TDDD56 Multicore and GPU Programming

Optimization and Parallelization of Sequential Programs

Lecture 8

Christoph Kessler

IDA / PELAB
Linköping University
Sweden

Outline

Towards (semi-)automatic parallelization of sequential programs

- Data dependence analysis for loops
- Some loop transformations
  - Loop invariant code hoisting, loop unrolling, loop fusion, loop interchange, loop blocking / tiling, scalar expansion
- Static loop parallelization
- Run-time loop parallelization
  - Doacross parallelization
  - Inspector-executor method
- Speculative parallelization (later, if time)
- Auto-tuning (later, if time)

Foundations: Control and Data Dependence

- Consider statements *S, T* in a sequential program (*S*=*T* possible)
  - Scope of analysis is typically a function, i.e. intra-procedural analysis
  - Assume that a control flow path *S* … *T* is possible
  - Can be done at arbitrary granularity (instructions, operations, statements, compound statements, program regions)
  - Relevant are only the read and write effects on memory (i.e. on program variables) by each operation, and the effect on control flow

Example:

```
S:  if (…) {
       …
T:     …
       …
    }
```

- **Control dependence** *S* → *T*, if the fact whether *T* is executed may depend on *S* (e.g. condition)
  - Implies that relative execution order *S* → *T* must be preserved when restructuring the program
  - Mostly obvious from nesting structure in well-structured programs, but more tricky in arbitrary branching code (e.g. assembler code)

Foundations: Control and Data Dependence

- **Data dependence** *S* → *T*, if statement *S may* execute (dynamically) before *T* and both *may* access the same memory location and at least one of these accesses is a write
  - Means that execution order "*S* before *T*" must be preserved when restructuring the program
  - In general, only a conservative over-estimation can be determined statically
  - **flow dependence**: (RAW, read-after-write)
    - *S* may write a location z that *T* may read
  - **anti dependence**: (WAR, write-after-read)
    - *S* may read a location x that *T* may overwrite
  - **output dependence**: (WAW, write-after-write)
    - both *S* and *T* may write the same location

Example:

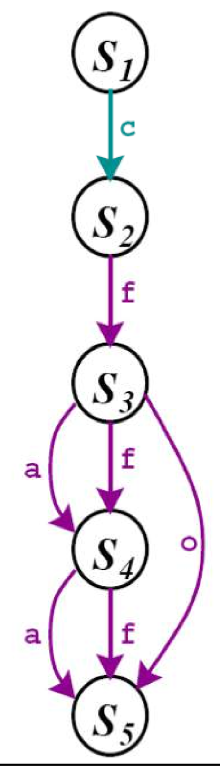
```
S:  z = … ;
    …
T:  … = ..z.. ;
```

(flow dependence)

Dependence Graph

- **(Data, Control, Program) Dependence Graph:** Directed graph, consisting of all statements as vertices and all (data, control, any) dependences as edges.

S_1:	**if** (e) **goto** S_3
S_2:	$a \leftarrow \ldots$
S_3:	$b \leftarrow a * c$
S_4:	$c \leftarrow b * f$
S_5:	$b \leftarrow x + f$



control dependence by control flow: $S_1 \delta^c S_2$

data dependence:

flow / true dependence: $S_3 \delta^f S_4$
$S_3 \lhd S_4$ and $\exists b : S_3$ writes b, S_4 reads b

anti-dependence: $S_3 \delta^a S_4$
$S_3 \lhd S_4$ and $\exists c : S_3$ reads c, S_4 writes c

output dependence: $S_3 \delta^o S_5$
$S_3 \lhd S_5$ and $\exists b : S_3$ writes b, S_5 writes b

