Overview of Parallelization Techniques I

Christian Lengauer

Fakultät für Informatik und Mathematik

16. June 2008, Schloss Rheinfels, St.Goar, Germany

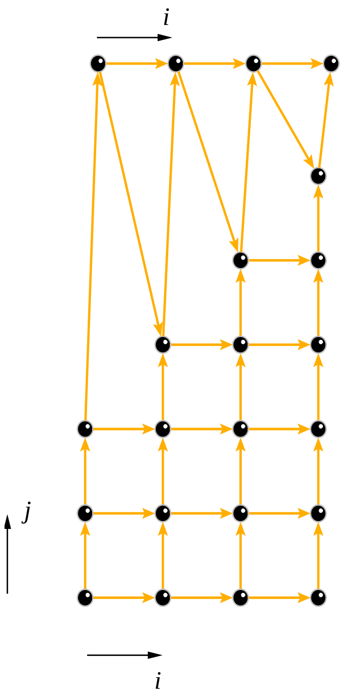
Text-Based Loop Transformations

- Text-Based Loop Transformations:
  - fission, fusion, permutation, skewing, unrolling, unswitching
  - \+ easy to apply
  - \+ in simple cases often all you need
  - – do not support a search for the best solution
  - – favor some solutions over others that may be better
- Model-Based Loop Transformations:
  - map source code to an execution model
  - find the optimal parallel solution in this model
  - \+ quality metric: a given objective function
  - \+ search and transformation completely automatic
  - – analysis and target code can become complex
  - – optimality in the model need not imply efficient target code

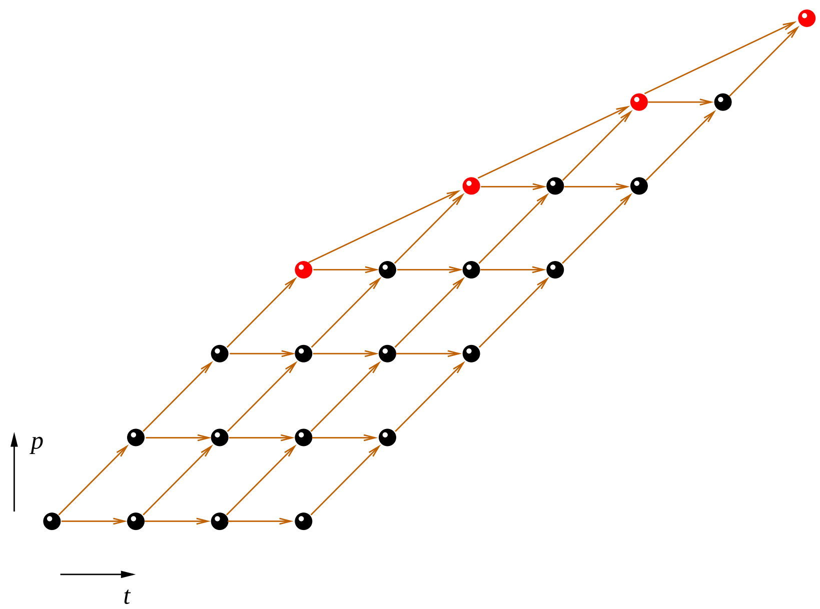
The Basic Polytope Model

for $i = 1$ to n do
  for $j = 0$ to $i + m$ do
    $A(i,j) = A(i-1,j) + A(i,j-1)$
  od
  $A(i,i+m+1) = A(i-1,i+m)+A(i,i+m)$
od

parfor $p = 1$ to $m + n$ do
  for $t = \max(p-1, 2*p-m-2)$ to $n+p-2$ do
    $A(2+t-p,p) = A(1+t-p,p) + A(2+t-p,p-1)$
  od
  if $p \geq m+1$ then
    $A(p-m,1+p) = A(p-m,p) + A(p-m-1,p)$
  fi
od



source operation dependence graph



target operation dependence graph

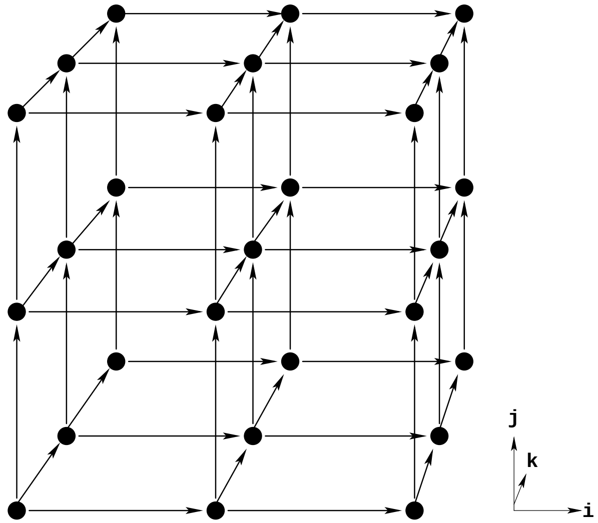
Capabilities of the Basic Model

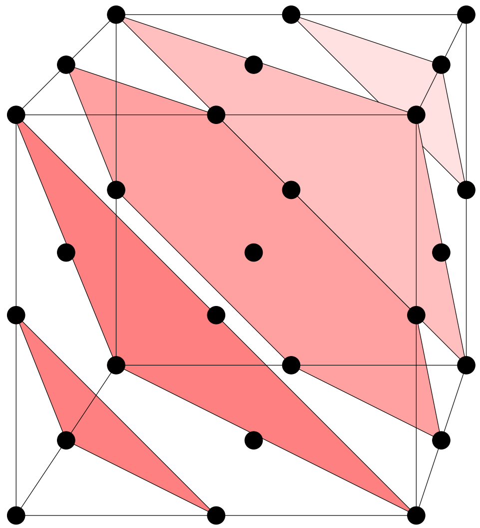
- Fully automatic dependence analysis
- Optimizing search of the best solution in the solution space, w.r.t. a given objective function
- Exemplary objective functions:
  - minimal number of parallel steps; minimal number of processors
  - minimal number of parallel steps; maximal throughput
  - minimal number of communications
- Challenge: efficient target code
- Standard example: square matrix product
  - source code:

