
  forall i,j where 1≤i,j≤n pardo
    if ((sb[i,j]<sc[i,j] & sc[i,j]<db[i,j]) or
        (sb[i,j]<dc[i,j] & dc[i,j]<db[i,j]) or
        (sc[i,j]<sb[i,j] & sb[i,j]<dc[i,j]) or
        (sc[i,j]<db[i,j] & db[i,j]<dc[i,j]) )
      then d[i,j]=0;
    else if sb[i,j]<sc[i,j]
      then d[i,j]=sc[i,j]-db[i,j];
      else d[i,j]=sb[i,j]-dc[i,j];
  forall j where 1≤j≤n pardo
    MAX_POS(d[*,j], colMax[j], rowsIdx[j], n);
  MAX_POS(colMax, tmp, colIdx, n);
  resX=colIdx;
  resY=rowsIdx[colIdx];
```

CAPITOLO 31: *Algoritmi concorrenti*

Sommatoria concorrente di n el.ti $a_1..a_n$ con p processori $p_1..p_p$, p≤n processori, ogni proc. Lavora con n/p el.ti. T=O(n/p + p).

```
procedure SOMMATORIA_CONC(A1..An):num;
  varGlobale=[sum|or|xor=0, and=1, min|max=±∞];
  forall i where 1≤i≤p concdo   SOMMA(i);
  // Solo per XOR:
  // if ODD(varGlobale)   then XOR=1;
  //                      else XOR=0;
  // Solo per conteggio 0-1:
  // if varGlobale > n/2   then CONTA=1;
  //                       else CONTA=0;
  SOMMATORIA_CONC=varGlobale;

process SOMMA(i);
  varLocale=[sum|or|xor=0, and=1, min|max=±∞];
  for j=i to n by p do
    varLocale=SUM(varLocale, a[j]);
    // per xor si usa sempre SUM()
  lock(varGlobale);
    varGlobale=SUM(varGlobale, varLocale);
  unlock(varGlobale);
```

Somme prefisse concorrente di n el.ti $a_1..a_n$, Array di appoggio $b_1..b_{n/p}$. T=O(n/p + p).

```
procedure SOMME_PREFISSE_CONC(A1..An);
  forall i where 1≤i≤p concdo   SOMME_PREF(i);
  for i=2 to p do   b[i]=b[i]+b[i-1];
  forall i where 2≤i≤p concdo   RISOMMA(i);

process SOMME_PREF(i)
  for j=(i-1)·p+2 to i·p do
    a[j]=a[j]+a[j-1];
  b[i]=a[i·p];

process RISOMMA(i)
  for j=(i-1)·p+1 to i·p do
    a[j]=a[j]+b[i-1];
```

Data una sequenza ordinata di interi $S_1..S_n$ (compresi fra 0 ed n) individuare l'unico intero mancante. T=O(n/p).

```
function INT_ASSENTE_ORDIN(S1..Sn):int;
  forall i where 1≤i≤p concdo   CALC_ASS(i);
  INT_ASSENTE_ORDIN=assGlobale;

process CALC_ASS(i);
  if (i=1 and S[i]!=0)
    then assGlobale=0;
  else if (i=n and S[i]!=n)
    then assGlobale=n;
  else if (i<n and S[i]+1!=S[i+1])
    then assGlobale=S[i]+1;
```

Data una sequenza non ordinata di interi $S_1..S_n$ (compresi fra 0 ed n) individuare l'unico intero mancante. T=O(n/p).

```
function INT_ASSENTE_NON_ORDIN(S1..Sn):int;
  sum=SOMMATORIA_CONC(S, n);
  INT_ASSENTE_NON_ORDIN=(n·(n+1)/2) - sum;
```

*Moltiplicazione di 2 matrici n*n. SPEEDUP=O($(n^2+n)/p$).*

```
procedure MOLT_MATRICI(A, B: matrix; n:int);
  forall i where 1≤i≤p concdo   MOLT(i);
  INT_ASSENTE_ORDIN=assGlobale;

process MOLT(i);
  for j=i to n^2 by p do
    result=0;
    row=((j-1) mod n)+1;
    col=((j-1) div n)+1;
    for k=1 to n do
      result=result+a[row,k]·b[k,col];
    lock(c); // c globale
      c[row,col]=result;
    unlock(c);
```

CAPITOLO 32: *Algoritmi distribuiti*

Colorazione di un anello unidirezionale. Solo un nodo per volta può iniziare la computazione. MSG=O(n) e T=O(n).

