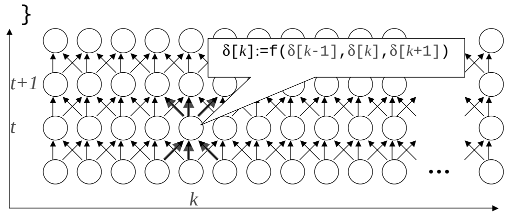
Data Distributions

Distribution Functions
Distributions with Redundancies
Redistribution as Configuration Problem

Example

```
for (t=1;t<T;t++) {
  forall (k=1;i<D;k++) in parallel {
    δt+1[k]= f(δt[k -1],δt[k],δt[k +1])
  }
}
```



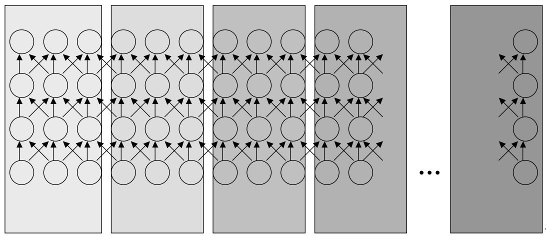
Mapping to distributed systems

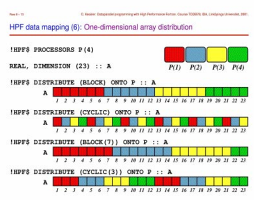
- Two principle approaches:

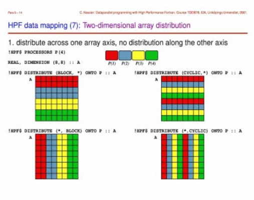
1. Distribute the data
   - Each array element δ[k] is assigned to a processor
   - Computation follows:
     - All computations defining (writing) δ^{t+1}[k]=… , t=1..T are executed on the processor δ[k] is assigned to
     - Owner computes rule
2. Schedule the computations
   - Each computation is scheduled individually
   - Consumed array elements δ^{t}[k-1] δ^{t}[k] δ^{t}[k+1] and produced array elements δ^{t+1}[k] become local variables of the tasks

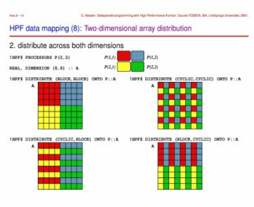
Distribute the Data

Distribution d is a function: a $\rightarrow \{1 \ldots P\}$







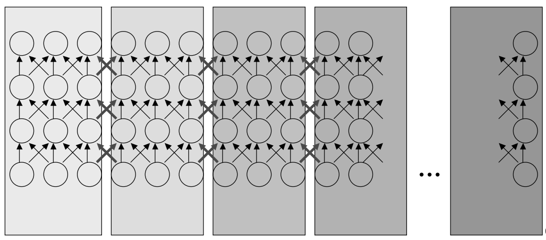


Blockwise:
d(a[i])=i div n/P
Cyclic:
d(a[i])=i mod P
Block-cyclic:
d(a[i])=(i div c) mod P

Generalized to more dimensions

Good Data Distribution

- Compare execution costs in cost model
- In general: compute locally, avoid communications synchronization



Problem I: Alignment

Example Vector Product:

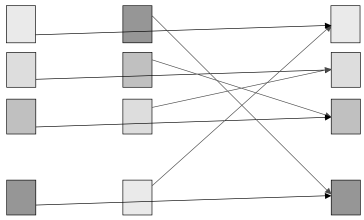
$y = a \bullet x$

$y_i = a_{i,} x_i$

```
forall (i=0;i<n;i++) in parallel{
      y[i]=a[i]*x[i]
  }
}
```

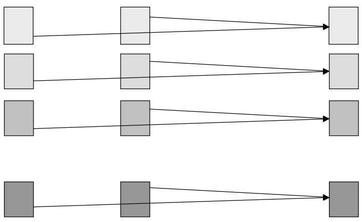

Align distribution of arrays

Vector a distributed blockwise | Vector x distributed blockwise, not aligned to a | Vector y then computed with communication



Align distribution of arrays

Vector a distributed blockwise | Vector x distributed blockwise, aligned to a | Vector y (aligned to a,x) then computed without any communication



Problem II: Redundancy

Example: Matrix–Vector Multiplication:

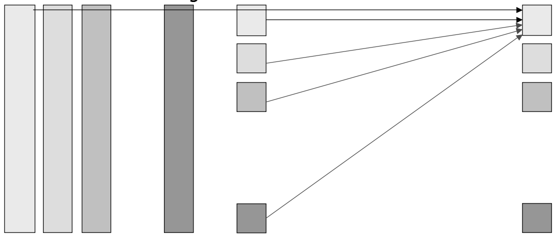
$y = A\, x$

$y_i = a_{i,1}\, x_{\,1} + \ldots + a_{i,n}\, x_n$

```
forall (i=0;i<n;i++) in parallel{
  y[i]=0;
  for (j=0;j<n;j++) {
      y[i]=y[i] + a[i,j]*x[j]
  }
}
```