for $i = 0$ to $n-1$ do
  for $j = 0$ to $n-1$ do
    for $k = 0$ to $n-1$ do
      $C(i,j) = C(i,j) + A(i,k) * B(k,j)$
    od
  od
od

Example: Square Matrix Product

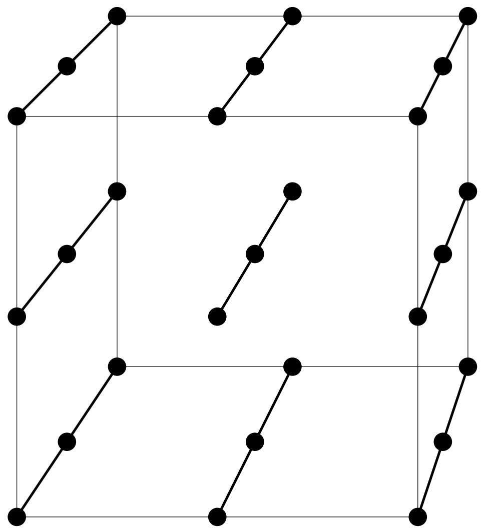
index space,
dependences

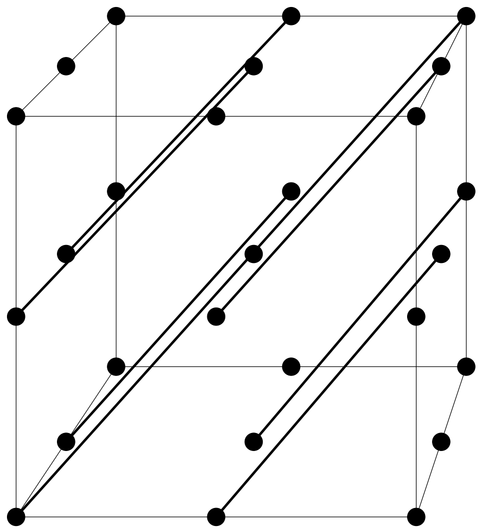




parallel
steps

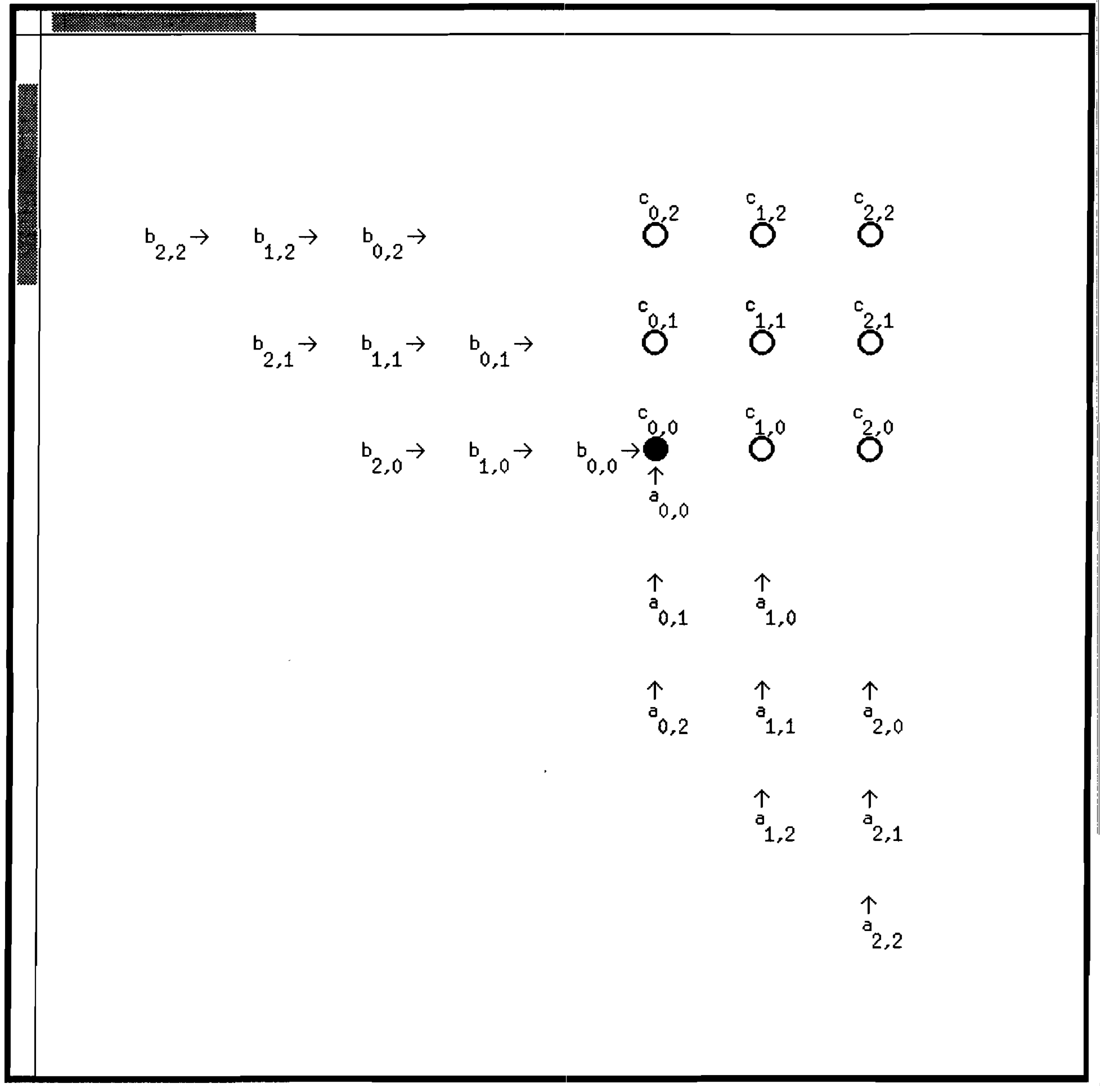
square
processor
array



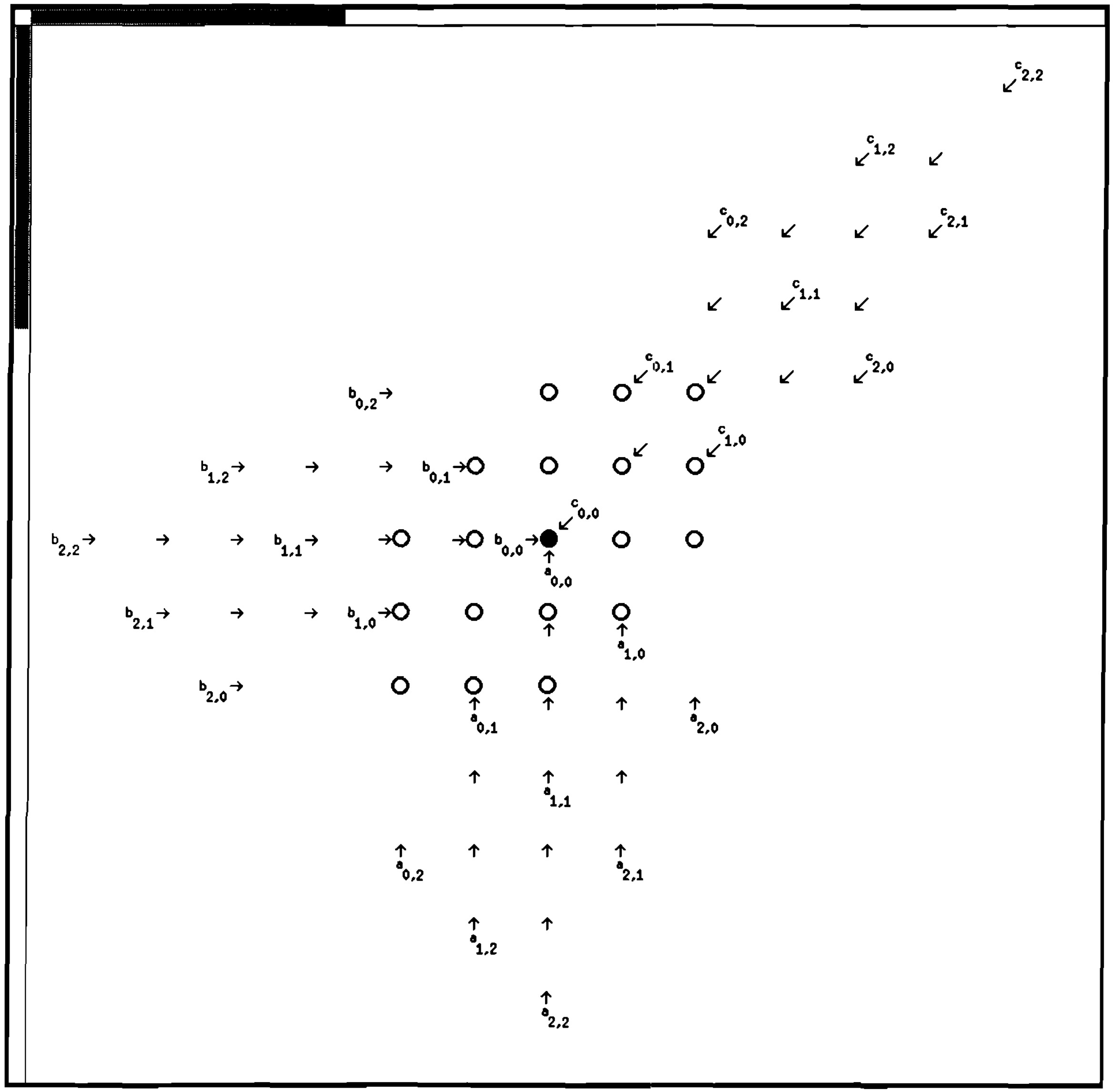


hexagonal
processor
array

Quadratic Solution



Hexagonal Solution



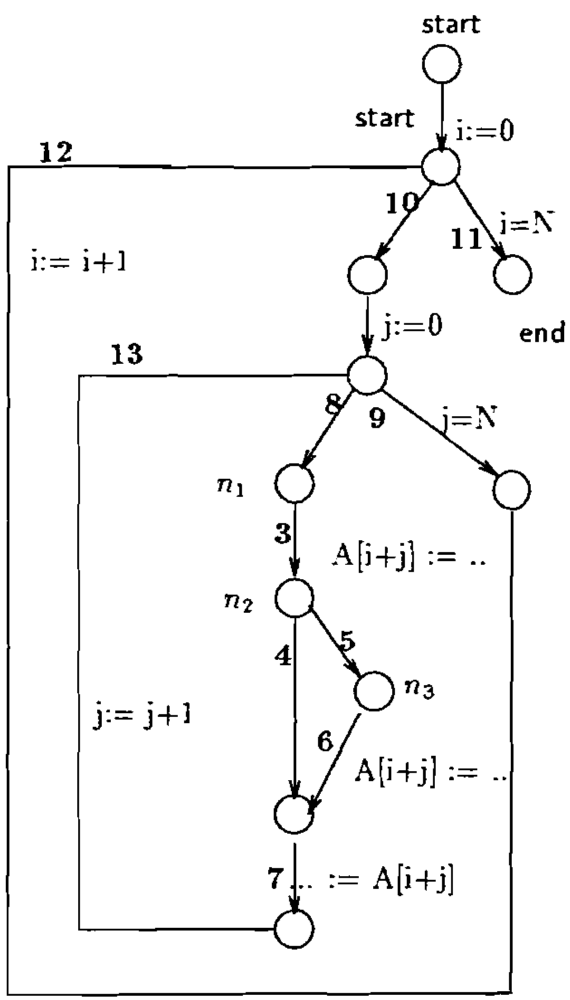
Restrictions and Uses of the Basic Model

- Restrictions:
  - Loop bounds must be affine expressions in the outer loop indices and structure parameters.
  - Array indices must be affine expressions in the loop indices.
  - Assigments may be to array variables or scalar variables.
