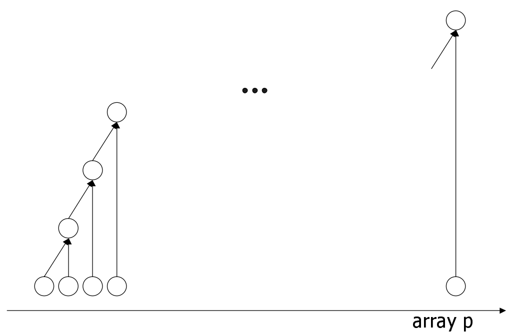
Compiling for Parallel Computers

Dependency Analysis
Parallelization
Loop Transformations

Sequential Prefix Sums

```
right[0…n]=0…n;
for (p=1;p<n;p++) {
  right[p]=right[p-1]+right[p];
}
```

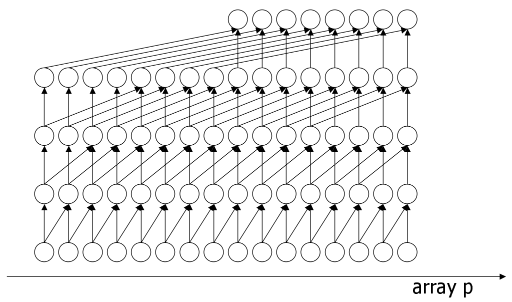

Sequential Data Dependencies



Weird Sequential Prefix Sums

```
right[0…n]=n;
for (i=1;i<n;i*=2) {
  for (p=0;p<n;p++) {
    if (p >= i)
      aux[p]=right[p-i]+right[p];
  }
  for (p=0;p<n;p++) {
    if (p >= i)
      right[p]=aux[p];
  }
}
```

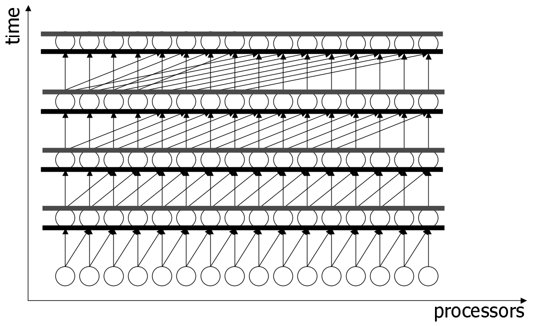

Weird Seq. Data Dependencies



PRAM Prefix Sums

```
right[0…n]=n;
for (i=1;i<n;i*=2) {
  forall (p=0;p<n;p++) in parallel{
    if (p >= i)
      right[p]=right[p-i]+right[p];
  }
}
```

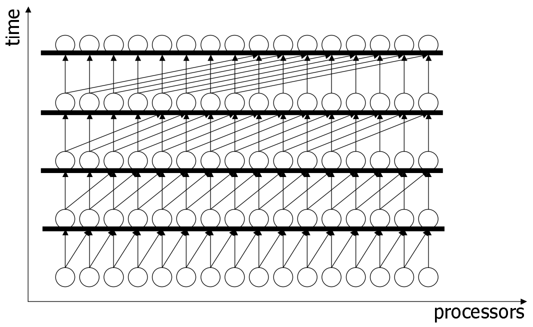
Execution of PRAM Prefix Sums



BSP Prefix Sums

```
for (p=0;p<n;p++) in parallel {
  right=p; left=0;
  for (i=1;i<n;i*=2) {
    if (p+i < n)
      put(p+i,right,left);
    barrier_synchronize();
    if (p >= i)
      right=right+left;
  }
}
```