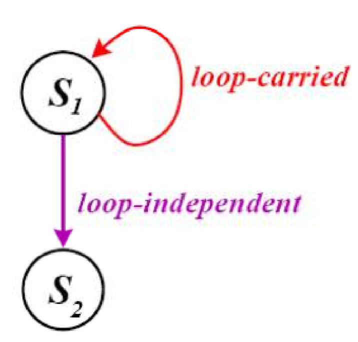
Data Dependence Graph

- **Data dependence graph for straight-line code** ("basic block", no branching) is always acyclic, because relative execution order of statements is forward only.
- **Data dependence graph for a loop:**
  - Dependence edge *S*→*T* if a dependence may exist *for some pair of instances* (iterations) of *S, T*
  - Cycles possible
  - Loop-independent versus loop-carried dependences

Example:

```
for (i=1; i<n; i++) {
S1:  a[i] = b[i] + a[i-1];
S2:  b[i] = a[i];
}
```

(assuming we know statically that arrays a and b do not intersect)



Example

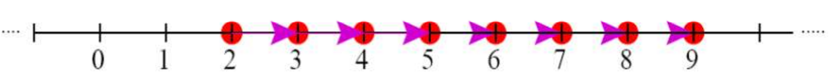
for i **from** 2 **to** 9 **do**
$S_1 \quad X[i] \leftarrow Y[i] + Z[i]$
$S_2 \quad A[i] \leftarrow X[i-1] + 1$
od

(assuming that we statically know that arrays A, X, Y, Z do not intersect, otherwise there might be further dependences)

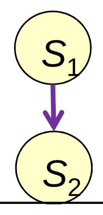
	$i = 2$	$i = 3$	$i = 4$	...
S_1	$X[2] \leftarrow Y[2] + Z[2]$	$X[3] \leftarrow Y[3] + Z[3]$	$X[4] \leftarrow Y[4] + Z[4]$	...
S_2	$A[2] \leftarrow X[1] + 1$	$A[3] \leftarrow X[2] + 1$	$A[4] \leftarrow X[3] + 1$	...

There is a loop-caried, forward, flow dependence from S_1 to S_2.

Iteration space dependence graph:
(Iterations unrolled)



Data dependence graph:



Why Loop Optimization and Parallelization

Loops are a promising object for program optimizations, including automatic parallelization:

- High execution frequency
  - Most computation done in (inner) loops
  - Even small optimizations can have large impact (cf. Amdahl's Law)
- Regular, repetitive behavior
  - compact description
  - *relatively* simple to analyze statically
- Well researched

Loop Optimizations – General Issues

- Move loop invariant computations out of loops
- Modify the order of iterations or parts thereof

Goals:

- Improve data access locality
- Faster execution
- Reduce loop control overhead
- Enhance possibilities for loop parallelization or vectorization

Only transformations that preserve the program semantics (its input/output behavior) are admissible

- Conservative (static) criterium: preserve data dependences
- Need data dependence analysis for loops (→ DF00100)

Loop Invariant Code Hoisting

- **Move loop invariant code out of the loop**
  - Compilers can do this automatically *if* they can statically find out what code is loop invariant
  - Example:

```
for (i=0;  i<10;  i++)
    a[i] = b[i]  + c / d;
```

```
tmp = c / d;
for (i=0;  i<10;  i++)
    a[i] = b[i]  + tmp;
```


Loop Unrolling

- **Loop unrolling**
  - Can be enforced with compiler options e.g. –funroll=2
  - Example:

```
for (i=0; i<50; i++) {
    a[i] = b[i];
}
```

Unroll by 2:

```
for (i =0; i<50; i+=2) {
    a[i] = b[i];
    a[i+1] = b[i+1];
}
```

☺ Reduces loop overhead (total # comparisons, branches, increments)

☺ Longer loop body may enable further local optimizations (e.g. common subexpression elimination, register allocation, instruction scheduling, using SIMD instructions)