  - Loop nests may be imperfect.
  - Calls of subprograms are considered atomic, i.e., are not subject to parallelization.
  - Pointer structures are not considered (unless coded as arrays)
  - Target loop nests may be
    - synchronous (outer loops sequential),
    - asynchronous (outer loops parallel).
- Uses:
  - Loop parallelization
  - Cache optimization

Extension 1: Conditional Statements in the Body

- Consequence:
  - Dependences vary between branches.

```
    real A[0:2*N+1]
    for i = 0 to N
       for j = 0 to N
3             A[i+j+1] := ..
              if (P) then
6                A[i+j] := ..
              end if
7             .. := A[i+j]
       end for
    end for
```

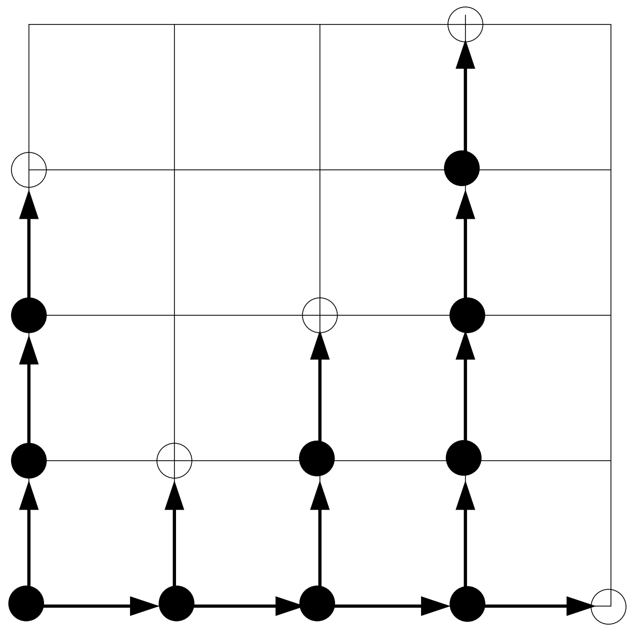