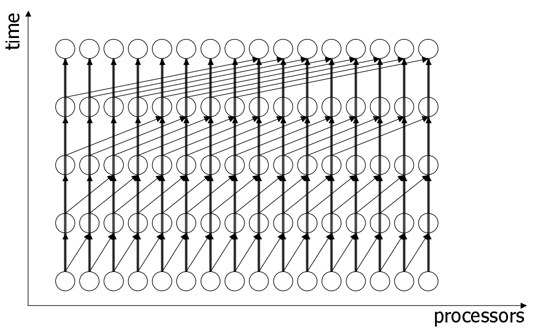

Execution of BSP Prefix Sums



LogP Prefix Sums

```
Process(p) {            //i∈[0…n-1]
  right=p; left=0;
  for (i=1;i<n;i*=2) {
    if (p+i < n)
      send(p+i,right);
    if (p >= i) {
      left=receive(p-i);
      right=right+left;
    }
  }
}
```


Execution of LogP Prefix Sums



Observations

- Dependencies of operation due to programming language semantics
- Data dependencies of operations
- Former is more restrictive than latter
- Questions:
  - How can we analyze data dependencies
  - How can we relax dependencies induced by programming language automatically
    - How to derive a PRAM program from a sequential
    - How to derive a BSP program from a PRAM
    - How to derive a LogP program from a BSP

How can we analyze data dependencies?

Definitions
Brute force method
Dependency analyses

Data Dependencies

- Between operations defining and using variables
  - $DEF(o)=\{v \mid v{:=}o(v_1 \dots v_n)\}$
  - $USE(o)=\{v_i \mid v{:=}o(v_1 \dots v_n),\ i \in [1 \dots n]\}$
- True dependencies
  - Operation uses variable defined by another
  - $true_v(o_1, o_2) \Leftrightarrow \exists v \in DEF(o_1) \cap USE(o_2)$
- Anti dependencies
  - Operation (re-)defines variable used by another
  - $anti_v(o_1, o_2) \Leftrightarrow \exists v \in USE(o_1) \cap DEF(o_2)$
- Output dependencies
  - Operation defines variable also defined by another
  - $output_v(o_1, o_2) \Leftrightarrow \exists v \in DEF(o_1) \cap DEF(o_2)$

Execution order

- Total order (sequential languages)
  Partial order (parallel languages)
  - Denoted with $\lhd$
  - Defined by programming language semantics
  - Over operations of a program run
- Assume
  - Operations defining and using variables
  - Nested loops over those operations
- Operations of a program run and $\lhd$ defined by the vectors of loop indices

Vectors of loop indices (Example)

```
right[0…n]=n;
for (i=1;i<n;i*=2) {
  for (p=0;p<n;p++) {
    if (p >= i)
      aux[p]=right[p-i]+(i,p)right[p];
  }
  …
}
```

i∈ [0…n], p∈ [0…n]

$+_{(i1,p1)} \rightarrow +_{(i2,p2)} \Leftrightarrow i1<i2 \lor i1=i2 \land p1<p2$

Vectors of loop indices (general)

- Define: vector of loop indices:
  - index vector i of an operation o
  - element of the vector space $I(o)$ defined by the Cartesian product of ranges of index variables of enclosing loops L_1 L_2 … $L_{\text{loop depth}}$ of operation o
  - Order from outer to innermost loop
  - In example: i∈ [0…n], p∈ [0…n], i.e. index vector of + is an element of $[0…n]^2$
- Define: maximum common loop index of two operations $max(o_1, o_2)$ ::= highest index of a loop enclosing both operations
- Define: syntactical order of two operations $pred(o_1, o_2)$ ::= o_1 occurs before o_2 in program text

Execution order (general)

- Let o be a (syntactical) operation of a program: each execution of o in a run of the program is uniquely defined by $i \in I(o)$, denoted by o_i
- Let < be the lexicographical order of integer vectors
- Let o_i, o_j' be two operation executions:
  $o_i \lhd o_j' \Leftrightarrow$
  $i[1 \dots max(o, o')] < j[1 \dots max(o, o')] \lor$
  $i[1 \dots max(o, o')] = j[1 \dots max(o, o')] \land pred(o,o')$
- Corollary: Let o_i, o_j be two operation executions of the same (syntactical) operation o: $o_i \lhd o_j \Leftrightarrow i < j$