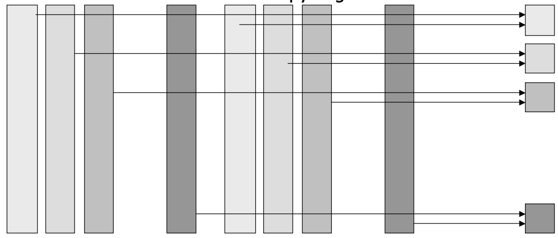

Distribution function for arrays

Matrix A distributed blockwise | Vector x distributed however aligned to A however | Vector y always required communication



Redundant Distribution

Matrix A distributed blockwise | Vector x redundantly distributed, each copy aligned to A | Vector y required no communication



Redundancy

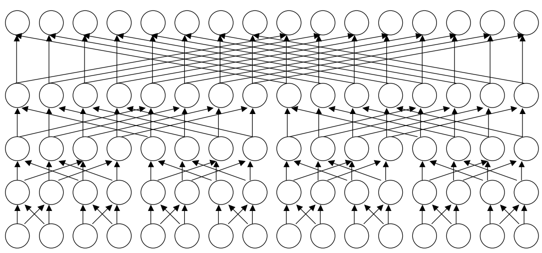
- Cannot be expressed by a function
  - Relation between array elements and processors
  - If derived automatically, larger solution space
- Could save communication
  - Costs local computation time, e.g. due to caching effects on processors
  - Could only be biased by a more elaborated cost model including memory hierarchies
  - Too expensive to optimize for

Problem III: Changing Alignment

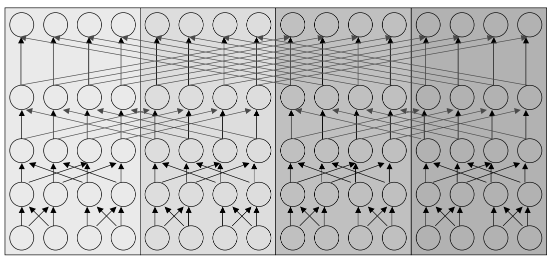
- Example: FFT (with ω=n-th unity root)

```
forall (i=0;i<n;i++) in parallel
  x[i]=x[r(i)];
for (i=0;i<log(n);i++) {
  forall (j=0;j<n;j++) in parallel{
    if (j mod pow(2,i) < pow(2,i-1))
       x[j]=x[j] + ω^j_2i * x[j+pow(2,i-1)]);
    else
       x[j]=x[j-pow(2,i-1)] + ω^j_2i * x[j]
  }
}
```

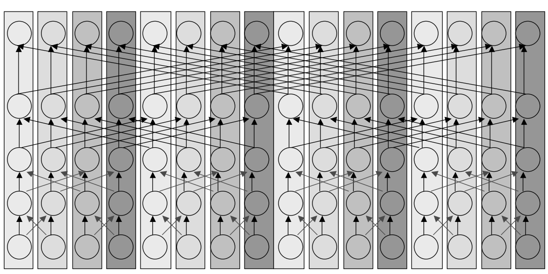

FFT Dependency Graph n=16



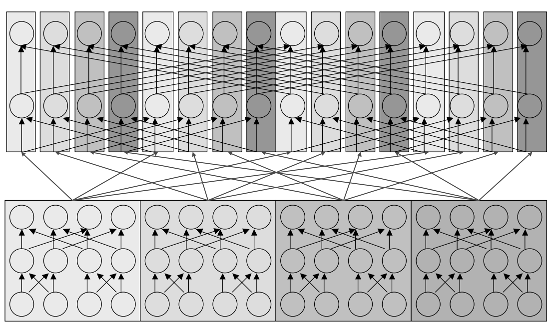
Block distribution P=4



Cyclic distribution P=4



Redistribution (4-relation)



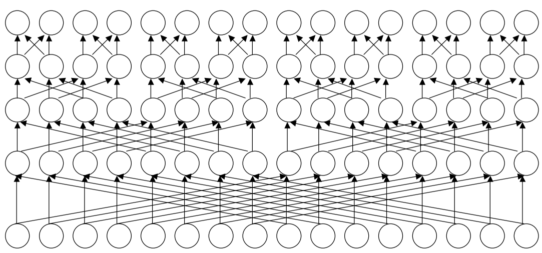
Redistributions

- Cannot be expressed by relation between array elements and processors
  - Requires relation between array elements, iteration vectors (time axis) and processors
  - If derived automatically, larger solution space
- Could save communication
  - Sometimes it only bundles communication
  - Could only be biased by a more elaborated cost model including communication parameters as functions on the message size