☹ longer code

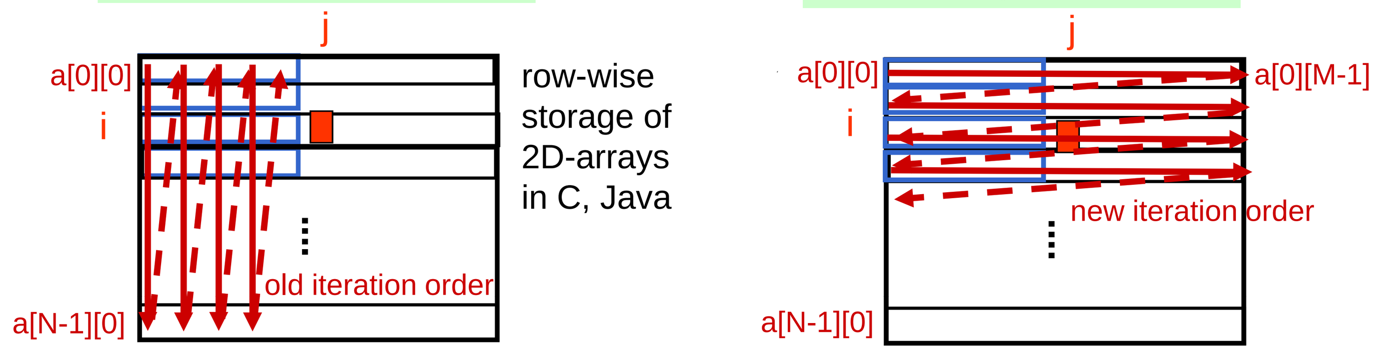
→ Exercise: Formulate the unrolling rule for statically unknown upper loop limit

Loop Interchange (1)

- For properly nested loops (statements in innermost loop body only)
  - Example 1:

```
for (j=0;  j<M;  j++)
  for (i=0;  i<N;  i++)
    a[ i ][ j ] = 0.0 ;
```

```
for (i=0;  i<N;  i++)
  for (j=0;  j<M;  j++)
    a[ i ][ j ] = 0.0 ;
```



- Can improve data access locality in memory hierarchy (fewer cache misses / page faults)

Foundations: Loop-Carried Data Dependences

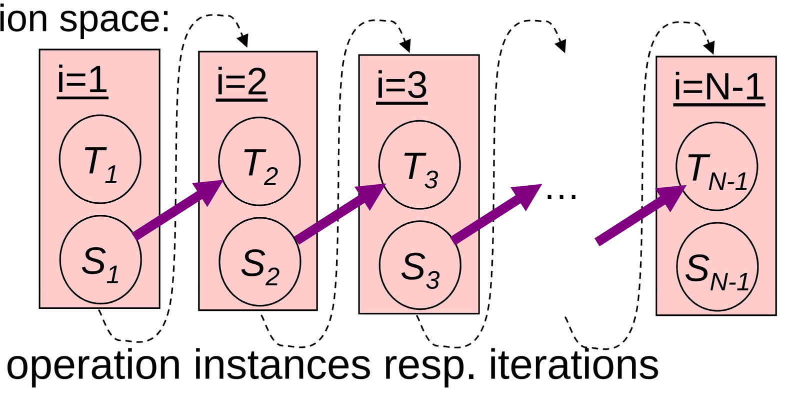
- Recall: **Data dependence** *S* → *T*,
  if operation *S may* execute (dynamically) before operation *T*
  and both *may* access the same memory location
  and at least one of these accesses is a write
  - In general, only a conservative over-estimation can be determined statically.

```
S:  z = ... ;
      ...
T:  ... = ..z.. ;
```

- Data dependence *S*→*T* is called ***loop carried*** by a loop *L*
  if the data dependence *S*→*T* may exist for instances of *S* and *T*
  in different iterations of *L*.
  - Example:

```
L:  for (i=1; i<N; i++) {
T_i:    ... = x[ i-1 ];
S_i:    x[ i ] = ...;
    }
```

Iteration space:



→ partial order between the operation instances resp. iterations

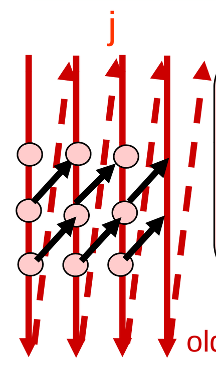
Loop Interchange (2)

- **Be careful** with loop carried data dependences!
  - Example 2:

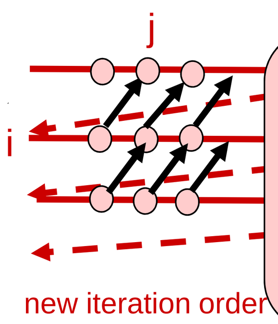
```
for (j=1; j<M; j++)
  for (i=0; i<N; i++)
    a[i][j] =...a[i+1][j-1]...;
```

```
for (i=0; i<N; i++)
  for (j=1; j<M; j++)
    a[i][j] =...a[i+1][j-1]...;
```

Iteration space:



Iteration (*j,i*) reads location a[*i+1*][*j-1*] that was written in an earlier iteration, (*i-1,j+1*)



Iteration (*i,j*) reads location a[*i+1*][*j-1*], that will be over-written in a later iteration (*i+1,j-1*)

- Interchanging the loop headers would violate the partial iteration order given by the data dependences

Loop Interchange (3)

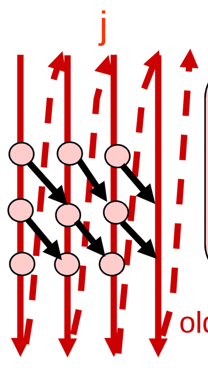
- **Be careful** with loop-carried data dependences!
  - Example 3:

```
for (j=1; j<M; j++)
  for (i=1; i<N; i++)
    a[i][j] =...a[i-1][j-1]...;
```

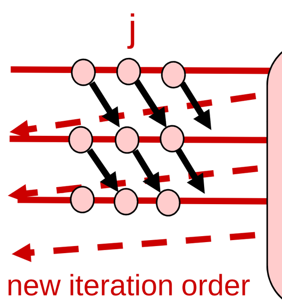
OK

```
for (i=1; i<N; i++)
  for (j=1; j<M; j++)
    a[i][j] =...a[i-1][j-1]...;
```

Iteration space:



Iteration (*j,i*) reads location a[*i-1*][*j-1*] that was written in earlier iteration (*j-1,i-1*)



Iteration (*i,j*) reads location a[*i-1*][*j-1*] that was written in earlier iteration (*i-1,j-1*)

- Generally: Interchanging loop headers is only admissible if loop-carried dependences have the same direction for all loops in the loop nest (all directed along or all against the iteration order)

Loop Fusion

- Merge subsequent loops with same header
  - Safe if neither loop carries a (backward) dependence
  - Example:

```
for (i=0; i<N; i++)
  a[ i ] = ... ;
for (i=0; i<N; i++)
  ... = ... a[ i ] ... ;
```

```
for (i= 0; i<N; i++) {
  a[ i ] = ... ;
  ... = ... a[ i ] ... ;
}
```

For N sufficiently large, a[*i*] will no longer be in the cache at this time

OK – Read of a[*i*] still after write of a[*i*], for all *i*

☺ Can improve data access locality and reduces number of branches

Loop Iteration Reordering

A transformation that reorders the iterations of a level-*k*-loop,
without making any other changes,
is valid if the loop carries no dependence.

Example:

```
   for (i=1; i<n; i++)
     for (j=1; j<m; j++)
       for (k=1; k<r; k++)
S:       a[i][j][k] = ... a[i][j-1][k] ...          (=,<,=)
```

j-loop carries a dependence, its iteration order must be preserved

Loop Parallelization

A transformation that reorders the iterations of a level-*k*-loop,
without making any other changes,
is valid if the loop carries no dependence.