Extension 1: Conditional Statements in the Body

- Technique:
  - A precise reaching definition analysis that combines
    - the iterative solution of data flow equations,
      discovers dependences between entire arrays,
      can handle conditionals
    - integer linear programming,
      discovers dependences between individual array elements
  - attaches conditions to dependences
  - builds the unconditional union of all conditional dependences

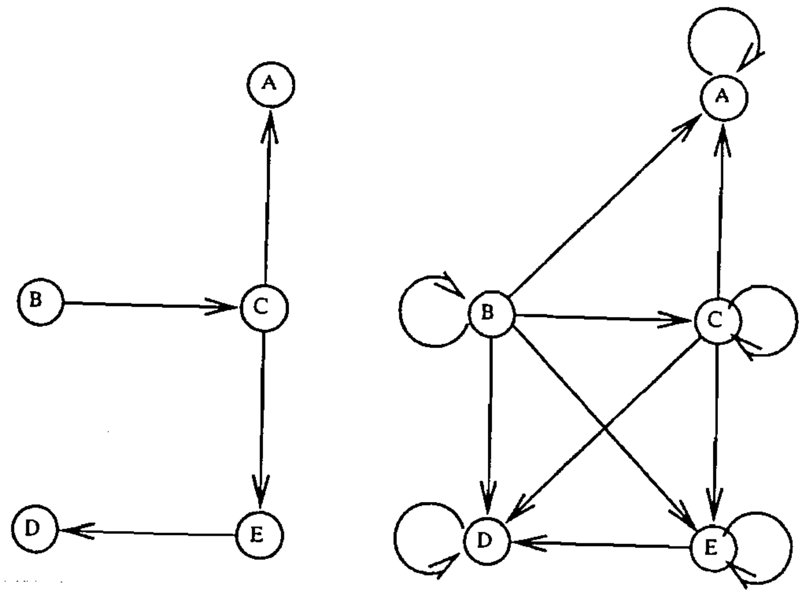
Extension 2: WHILE Loops in the Loop Nest

- Consequences:
  - In WHILE dimensions, the number of loop steps is determined at run time.
  - The static index space is not a polytope but a polyhedron.
  - The dynamic index space in the unbounded direction is uneven (a “comb”).