Dependence

- Relation δ over operations of a program run
- $o_i \,\delta\, o_j' \Leftrightarrow$
  program ordered: $o_i \lhd o_j'$ $\wedge$

  data dependent: $true_v(o_i, o_j') \vee$
  $anti_v(o_i, o_j') \vee$
  $output_v(o_i, o_j')$ $\wedge$

  not covered $\nexists\, o_k'' : o_i \lhd o_k'' \lhd o_j' \wedge$
  $v \in DEF(o_k'')$

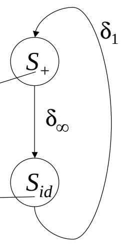
Distance and Loop Dependence

- Let o_i, o_j' be two operation executions $o_i \lhd o_j'$
  - Distance vector:
    $d(o_i, o_j') = j\,[1\ldots max\,(o, o')] - i\,[1\ldots max\,(o, o')]$
- Let o_i, o_j' be two operation executions $o_i \,\delta\, o_j' \Rightarrow$
  $o_i \lhd o_j' \Leftrightarrow$
  (i) $i[1\ldots max(o, o')] < j[1\ldots max(o, o')] \vee$
  (ii) $i[1\ldots max(o, o')] = j[1\ldots max(o, o')] \wedge pred(o, o')$
- (i) $\Rightarrow \exists\, level$: $d(o_i, o_j')[level] > 0 \wedge d(o_i, o_j')\,[0\ldots0_{level}]$
  Loop carried dependency, carried at *level*
- (ii) $\Rightarrow d(o_i, o_j') = [0\ldots0_{max(o,\,o')}]$
  Loop independent dependency

Dependence Graph

- Dependence Graph $DG=(N,E)$ directed graph
  - Any (syntactical) operation $o \Leftrightarrow S_o \in N$
  - $o_i \,\delta\, o_j' \Leftrightarrow (S_o, S_{o'}) \in E$

```
right[0…n]=n;
for (i=1;i<n;i*=2) {
  for (p=0;p<n;p++) {
    if (p >= i)
      aux[p]=right[p-i]+right[p];
  }
  for (p=0;p<n;p++) {
    if (p >= i)
      right[p]=aux[p];
  }
}
```



Dependence

- How to find out all dependences in the program?
- Exactly:
  - Not decidable in general, expensive anyway
  - All independent operations can be executed in parallel, sequentialization only for optimization
- Conservatively:
  - Efficient
  - Some non essential dependencies remain an obstacle for parallelization

Brute Force Method

- Check if the program is oblivious
  - Dependences are not data dependent
  - Sufficient condition
- Unroll the program and consider the operations tasks of a task graph

Check if the program is oblivious

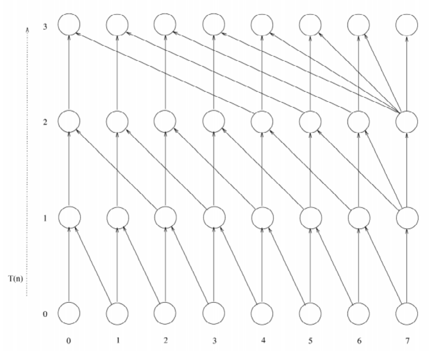
- Oblivious ::=
  Data dependencies do not depend on the input data but only on the input data size
- Sufficient
  - No indirect (data dependent) memory access
  - No `while` loops (with data dependent condition)
  - No `if` statements (with data dependent condition)

Non Oblivious: Pointer Jumping

```
p[0…n]=//some initialization;
for (j=1;j<n;j*=2) {
  forall (i=0;i<n;i++) in parallel{
    p[i]=p[p[i]];
  }
}
```