Example:

```
   for (i=1; i<n; i++)
     for (j=1; j<m; j++)
       for (k=1; k<r; k++)
S:       a[i][j][k] = ... a[i][j-1][k] ...          (=,<,=)
```

j-loop carries a dependence, its iteration order must be preserved

It is valid to convert a sequential loop to a parallel loop
if it does not carry a dependence.

Example:

```
   for (i=1; i<n; i++)
S:   b[i] = 2 * c[i];
```

Loop parallelization

```
forall ( i, 1, n, p )
  b[i] = 2 * c[i];
```

Remark on Loop Parallelization

- Introducing temporary copies of arrays can remove some antidependences to enable automatic loop parallelization
- Example:

```
for (i=0; i<n; i++)
    a[i] = a[i] + a[i+1];
```

- The loop-carried dependence can be eliminated:

```
for (i=0; i<n; i++)
    aold[i+1] = a[i+1];        <- Parallelizable loop
for (i=0; i<n; i++)
    a[i] = a[i] + aold[i+1];   <- Parallelizable loop
```


Strip Mining / Loop Blocking / -Tiling

```
for (i=0; i<n; i++)
  a[i] = b[i] + c[i];
```

↓ loop blocking with block size s

```
for (i1=0; i1<n; i1+=s)              // loop over blocks
  for (i2=0; i2<min(n-i1,s); i2++)   // loop within blocks
    a[i1+i2] = b[i1+i2] + c[i1+i2];
```

Tiling = blocking in multiple dimensions + loop interchange

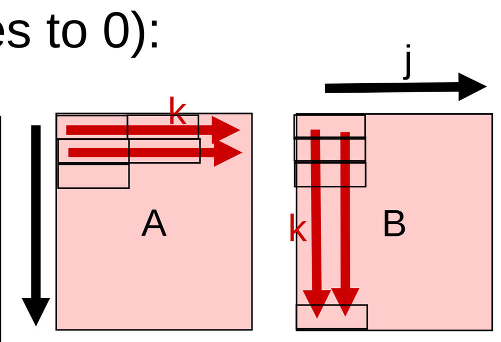
Goal: increase locality; support vectorization (vector registers)

Reverse transformation: Loop linearization

Tiled Matrix-Matrix Multiplication (1)

- Matrix-Matrix multiplication $C = A \times B$ here for square ($n \times n$) matrices C, A, B, with n large (~10^3):
  - $C_{ij} = \Sigma_{k=1..n} A_{ik} B_{kj}$ for all $i, j = 1...n$
- Standard algorithm for Matrix-Matrix multiplication (here without the initialization of C-entries to 0):

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      C[i][j] += A[i][k] * B[k][j];
```



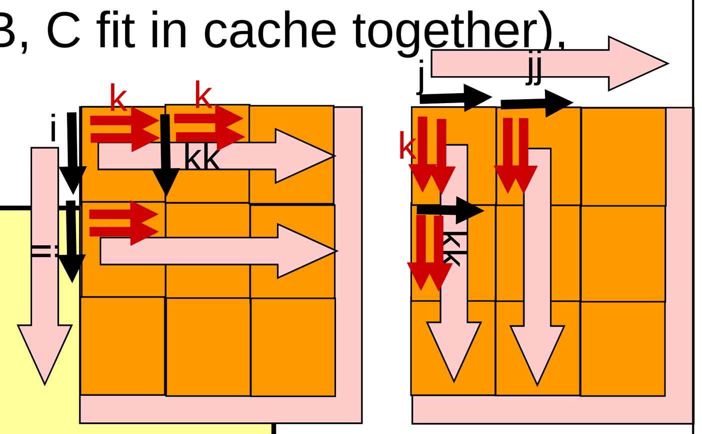
Good spatial locality on A, C

Bad spatial locality on B (many capacity misses)

Tiled Matrix-Matrix Multiplication (2)

- Block each loop by block size S (choose S so that a block of A, B, C fit in cache together), then interchange loops
- Code after tiling:

```
for (ii=0; ii<n; ii+=S)
  for (jj=0; jj<n; jj+=S)
    for (kk=0; kk<n; kk+=S)
      for (i=ii; i < ii+S; i++)
        for (j=jj; j < jj+S; j++)
          for (k=kk; k < kk+S; k++)
            C[i][j] += A[i][k] * B[k][j];
```



Good spatial locality for A, B and C

Scalar Expansion / Array Privatization

promote a scalar temporary to an array to break a dependence cycle

for i **from** 1 **to** N **do**
  $t \leftarrow a[i] + b[j]$
  $c[i] \leftarrow t + 1$
od

expand scalar t: →

if $N \geq 1$
  allocate $t'[1..N]$
  for i **from** 1 **to** N **do**
    $t'[i] \leftarrow a[i] + b[j]$
    $c[i] \leftarrow t'[i] + 1$
  od
  $t \leftarrow t'[N]$ // *if t live on exit*
fi

\+ removes the loop-carried antidependence due to t
  → can now parallelize the loop!

\- needs more array space

Loop must be countable, scalar must not have upward exposed uses.

May also be done conceptually only, to enable parallelization: just create one private copy of t for every processor = array privatization

Idiom recognition and algorithm replacement

Traditional loop parallelization fails for loop-carried dep. with distance 1:

```
S0:   s = 0;
      for (i=1; i<n; i++)
S1:      s = s + a[i];

S2:   a[0] = c[0];
      for (i=1; i<n; i++)
S3:      a[i] = a[i-1] * b[i] + c[i];
```

C. Kessler: Pattern-driven automatic parallelization. *Scientific Programming*, 1996.

A. Shafiee-Sarvestani, E. Hansson, C. Kessler: Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization. *Int. J. on Parallel Programming*, 2013.

↓ Idiom recognition (pattern matching)

```
S1':  s = VSUM( a[1:n-1], 0 );

S3':  a[0:n-1] = FOLR( b[1:n-1], c[0:n-1], mul, add );
```

↓ Algorithm replacement

```
S1'': s = par_sum( a, 0, n, 0 );
```

TDDD56 Multicore and GPU Programming

For further loop transformations…

… see DF00100 (TDDC86)
Advanced Compiler Construction

Index set splitting, Loop unswitching,
Loop skewing, Loop distribution,
Software Pipelining of Loops, …

Christoph Kessler, IDA,
Linköpings universitet, 2013.

Remark on static analyzability (1)

- Static dependence information is always a (safe) overapproximation of the real (run-time) dependences
  - Finding out the real ones exactly is statically undecidable!
  - If in doubt, a dependence must be assumed
    → may prevent some optimizations or parallelization
- One main reason for imprecision is **aliasing**, i.e. the program may have several ways to refer to the same memory location
  - Example: Pointer aliasing

```
void mergesort ( int* a, int n )
{ …
   mergesort ( a,  n/2 );
   mergesort ( a + n/2, n-n/2 );
   …
}
```

How could a static analysis tool (e.g., compiler) know that the two recursive calls read and write disjoint subarrays of a?

Remark on static analyzability (2)

- Static dependence information is always a (safe) overapproximation of the real (run-time) dependences
  - Finding out the latter exactly is statically undecidable!
  - If in doubt, a dependence must be assumed
    → may prevent some optimizations or parallelization
- Another reason for imprecision are **statically unknown values** that imply whether a dependence exists or not
  - Example: Unknown dependence distance

```
// value of K statically unknown
for ( i=0; i<N; i++ )
{ …
   S:   a[i] = a[i] + a[K];
   …
}
```

Loop-carried dependence if K < N.
Otherwise, the loop is parallelizable.

Outlook: Runtime Parallelization

Sometimes parallelizability cannot be decided statically.