Extension 2: Example (Convex Hull)

n	$node$	$nrsuc$	suc	rt
0	A	0		A
1	B	1	C	B, C, A, E, D
2	C	2	A, E	C, A, E, D
3	D	0		D
4	E	1	D	E, D



Extension 2: Example (Convex Hull)

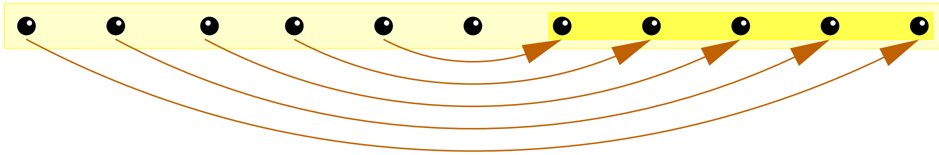
S_1: for $n := 0$ while $node[n] \neq \perp$ do
S_2: $rt[n, 0] := n$
S_3: $nxt[n] := 1$
S_4: for $d := 0$ while $rt[n, d] \neq \perp$ do
S_5: if $\neg tag[n, rt[n, d]]$ then
S_6: $tag[n, rt[n, d]] := tt$
S_7: for $s := 0$ to $nrsuc[rt[n, d]] - 1$ do
S_8: $rt[n, nxt[n] + s] := suc[rt[n, d], s]$
enddo
S_9: $nxt[n] := nxt[n] + nrsuc[rt[n, d]]$
endif
enddo
enddo

Extension 2: Two Approaches

- Conservative Approach:
  - The control dependence of the WHILE loop is respected.
  - One WHILE loop remains sequential, but may be distributed.
  - A nest of WHILE loops may also be parallel.
  - Challenge: detecting the end of a “tooth” of the “comb”; solved for shared and distributed memory.
- Speculative Approach:
  - The control dependence of the WHILE loop is ignored.
  - oNE WHILE loop may be parallel.
  - Additional memory space may be required.
  - A rollback of iterations may be necessary.
  - Challenges: implementing rollback; avoiding rollback; minimizing memory consumption.

Extension 3: Index Set Splitting

- Idea:
  - Partition the index space automatically to break a dependence pattern and increase parallelism.



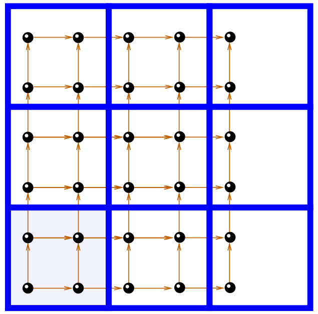
for $i = 0$ to $2*n-1$ do
  $A(i,0) = \dots A(2*n-i-1,0)$
od

$\Longrightarrow$

for $i = 0$ to $n-1$ do
  $A(i,0) = \dots A(2*n-i-1,0)$
od
for $i = n$ to $2*n-1$ do
  $A(i,0) = \dots A(2*n-i-1,0)$
od

- Technique:
  - Separate the sinks in the graph from the rest.
  - Propagate splits backwards to the context
    - Challenge: termination in the presence of cycles (cut off)
    - Challenge: exponential growth in the number of sources (heuristics)

Extension 4: Tiling



- Goal: Determine the optimal granularity of parallelism
  - When? (Before or after the parallelization.)
  - How? (Shape, form and size of the tiles.)
  - What? (Space and/or time.)
- When: After the parallelization
  - It's simpler and more widely applicable: only one perfect target loop nest.
  - It's more powerful: flexible space-time mapping before inflexible tiling.
- How: Space and time separately
  - The risk: heuristics wins only with certain allocations.
  - The gain: precise and independent adaptation to hardware parameters.
- What:
  - Tiling space: adapts to resources (# processors).
  - Tiling time: adapts to performance (computation/communication ratio).

Extension 5: Expressions

- Goal: Avoid duplicate computations
- Technique: Loop-carried code placement (LCCP)
  - Identifies expressions that have the same value.
  - Determines the optimal time and place for the evaluation.
  - Determines the optimal place for the result.
  - Hoists x-dimensional expressions out of y-dimensional loop nests ($y > x$)
- Example: Shallow Water Simulation

```
FORALL (j=1:n,i=1:m) H(i,j) =
&   P(i,j) + 0.25 * (U(i+1,j)*U(i+1,j) + U(i,j)*U(i,j)
&                  + V(i,j+1)*V(i,j+1) + V(i,j)*V(i,j))
```

$$\downarrow$$

```
FORALL (j=1:n,i=1:m+1) TMP1(i,j) = U(i,j)*U(i,j)
FORALL (j=1:n+1,i=1:m) TMP2(i,j) = V(i,j)*V(i,j)
FORALL (j=1:n,i=1:m) H(i,j) =
&   P(i,j) + 0.25 * (TMP1(i+1,j) + TMP1(i,j)
&                  + TMP2(i,j+1) + TMP2(i,j)
```