Problem IV: Composition

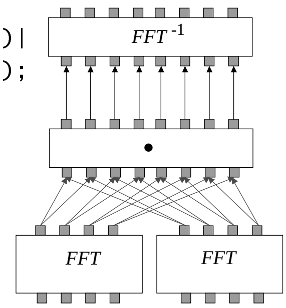
- Example: Polynom Multiplication:
  $p(x)=a_0+a_1x + \dots a_{n-1}x^{n-1}$,
  $q(x)=b_0+b_1x + \dots b_{m-1}x^{m-1}$
  $p(x)q(x)=c_0+c_1x + \dots c_{n+m-2}x^{n+m-1}$,
  $c_k=\Sigma_{j\in[0,k]}\dots a_j b_{k-j}$, $k\in[0,n+m-2]$
- Computation by:
  $c=FFT^{-1}(FFT(a') \bullet FFT(b'))$
  $a_i=\Sigma_{j\in[0,k]}\dots a_j b_{k-j}$, $k\in[0,n+m-2]$

FFT^{-1} Dependency Graph n=16



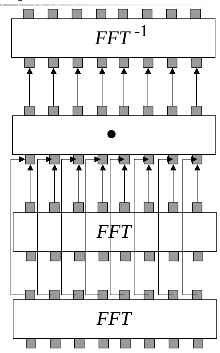
Implementation A

```
pardo
  a''= FFT(a',P/2)|
  b''= FFT(b',P/2);
c'= •(a',b',P);
c = FFT-1(c',P)
```



Implementation (B)

```
a'= FFT(a,P);
b'= FFT(b,P);
c'= •(a',b',P);
c = FFT-1(c',P)
```



Composition

- Composition
  - Sequential `P,P'`: output distribution of `P` must conform to input distribution of `P'`
  - Parallel `P|P'`: distribution of `P` uses p processors, distribution of `P'` uses p' processors where $p+p'=P$
- Optimal local solutions could be suboptimal globally
- Solutions:
  - Perform redistribution
  - Find globally good distribution

Conclusion of Problems

- Find right alignments
- Introduce redundancy if applicable
- Redistribute within loops
- Compose according to constrains – redistribute/find globally good distributions

25

Approach

1. Propose distribution functions d for each individual parallel assignment
2. For each pair of distribution functions (d,d') for consecutive parallel assignments S,S'
   - Extend d such that (d,d') does not require communication (introduce redundancy)
   - Calculate redistribution costs for (d,d')
3. Find the global optimum configuration

26

1. Propose distribution functions

- Refers to distribution of computed array d_{out}
- Induces a distribution and alignment of the input array(s) d_{in} (eventually redundant)
- Arbitrary basis for proposals
  - Analyze dependencies
  - Analyze task graph
  - Propose usual suspects (e.g. block, cyclic, …)
  - Proposals of programmer (e.g. HPF directives)

27

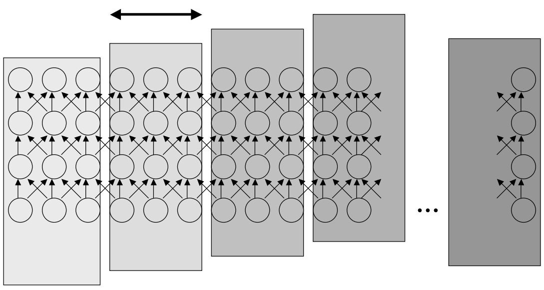
Example

```
for (t=1;t<T;t++) {
  forall (k=1;i<D;k++) in parallel {
     δ^(t+1)[k]= f(δ^t[k -1],δ^t[k],δ^t[k +1])
  }
}
```

- Assign: δ^{t+1}`[k]= f(`δ^t`[k-1],`δ^t`[k],`δ^t`[k+1])`
- Proposal block distribution:
  - d_{out}(δ`[k]`)=`k` div n/P
  - d_{in}(δ`[k]`)=`k` div n/P,
d_{in}(δ`[k-1]`)=`k` div n/P
d_{in}(δ`[k-1]`)=`k` div n/P
  - d_{in} is redundant