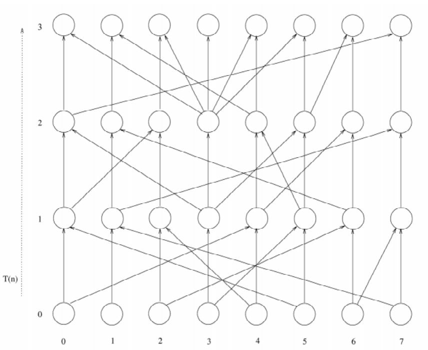
Indirect
(data dependent)
addressing

A Data Dependence



Init:
p[0]=1
p[1]=2
p[2]=3
p[3]=4
p[4]=5
p[5]=6
p[6]=7
p[7]=7

Another Data Dependence

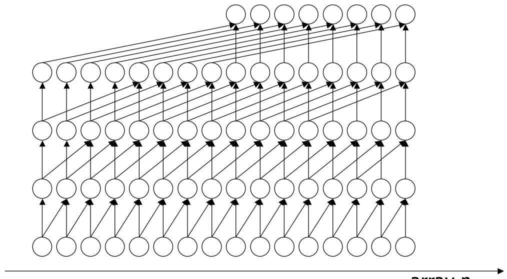


Init:
p[0]=5
p[1]=7
p[2]=4
p[3]=3
p[4]=0
p[5]=3
p[6]=2
p[7]=6

Oblivious: Sequential Prefix Sums

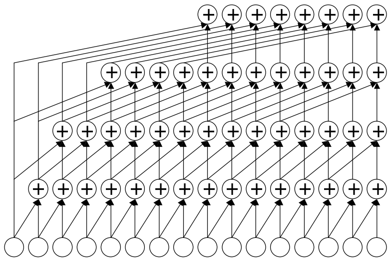
```
right[0…n]=0…n;
for (i=1;i<n;i*=2) {
  for (p=0;p<n;p++) {
    if (p >= i)
      right[p]=right[p-i]+right[p];
  }
  for (p=0;p<n;p++) {
    if (p >= i)
      right[p]=aux[p];
  }
}
```


Same Data Dependencies



array p

Task Graph



Translation (sketched)

- Task graph defines a maximum parallel program
- Correct translation trivial:
  - Compute task graph by unrolling the oblivious program
  - Each tasks gets a process sending and receiving along the (completely known) dependences
- All (deadlock free) sequentializations of that program with more conservative program dependencies are correct translations, too

Oblivious: PRAM Prefix Sums

```
right[0…n]=0…n;
for (i=1;i<n;i*=2) {
  forall (p=0;p<n;p++) in parallel{
    if (p >= i)
      right[p]=right[p-i]+right[p];
  }
}
```


Oblivious: BSP Prefix Sums

```
for (p=0;p<n;p++) in parallel {
  right=p; left=0;
  for (i=1;i<n;i*=2) {
    if (p+i < n)
      put(p+i,right,left);
    barrier_synchronize();
    if (p >= i)
      right=right+left;
  }
}
```

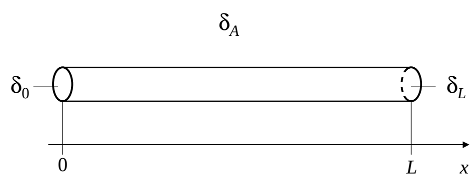

Oblivious: LogP Prefix Sums

```
Process(p) {           //i∈[0…n-1]
  right=p; left=0;
  for (i=1;i<n;i*=2) {
    if (p+i < n)
      send(p+i,right);
    if (p >= i) {
      left=receive(p-i);
      right=right+left;
    }
  }
}
```


Examples

- Non-Oblivious Program are many algorithms
  - on Graphs
  - on Lists
  - sorting by divide and conquer
- Oblivious Program
  - Sorting networks
  - Signal transformation algorithms (e.g. FFT)
  - Matrix algorithms (multiplication, potentiation)
  - Sum, Maximum, Prefix Sum,
  - General linear recurrences
  - Gauss-Elimination