Extension 6: Non-Affine Array Index Expressions

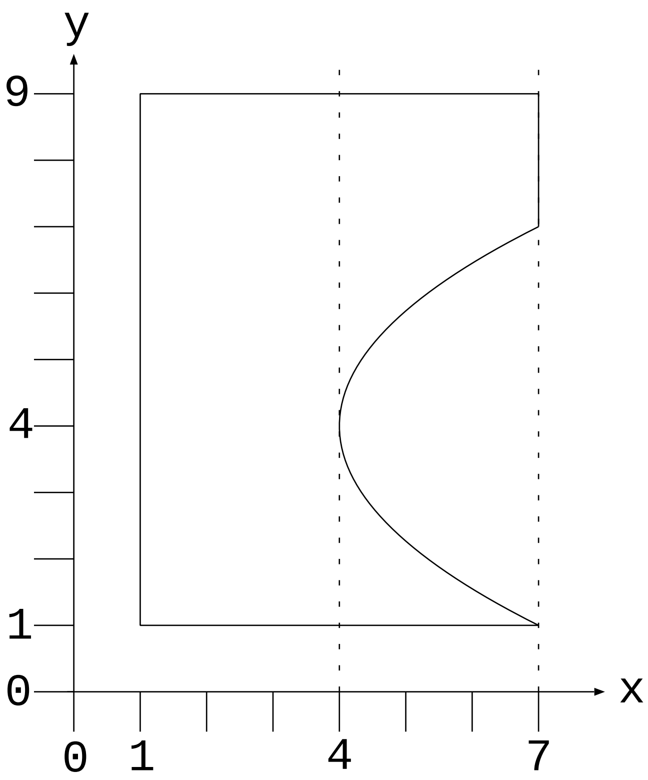
- Goal: Be able to handle array expressions of the form `A(p*i)`
- "Parameter" p:
  - Has a previously unknown but fixed value
  - Typical case: Extent of the polytope in some fixed dimension
- Application: Select a row or column of a matrix as a vector
- Technique:
  - Solve conflict equation system in $\mathbb{Z}$.
  - Algorithm known for exactly one parameter.
  - Math: entire quasi-polynomials.
- Challenge: Dependence analysis
  - Are the solutions inside or outside the iteration space?
    (Solving the existence inequations...)
  - What is the direction of a dependence?
    (Establishing an order...)

Extension 7: Non-Affine Loop Bounds

- Goal: Be able to scan domains with curved bounds.
  - Curves must be described by polynomials.
  - Domains are semi-algebraic sets
    (sets of solutions of inequation systems of polynomials in $\mathbb{Z}$).
- Applications:
  - Normal source code: Sieve of Eratosthenes (bound `i*i<=n`)
  - Loops with non-constant stride:
    - Example: `for (j=0; j<=n; j+=i)`
      $\longrightarrow$ `for (k=0; k*i<=n; k++)`

      in the loop body: `j` $\longrightarrow$ `k*i`
  - Non-linear loop transformations:
    - Non-linear schedules can substantially improve the performance of solving affine recurrence equations (i.e., of executing loop nests) over linear schedules.
- Challenges:
  - Avoid non-affinities in the dependence analysis;
    postpone them to the code generation
  - Code simplification.

Extension 7: Example




```
for (x=1; x<=4; x++)
  for (y=1; y<=9; y++)
    T1(x,y);
for (x=5; x<=7; x++) {
  for (y=1; y<=⌊4−√(3x − 12)⌋; y++)
    T1(x,y);
  for (y=⌈4+√(3x − 12)⌉; y<=9; y++)
    T1(x,y);
}
```

Extension 7: Cases and Techniques

- Non-Linear Parameters: e.g., p^2`*i`, `p*q*i`, `p*i`
  - LP solution methods like Fourier-Motzkin and Simplex can be generalized to handle several non-linear parameters.
  - Application: tiling and code generation.
  - Math: quantifier elimination in $\mathbb{R}$.
- Also Non-Linear Variables: e.g., $p^2 * i^2$, $p * i^2$, `i*j`
  - Math: Cylindrical algebraic decomposition.
  - Application: Code generation for scanning arbitrary semi-algebraic sets.

The Loop Parallelizer LooPo

- Input:
  - Loop code without parallelism (FORTRAN, C, recurrence equations)
  - Specification of a data flow graph (skip next step)
- Dependence Analysis: transition to the model
  - Method: Banerjee (restricted), Feautrier (complete), control flow fuzzy array dependence analysis (CfFADA, can also handle alternations)
  - Optional: index set splitting, single-assignment conversion
- Space-Time Mapping:
  - Schedule: Lamport (simple), Feautrier (complete), Darte-Vivien (compromise)
  - Allocation: Feautrier (complete), Dion-Robert (more practical), forward-communication only (prepares for tiling)
- Code Generation:
  - Based on the French loop code generator CLooG
  - Generates loops and communication
  - Tiles