```
procedure START_COL_UNIDIR(...);
  colorato=true;
  myColor=nero;
  sendRight(colorazione, bianco);

when received(colorazione, colore) do
  if colorato=false   then
    myColor=colore;
    colorato=true;
    if colore=bianco   then
      sendRight(colorazione, nero);
    else
      sendRight(colorazione, bianco);
  else
    if myColor!=colore   then myColor=red;
```

Colorazione di un anello (bi/uni)direzionale. Qualsiasi nodo può iniziare la computazione, anche insieme. MSG=O(n^2) e T=O(n).

```
procedure START_COLORAZIONE2(...);
  sendRight(colorazione, myID, 1);
  partecipante=true;

when received(colorazione, ID, n) do
  if ID>myID   then
    sendRight(colorazione, ID, n+1);
    partecipante=false;
  if (ID<myID and partecipante=false) then
    sendRight(colorazione, myID, 1);
    partecipante=true;
  if ID==myID   then
    colore=bianco;
    partecipante=false;
    sendRight(colora, nero, n, 1);

when received(colora, col, numTot, n) do
  n=n+1;
  partecipante=false;
  if numTot==n   then
    if EVEN(numTot)   then colore=col;
    else colore=red;
  else
    colore=col;
    if colore==bianco   then
      sendRight(colora, nero, numTot, n);
    else
      sendRight(colora, bianco, numTot, n);
```

Colorazione di un anello bidirezionale. Solo un nodo per volta può iniziare la computazione. MSG=O(n) e T=O(n).

```
procedure START_COL_BIDIR(...);
  colorato=true;
  myColor=nero;
  sendLeft (colorazione, true,  bianco);
  sendRight(colorazione, false, bianco);

when received(colorazione, verso, colore) do
  if colorato=false   then
    myColor=colore;
    colorato=true;
    if colore=bianco   then
      pass(colorazione, verso, nero);
    else
      pass(colorazione, verso, bianco);
  else
    if (myColor!=colore and verso)  then
      myColor=red;
```

Elezione del leader in un albero radicato. MSG=O(n) e T=O(n).

```
procedure ELEZIONE_LEADER_ALBERO(...);
  if padre==null   then
    sendFigli(nuovoLeader, myID);
    leader=myID;
  else
    elezIndetta=true;
    sendPadre(richiestaLeader);

when received(richiestaLeader) do
  if padre==null  then
    elezIndetta=false;
    leader=myID;
    sendFigli(nuovoLeader, myID);
  else if elezIndetta==false   then
    sendPadre(richiestaLeader);
    elezIndetta=true;

when received(nuovoLeader, ID) do
  leader=ID;
  elezIndetta=false;
  sendFigli(nuovoLeader, ID);
```

Sommatoria in un albero radicato. Ogni nodo può iniziare la Computazione, anche insieme. Risultato in tutti i nodi. Numero ottimo di messaggi. MSG=O(n) e T=O(n).

```
procedure INDICI_SOMMATORIA_ALBERO(...);
  if padre==null   then
    sendFigli(sommatoria);
  else
    sommatoriaIndetta=true;
    sendPadre(indiciSommatoria);

when received(sommatoria) do
  if numFigli==0   then
    sendPadre(somma, myValue);
  else
    sendFigli(sommatoria);

when received(somma, val) do
  numValues=numValues+1; //iniz a 0
  mySum=SOMMA(mySum, val); //iniz a 0
  if (numValues=numFigli and padre!=null)
    then sendPadre(somma, mySum+myValue);
         numValues=0;
         mySum=0;
  if (numValues==numFigli and padre==null)
    then sommaTot=SOMMA(mySum, myValue);
         numValues=0;
         mySum=0;
         sendFigli(risultato, sommaTot);

when received(risultato, val) do
  sommaTot=val;
  sendFigli(risultato, val);
```