Example: Temperature Simulation



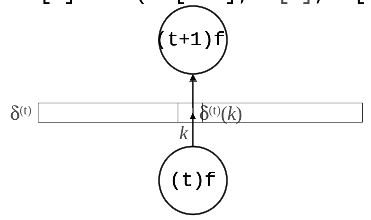
Problem	Temperature Simulation
Problem model	$\delta'' - \alpha^2(\delta - \delta_A) = 0$ $\delta(0) = \delta_0 \; \delta(L) = \delta_L$
Numeric Recipe	$\delta^{(t+1)}(x) = (1-\omega)\,\delta^{(t)}(x_k) +$ $\omega(-2/\Delta^2 - \alpha^2)^{-1} \times$ $(-\alpha^2\delta_A - \delta^{(t)}(x_{k-1})/\Delta^2 - \delta^{(t)}(x_{k+1})/\Delta^2)$

(Data parallel) program	`for t :=1..s do`
	 `for k :=1..n-2 do`
	 `δ[k]:=f(δ[k-1],δ[k],δ[k+1])`

Control parallel program	Translate - then optimize by clustering/scheduling

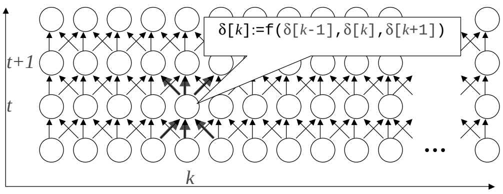
Dependencies

```
(t)    δ(t)[k] := f(δ(t-1)[k-1],δ(t-1)[k],δ(t-1)[k+1])
(t+1)  δ(t+1)[k] := f(δ(t)[k-1],δ(t)[k],δ(t)[k+1])
```

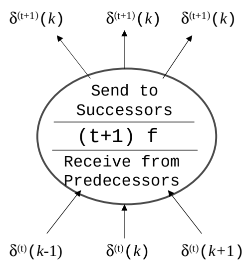


Task Graph

Computable for oblivious program

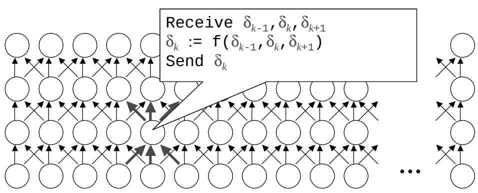


Processes



Process System

Start all processes in parallel



Iterative Oblivious Program

- Iterative Oblivious Program ::= Iteration (with data dependent condition) over
  - oblivious and
  - iterative oblivious programs
- Examples
  - Jacobi-iteration,
  - CG methods
  - Finite-Elements methods
  - Adaptive multi-grid methods

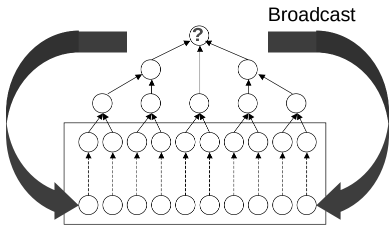
Translation (sketched)

- Loop over an oblivious program:
  - Broadcast to initialize the body
  - Translation of oblivious bodies as before
  - Inverse broadcast to collect the result
  - Decide on the termination
  - Broadcast stop or continue

  (actually barrier synchronize the loop)
- Loop over an iterative oblivious program similar

43

Program Execution



44

Situation revisited

- Brute force ok for
  - Oblivious, Iterative oblivious programs
  - Repeated same computations with different data
  - Then one step from sequential/PRAM program to LogP
- Problems if
  - Only one (or few) computations
  - Task graphs become too large (remember size of the task graph is in the order of the programs work)
  - Smarter solutions for these cases?
  - Unfortunately: good solution only for oblivious programs

45

Dependency Analysis

- Analyze dependencies without unrolling the program
- Lower bound for translation
  - With unrolling: work of the program (execution)
  - Without unrolling: size of program (text)