Parallel Program Skeletons

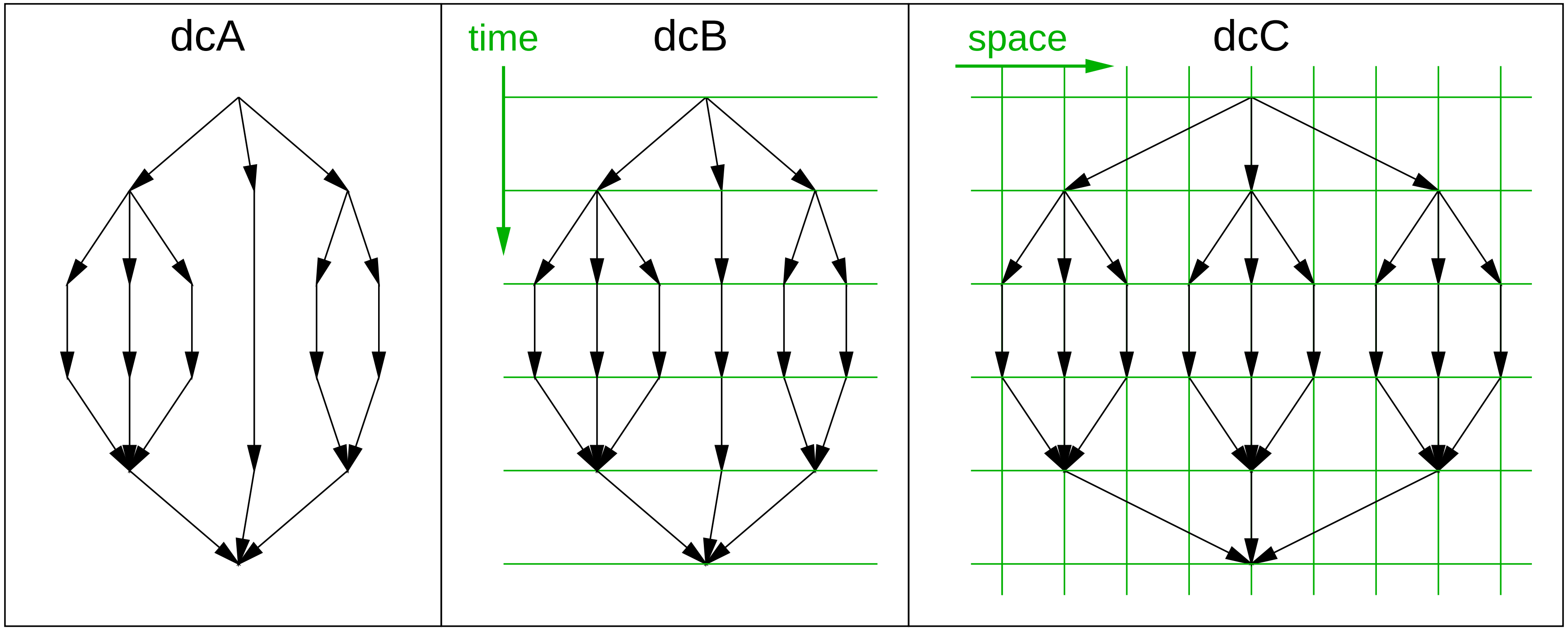
- Idea:
  - Predefine frequently used patterns of parallel computation
  - Specify each pattern as a higher-order function
  - Provide implementations for a variety of parallel platforms
  - Possibly use metaprogramming to make skeletons adaptive
- Examples:
  - Small scale: collective operations
    - data transfer: broadcast, scatter, gather, all-to-all
    - data transfer + computation: reduce, scan
  - Larger scale: algorithmic patterns
    - divide-and-conquer, branch-and-bound
    - dynamic programming, searching in suffix trees
    - algorithms on labelled graphs
- Technique:
  - Functional source language: Template Haskell, MetaML, MetaOCaml
  - Imperative target language: C, C+MPI,...
  - Compilation step: no standard tools so far

Collective Operations

t_s: start-up time t_w: per-word transfer time m: blocksize

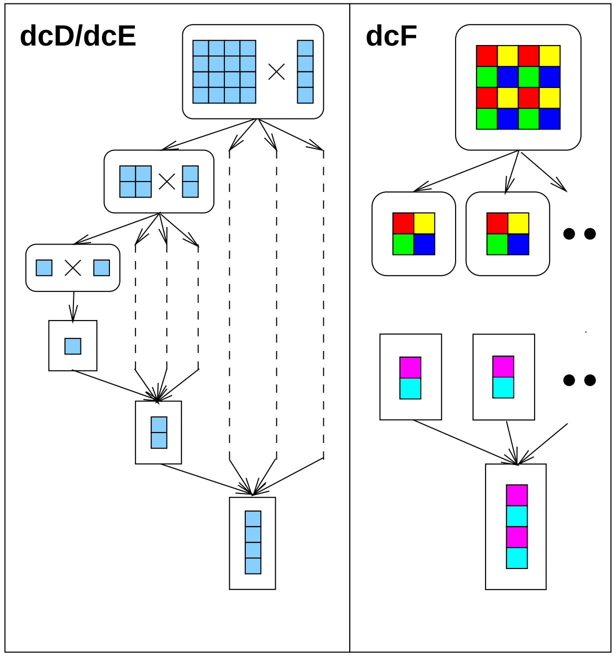
composition rule	improvement if
`Scan_1; Reduce_2` $\rightarrow$ `Reduce`	always
`Scan; Reduce` $\rightarrow$ `Reduce`	$t_s > m$
`Scan_1; Scan_2` $\rightarrow$ `Scan`	$t_s > 2m$
`Scan; Scan` $\rightarrow$ `Scan`	$t_s > m(t_w + 4)$
`Bcast; Scan` $\rightarrow$ `Comcast`	always
`Bcast; Scan_1; Scan_2` $\rightarrow$ `Comcast`	$t_s > \frac{m}{2}$
`Bcast; Scan; Scan` $\rightarrow$ `Comcast`	$t_s > m(\frac{1}{2}t_w + 4)$
`Bcast; Reduce` $\rightarrow$ `Local`	always
`Bcast; Scan_1; Reduce_2` $\rightarrow$ `Local`	always
`Bcast; Scan; Reduce` $\rightarrow$ `Local`	$t_w + \frac{1}{m}t_s \geq \frac{1}{3}$