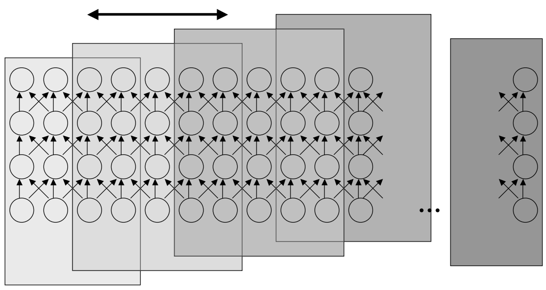
28

d_{out}



29

d_{in}



30

2. Consecutive assignments

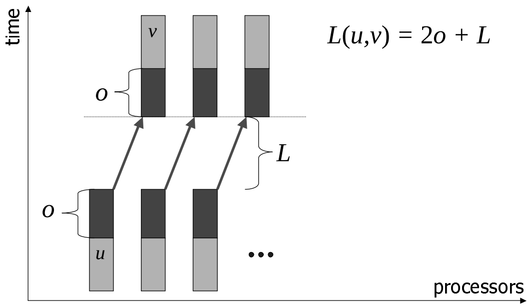
- Not uniquely defined at compile time
- Treat loops and branches conservatively
- No redundancy/redistribution costs for (d_{out}, d'_{in}) iff $d'_{in} \leq d_{out}$
  - Note that d_{out}, d'_{in} are relations, i.e. sets of pairs
  - $\leq$ defined to be partial subset order relation $\subseteq$
- Several iterations possible

Example

```
for (t=1;t<T;t++) {
  forall (k=1;i<D;k++) in parallel {
    δt+1[k]= f(δt[k -1],δt[k],δt[k +1])
  }
}
```

- New redundant distribution $d'_{out} = d_{in}$ because of loop, compute d'_{in} accordingly
- Communication cost $L(d'_{out}, d_{in}) = 0$ by definition
- Redistribution costs $L(d_{out}, d_{in})$ $L(d_{out}, d'_{in})$ $L(d'_{out}, d'_{in})$ depend on cost model

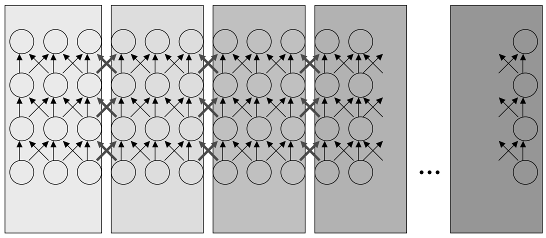
LogP Communication Step



Example with LogP as cost model

$L(d_{out}, d_{in}) = L(d'_{out}, d'_{in}) = 2(2o + L)$ (2 elements overlap)

$L(d_{out}, d'_{in}) = 4(2o + L)$ (4 elements overlap)

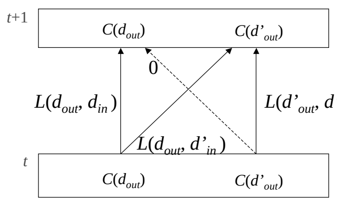


3. Global optimum configuration

- Idea for not iteratively composed programs:
  - Compute the basic block graph of the program
  - Assign computation costs for each $C(d_{out})$ to nodes (this may be different since redundant distributions require redundant computations)
  - Assign Redistribution costs $L(d_{out}, d_{in})$ to the edges
  - Find minimum path in graph
- *NP hard* problem in general (polynomial for `goto` free languages, ok in practice)

Example (unrolled assignments)

```
forall (k=1;i<D;k++) in parallel
   δt+1[k]= f(δt[k -1],δt[k],δt[k +1])
forall (k=1;i<D;k++) in parallel
   δt+2[k]= f(δt+1[k -1],δt+1[k],δt+1[k +1])
```

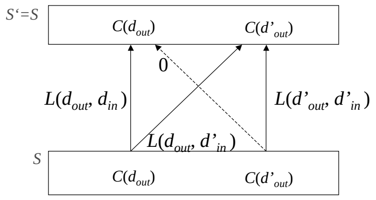


3. Iterations

- Brute force (exact)
  - Unroll loop
  - Treat as them before
  - Unavoidable, if costs depend on the iteration
- Approximation
  - Let S,S' be last and first loop statement, resp.
  - Assume an artificial redistribution $d_{out}(S)$, $d_{in}(S')$ also for the last iteration
  - Let S'' be the first statement after loop
  - Connect $d_{out}(S')$, $d_{in}(S'')$

Example

```
for (t=1;t<T;t++) {
  forall (k=1;i<D;k++) in parallel {
       δt+1[k]= f(δt[k -1],δt[k],δt[k +1])
  }
}
```



Conclusion

- Only approximations of the optimum
  - Compilers can give good results
  - Sometimes not good enough
  - Therefore programmers must be able to find better solutions for specific problems "by hand" (preferably "by head")
- Alternative to data distribution: task scheduling