```
if is_parallelizable(...)
   forall i in [0..n-1] do      // parallel version of the loop
      iteration(i);
   od
else
   for i from 0 to n − 1 do     // sequential version of the loop
      iteration(i);
   od
fi
```

The runtime dependence test is_parallelizable(...)
itself may partially run in parallel.

TDDD56 Multicore and GPU Programming

Run-Time Parallelization

Christoph Kessler, IDA,
Linköpings universitet, 2013.

Goal of run-time parallelization

- Typical target: **irregular loops**

  for (i=0; i<n; i++)
     a[i] = *f* (a[*g*(i)], a[*h*(i)], ...);

  - Array index expressions *g, h...* depend on run-time data
  - Iterations cannot be statically proved independent
    (and not either dependent with distance +1)
- **Principle:**
  At runtime, inspect *g, h* ... to find out the real dependences and compute a schedule for partially parallel execution
  - Can also be combined with speculative parallelization

Overview

- **Run-time parallelization of irregular loops**
  - DOACROSS parallelization
  - Inspector-Executor Technique (shared memory)
  - Inspector-Executor Technique (message passing) *
  - Privatizing DOALL Test *
- **Speculative run-time parallelization of irregular loops ***
  - LRPD Test *
- **General Thread-Level Speculation**
  - Hardware support *

* = not covered in this course. See the references.

DOACROSS Parallelization

- Useful if loop-carried dependence distances are unknown, but often > 1
- Allow independent subsequent loop iterations to overlap
- Bilateral synchronization between really-dependent iterations

Example:

```
for ( i=0; i<n; i++)
    a[i]  =  f ( a[ g(i) ], ... );
```

```
sh float aold[n];
sh flag done[n];   // flag (semaphore) array
forall i in 0..n-1  {  // spawn n threads, one per iteration
    done[n] = 0;
    aold[i] = a[i];    // create a copy
}
forall i in 0..n-1  {  // spawn n threads, one per iteration
    if (g(i) < i)   wait until done[ g(i) ] );
                    a[i]  =  f ( a[ g(i) ], ... );
                    set( done[i] );
    else
                    a[i]  =  f ( aold[ g(i) ], ... );  set done[i];
}
```


Inspector-Executor Technique (1)

- Compiler generates 2 pieces of customized code for such loops:
- **Inspector**
  - calculates values of index expression by simulating whole loop execution
    - typically, based on sequential version of the source loop (some computations could be left out)
  - computes implicitly the real iteration dependence graph
  - computes a parallel schedule as (greedy) wavefront traversal of the iteration dependence graph in topological order
    - all iterations in same wavefront are independent
    - schedule depth = #wavefronts = critical path length
- **Executor**
  - follows this schedule to execute the loop

Inspector-Executor Technique (2)

- **Source loop:**

```
for ( i=0; i<n; i++)
    a[i]  =  f ( a[ g(i) ], a[ h(i) ], ... );
```

- **Inspector:**

```
int wf[n];  // wavefront indices
int depth = 0;
for (i=0; i<n; i++)
   wf[i] = 0;   // init.
for (i=0; i<n; i++) {
   wf[i] = max ( wf[ g(i) ], wf[ h(i) ], ... ) + 1;
   depth = max ( depth, wf[i] );
}
```

- Inspector considers only flow dependences (RAW), anti- and output dependences to be preserved by executor

Inspector-Executor Technique (3)

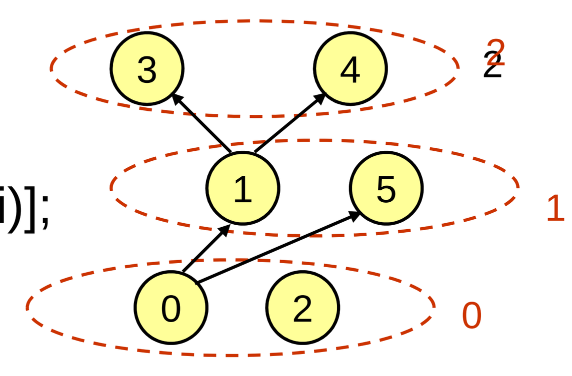
- **Example**:

```
for (i=0; i<n; i++)
    a[i] = ... a[ g(i) ] ...;
```

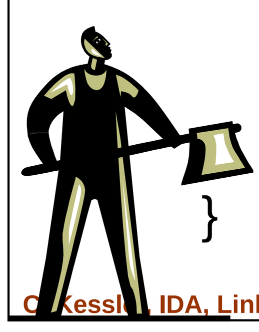
i	0	1	2	3	4	5
g(i)	2	0	2	1	1	0
wf[i]	0	1	0	2	2	1
g(i)<i ?	no	yes	no	yes	yes	yes

- **Executor**:

```
float aold[n];  // buffer array
aold[1:n] = a[1:n];
for (w=0; w<depth; w++)
   forall (i, 0, n, #)  if (wf[i] == w)  {
      a1 = (g(i) < i)? a[g(i)] : aold[g(i)];
      ...  // similarly, a2 for h etc.
      a[i] =  f ( a1, a2, ... );
   }
```



iteration (flow) dependence graph



Inspector-Executor Technique (4)

Problem: Inspector remains sequential – no speedup

Solution approaches:

- Re-use schedule over subsequent iterations of an outer loop if access pattern does not change
  - amortizes inspector overhead across repeated executions
- Parallelize the inspector using doacross parallelization [Saltz,Mirchandaney'91]
- Parallelize the inspector using sectioning [Leung/Zahorjan'91]
  - compute processor-local wavefronts in parallel, concatenate
  - trade-off schedule quality (depth) vs. inspector speed
  - Parallelize the inspector using bootstrapping [Leung/Z.'91]
  - Start with suboptimal schedule by sectioning, use this to execute the inspector → refined schedule

TDDD56 Multicore and GPU Programming

Questions?

Christoph Kessler, IDA,
Linköpings universitet, 2013.

Some references on Dependence Analysis, Loop optimizations and Transformations

- H. Zima, B. Chapman: *Supercompilers for Parallel and Vector Computers*. Addison-Wesley / ACM press, 1990.
- M. Wolfe: *High-Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
- R. Allen, K. Kennedy: *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

Idiom recognition and algorithm replacement:

- C. Kessler: Pattern-driven automatic parallelization. *Scientific Programming* **5**:251-274, 1996.
- A. Shafiee-Sarvestani, E. Hansson, C. Kessler: Extensible recognition of algorithmic patterns in DSP programs for automatic paral-lelization. *Int. J. on Parallel Programming*, 2013.

Some references on run-time parallelization

- R. Cytron: Doacross: Beyond vectorization for multiprocessors. Proc. ICPP-1986
- D. Chen, J. Torrellas, P. Yew: An Efficient Algorithm for the Run-time Parallelization of DO-ACROSS Loops, Proc. IEEE Supercomputing Conf., Nov. 2004, IEEE CS Press, pp. 518-527
- R. Mirchandaney, J. Saltz, R. M. Smith, D. M. Nicol, K. Crowley: Principles of run-time support for parallel processors, Proc. ACM Int. Conf. on Supercomputing, July 1988, pp. 140-152.
- J. Saltz and K. Crowley and R. Mirchandaney and H. Berryman: Runtime Scheduling and Execution of Loops on Message Passing Machines, *Journal on Parallel and Distr. Computing* 8 (1990): 303-312.
- J. Saltz, R. Mirchandaney: The preprocessed doacross loop. Proc. ICPP-1991 Int. Conf. on Parallel Processing.
- S. Leung, J. Zahorjan: Improving the performance of run-time parallelization. Proc. ACM PPoPP-1993, pp. 83-91.
- Lawrence Rauchwerger, David Padua: The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. Proc. ACM Int. Conf. on Supercomputing, July 1994, pp. 33-45.
- Lawrence Rauchwerger, David Padua: The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. Proc. ACM SIGPLAN PLDI-95, 1995, pp. 218-232.