Divide-and-Conquer Hierarchy: Tasks



skeleton	restriction	application
dcA	independent subproblems	Quicksort, maximum independent set
dcB	fixed recursion depth	*n* queens
dcC	fixed division degree *k*	Karatsuba integer product (*k*=3)

Divide-and-Conquer Hierarchy: Data



dcD	block recursion	triangular matrix inversion (k=2), Batcher sort (k=2)
dcE	elementwise operations	matrix-vektor product (k=4)
dcF	communication between corresponding elements	Karatsuba polynomial product (k=3), bitonic merge (k=2), FFT (k=2), Strassen matrix product (k=7)

Skeleton Implementation

- Principle:
  - specification *recX* = iterative form *itX*
  - transition from Haskell to domain-specific language
- `dcA (base, divide, combine, input)`
  - dynamic allocation of time and space
  - no load balancing
- `dcF (k, indeg, outdeg, basic, divide, combine, n, input)`
  - static allocation of time and space via additional parameters
    - number of subproblems
    - division degree of input data
    - combination degree of output data
    - depth of recursion
  - dependence regular but not affine (no analysis necessary)
  - similar to the polytope model but no search for a schedule
  - symbolic size inference on skeleton parameters

Skeleton Metaprogramming

- Metaprogramming:
  - Using a meta language to transform programs in an object language
  - Both languages can be the same (multi-stage programming)
- Advice:
  - Use a functional metalanguage
  - Model the syntax for the object language by abstract data types
  - Exploit the type structure of the metalanguage for transformations
- Adaptive Libraries:
  - Old approach:
    - Add switch parameters to the library functions to customize
    - Can’t handle “new” cases without reprogramming
    - Caller can provide inconsistent information
  - New approach:
    - Perform an analysis on type and shape of the arguments
    - Provides consistency and flexibility
    - Can reduce abstraction penalty due to a lack of domain-specific knowledge considerably

Conclusions

- How do the methods perform?
  - Automation required high (affine) regularity.
  - Constant number of breaks in regularity can be handled.
  - Non-affinity requires sophisticated mathematics.
  - Code generation very difficult in general; heuristics help.
- Is it for ArtistDesign?
  - Loop parallelization probably only in special cases.
  - Skeletons have high potential – simple or sophisticated.
  - There is experience with tool prototypes.
  - Build dedicated tools.

References

- Basic Polytope Model

  Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

  Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 79–103. Springer-Verlag, 1996.

- Extension 1: Conditionals

  Jean-François Collard and Martin Griebl. A precise fixpoint reaching definition analysis for arrays. In Larry Carter and Jean Ferrante, editors, *Languages and Compilers for Parallel Computing (LCPC'99)*, LNCS 1863, pages 286–302. Springer-Verlag, 1999.

- Extension 2: WHILE Loops

  Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. Parallel Programming*, 23(2):191–219, 1995.

  Martin Griebl. *The Mechanical Parallelization of Loop Nests Containing* while *Loops*. PhD thesis, University of Passau, 1996. Also available as technical report MIP-9701.

- Extension 3: Index Set Splitting

  Martin Griebl, Paul Feautrier, and Christian Lengauer. Index set splitting. *Int. J. Parallel Programming*, 28(6):607–631, 2000.

References

- Extension 4: Tiling

  Martin Griebl, Peter Faber, and Christian Lengauer. Space-time mapping and tiling: A helpful combination. *Concurrency and Computation: Practice and Experience*, 16(3):221–246, March 2004.

  U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. PLUTO: A practical and fully automatic polyhedral program optimization system. *Proc. ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation (PLDI 2008)*, ACM Press, 2008.

- Extension 5: Expressions

  Peter Faber. *Code Optimization in the Polyhedron Model – Improving the Efficiency of Parallel Loop Nests*. PhD thesis, University of Passau, 2008.

- Extension 6: Non-Affine Index Array Expressions

  Stefan Schuster. On algorithmic and heuristic approaches to integral problems in the polyhedron model with non-linear parameters. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 2007.

- Extension 7: Non-Affine Loop Bounds

  Armin Größlinger, Martin Griebl, and Christian Lengauer. Quantifier elimination in automatic loop parallelization. *Journal of Symbolic Computation*, 41(11):1206–1221, November 2006.

  Armin Größlinger. Scanning index sets with polynomial bounds using cylindrical algebraic decomposition. Technical report MIP-0803, Fakultät für Informatik und Mathematik, Universität Passau.

References

- Small-Scale Skeletons: Collective Operations

  Sergei Gorlatch. Send-receive considered harmful: Myths andc realities of message passing. *ACM TOPLAS*, 26(1):47–56, 2004.
- Large-Scale Skeletons: Divide-and-Conquer

  Christoph Armin Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*. PhD thesis, University of Passau, 2000. ISBN 3-89722-556-5.
- Metaprogrammed Skeletons

  Christoph Armin Herrmann and Christian Lengauer. Using metaprogramming to parallelize functional specifications. *Parallel Processing Letters*, 12(2):93–210, June 2002.