46

Definition revisited

- Relation δ over operations of a program run
- $o_i \, \delta \, o_j' \Leftrightarrow$

  program ordered: $o_i \lhd o_j' \quad \wedge$

  data dependent: $true_v(o_i, o_j') \vee anti_v(o_i, o_j') \vee output_v(o_i, o_j') \quad \wedge$

  not covered: $\nexists\, o_k'' : o_i \lhd o_k'' \lhd o_j' \wedge v \in DEF(o_k'')$

47

Central question

- Are the operations data dependent:
  $true_v(o_i, o_j') \vee anti_v(o_i, o_j') \vee output_v(o_i, o_j')$
- No need to distinguish kind ($true$, $anti$, $output$)
  - Simple question: are potentially the same variables accessed by o_i, o_j'
  - Kind of access (reed or write) trivially determines then kind of dependency $true$, $anti$, $output$
- Note: variables are array cells, i.e. even o_i, o_j access different variables

48

Example

```
for (i=10;i<20;i++) {
   a[2*i-1] = o(…);
   …
   … = o'(a[4*i-7]);
}
```

Questions:

- Is there in any iteration an array cell written by o and read by o' – *true, loop independent*
- Is there a particular iteration where o writes a cell that is read by o' in another (later in program order) iteration – *true, loop carried*
- Is there a iteration where o' reads a cell that is redefined by o in another (later in program order) iteration – *anti , loop carried*

Approach

- Assume (for a first try) restricted oblivious programs:
  - linear index functions
  - constant index bound
  - no conditionals and whiles

1. Is there any dependency possible without regarding the index bounds
2. If yes, is this dependency possible within the given bounds
3. If yes, are operations in the right order ⊲
   (Ignore coverage = be conservative)

Is there any dependency

- *o, o'* access (read, write) to the same array
- *o* access with $a_0 + a*i$, *o'* access with $b_0 + b*i$,
- Loop independent dependency:
  is $a_0 + a*i = b_0 + b*i$ for some i?
- Loop carried dependency:
  is $a_0 + a*i = b_0 + b*j$ for some i,j?
- Extended for more multi-dimensional arrays: check for each dimension of the array individually.

Solutions

- Loop independent dependency:
  $a_0 + a\, i = b_0 + b\, i$
- Solution of a linear equation:
  $i = (b_0 – a_0) / (a – b)$
- Loop carried dependency:
  $a_0 + a\, i = b_0 + b\, j$
  Solution of a linear Diophantine equation:
  $a\, i – b\, j = b_0 – a_0$

Linear Diophantine Equations

- Restricted form: $a\, x + b\, y = c$
- General form: $\sum_{1\leq i \leq n} a_i\, x_i = c$
- Solution criteria: $gcd(a_{1\leq i \leq n}) \mid c$
- Solution (restricted form, generalizeable)
  - Let $g=gcd(a_i)$ and u, v solution of $g = au + by$
  - Set of all solutions (x_t, y_t), $t \in \mathbb{Z}$ is given by:
    $x_t=uc/g + tb/g$
    $y_t=vc/g + ta/g$

Boundary Conditions

- Define: $z^+=(z>0)?z:0$ $\quad z^- =(z<0)?z:0$
- Boundaries are constants
- Applies to functions in $\mathbb{R}$
- Restricted form:
  - $\min\{ax + by \mid (x,y)\in [L_x:U_x, L_y:U_y]\} = a^+ L_x - a^- U_x + b^+ L_y - b^- U_y$
  - $\max\{ax + by \mid (x,y)\in [L_x:U_x, L_y:U_y]\} = a^+ U_x - a^- L_x + b^+ U_y - b^- L_y$
- General form:
  - $\min\{\sum_{1\leq i\leq n} a_i x_i \mid (x_1 \ldots x_n)\in [L_i:U_i]_{1\leq i \leq n}\} = \sum_{1\leq i\leq n}(a_i^+ L_i - a_i^- U_i)$
  - $\max\{\sum_{1\leq i\leq n} a_i x_i \mid (x_1 \ldots x_n)\in [L_i:U_i]_{1\leq i \leq n}\} = \sum_{1\leq i\leq n}(a_i^+ U_i - a_i^- L_i)$

Example (revisited)

```
for (i=10;i<20;i++) {
  a[2*i-1] = o(…);
  …
  … = o'(a[4*i-7]);
}
```

- Loop independent dependency: `2*i-1= 4*i-7`
  `i=3` (out of bounds)
- Loop carried dependency: `2*x-1 = 4*y-7`
  $\gcd(2, -4)=2 \mid -6$
  $2x - 4y = -6$, $10 \leq x, y < 20$: $\min(2x - 4y) = -60$, $\max(2x - 4y)=0$
  $(u,v)=(3,1)$, $(x,y)=(-9-2t,-3-t)$:
  true $10 \leq x<y < 20$: impossible
  anti $10 \leq y<x < 20$: $(x,y)=(17,10)$, $(x,y)=(19,11)$

55

General

- Conditions for dependencies based on the ideas presented
  - in specific cases exact
  - in general pessimistic (cannot disprove a dependence, assume it is there)
- All provable independent operations can be executed in parallel
- More details in
  Zima, Chapman: Compilers for Parallel and Vector Computers, ACM Press, 1990.

56

Relax dependencies induced by original program order

Parallelization
Loop Transformations

Example I
No loop-carried dependence

```
for (i=1;i<100;i++) {
  a[i] = b[i]*c[i]+d[i];
  b[i] = c[i]/d[i-1]+a[i];
}
```

```
forall (i=1;i<100;i++) in parallel{
  a[i] = b[i]*c[i]+d[i];
  b[i] = c[i]/d[i-1]+a[i];
}
```

58

Example I
No loop-carried dependence

```
forall (i=1;i<100;i++) in parallel{
  a[i] = b[i]*c[i]+d[i];
  b[i] = c[i]/d[i-1]+a[i];
}
```

```
Process(i) {        //i∈[1…100)
  a = b*c+d1;
  b = c/d2+a;
}
```

59

Example II
Loop-carried dependence

```
for (i=1;i<100;i++) {
  a[i] = b[i]*c[i]+d[i];
  b[i] = c[i]/d[i-1]+a[i-3];
}
```

```
forall (i=1;i<100;i++) in parallel{
  a[i] = b[i]*c[i]+d[i];
  barrier_synchronize();
  b[i] = c[i]/d[i-1]+a[i-3];
}
```

60

Example II

Loop-carried dependence

```
forall (i=1;i<100;i++) in parallel{
 a[i] = b[i]*c[i]+d[i];
 barrier_synchronize();
 b[i] = c[i]/d[i-1]+a[i-3];
}
```

```
Process(i) {          //i∈[1…100)
  a = b*c+d1;
  send(i+3,a);
  b = c/d2+receive(i-3);
}
```


Parallelization of loops I:

```
L: for (i=L;i<U;i++) {
  S1, … , Sn
}
```

If no loop-carried dependencies:

```
L: forall (i=L;i<U;i++) in parallel{
  S1, … , Sn
}
```


Parallelization of loops II:

```
L: for (i=L;i<U;i++) {
  S1, … , Sn
}
L': for (j=L';j<U';j++) {
  S'1, … , S'n
}
```

If no loop-carried nor interloop dependencies:

```
L: forall (i=L;i<U;i++) in parallel{
  S1, … , Sn
}
L': forall (j=L';j<U';j++) in parallel{
  S'1, … , S'n
}
```


Parallelization of loops III:

```
L: for (i=L;i<U;i++) {
  S1, … , Sn
}
L': for (j=L';j<U';j++) {
  S'1, … , S'n
}
```

If no loop-carried but interloop dependencies:

```
L: forall (i=L;i<U;i++) in parallel{
  S1, … , Sn
}
barrier_synchronize();
L': forall (j=L';j<U';j++) in parallel{
  S'1, … , S'n
}
```


Parallelization of loops IV:

```
L': for (j=L';j<U';j++) {
  L: for (i=L;i<U;i++) {
     S1, … , Sn
  }
}
```

If no loop-carried dependencies via `L` but via `L'`

```
L': for (j=L';j<U';j++) {
  L: forall (i=L;i<U;i++) in parallel{
    S1, … , Sn
  }
  barrier_synchronize();
}
```


Sequential loop fusion

```
L: for (i=L;i<U;i++) {
  S1, … , Sn
}
L': for (i=L;i<U;i++) {
  S'1, … , S'n
}
```

If no serial-fusion preventing dependencies:

```
L: for (i=L;i<U;i++) {
  S1, … , Sn
  S'1, … , S'n
}
```

Serial-fusion preventing

```
L: for (i=L;i<U;i++) {
  S: a[i]=…
}
L’: for (i=L;i<U;i++) {
  S’…=a[i+c] //defined in loop L
}
```

```
L L’: for (i=L;i<U;i++) {
  S:a[i]=…
  S’…=a[i+c] //defined before loop L L’
}
```

Dependence $S\,\delta\,S'$ due to index vectors i,i' with i[loop depth]>i'[loop depth]

Parallel loop fusion

```
L: forall (i=L;i<U;i++) in parallel{
  S1, … , Sn
}
barrier_synchronize();
L’: forall (i=L;i<U;i++) in parallel{
  S’1, … , S’n
}
```

If no parallel-fusion preventing dependencies:

```
L: for (i=L;i<U;i++) {
  S1, … , Sn
  S’1, … , S’n
}
```


Parallel-fusion preventing

```
L: forall (i=L;i<U;i++) in parallel{
  S: a[i]=…
}
barrier_synchronize();
L’: forall (i=L;i<U;i++) {
  S’:…=a[i-c] //defined in L
}
```

```
L L’: forall (i=L;i<U;i++) in parallel{
  S :a[i]=…
  S’: …=a[i-/+c] //potentially defined before L L’
}
```

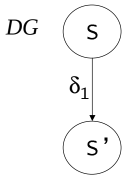
Dependence $S\,\delta\,S'$ due to index vectors i, i' with i[loop depth] $\neq$ i' [loop depth]

Dependence Graph

- Directed graph $DG=(N,E)$ induced by δ
  - Node for each statement S in the program
  - Edge iff $S\,\delta\,S'$
- Acyclic condensation $AC(DG)$ of DG
  - Node for each strongly connected component of DG
  - Edges in $AC(DG)$ for edges DG in between strongly connected components of DG
- Regions of DG are nodes of AC
- DG_c is DG restricted by a certain loop depth $>c$
- Other definitions restricted accordingly
- Decision based on these graphs

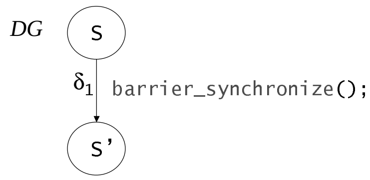
Example

```
for (i=1;i<100;i++) {
  S:a[i] = b[i]*c[i]+d[i];
  S’:b[i] = c[i]/d[i-1]+a[i-3];
}
```



Example

```
forall (i=1;i<100;i++) in parallel{
  a[i] = b[i]*c[i]+d[i];
  barrier_synchronize();
  b[i] = c[i]/d[i-1]+a[i-3];
}
```



Parallel code generation (sketch)

- Generate a parallel loop for the outermost possible region
  - Check for outermost loop (depth $c = 1$) if parallelizable (no loop carried dependency)
  - If not generate sequential code
  - Recursively, go on with loop depth $c = 1, 2, \ldots$
- Custer regions whenever possible by parallel or serial loop fusion (without barrier)