LooPo: Automatic Loop Parallelization

Michael Claßen

Fakultät für Informatik und Mathematik

UNIVERSITÄT PASSAU

Düsseldorf, November 27th 2008

Model-Based Loop Transformations

- model-based approach:
  - map source code to an execution model
  - find the optimal parallel solution in this model
- advantages:

  + quality metric: a given objective function

  + search and transformation completely automatic
- disadvantages:

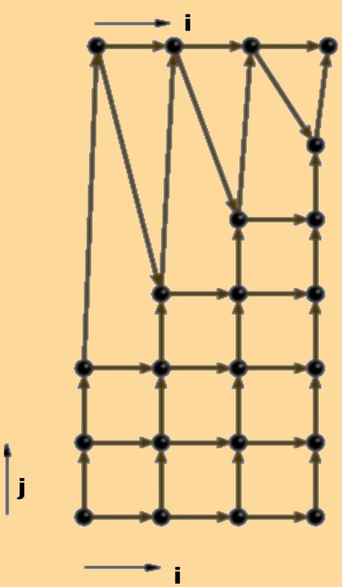
  – analysis and target code can become complex

  – optimality in the model need not imply efficient target code

The Polytope Model

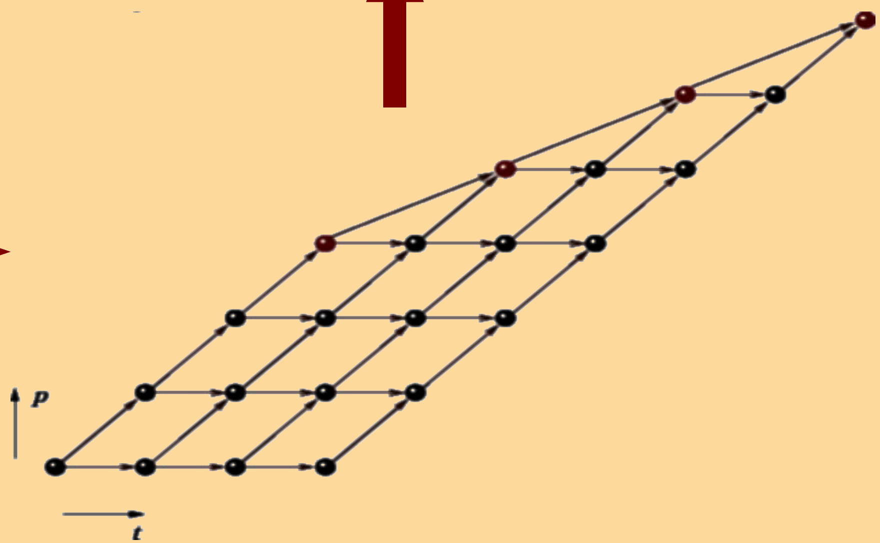
- input program:

```
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
```



- target code:

```
parfor p = 1 to m + n do
  for t = max(p-1, 2p-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 >= m+1 then
    A(p-m, 1+p) = A(p-m, p) +
                  A(p-m-1, p)
  fi
od
```



Overview: Parallelization Steps

- scanning / parsing
- dependence analysis
- space-time transformation:
  - schedule
  - allocation (placement)
- adapting granularity of parallelism (*tiling*)
- target code generation
- (post-processing / scripts)

Scanner / Parser

- input languages:
  - loop nests in C-like syntax (C/C++, Java), Fortran
  - specification language for dependences
- result:
  - abstract syntax tree + polytope description
- analyzes:
  - affine linear expressions in loop-bounds

    → polytope description per statement
  - affine linear expressions in array accesses:

    → affine linear function per array access

Dependences

- dependence:
  - different operations access same memory cell
  - first access: source
  - second access: destination
  - read / write access → four dependence types:
    - write, read: true
    - read, write: anti
    - write, write: output
    - (read, read: input)
  - flow dependences: optimized true dependences
  - uniform / non-uniform

Dependence Analysis

- analysis on polytope model representation
- implemented methods:
  - Banerjee (restricted)
  - Feautrier (more general)
  - control flow fuzzy (CfFADA), can handle alternations
- result:
  - polytope description of dependences
    (access pairs causing memory conflict)

Space-Time Transformation

- goals:
  - maximize parallel iterations
  - minimize sequential time steps
  - additional restrictions for communication / tiling
  - if possible: simple computation, reduced code complexity
- schedule: maps operation to (virtual) execution time step
- allocation: maps operation to (virtual) processor
- result: affine linear space-time transformation function
- multi-dimensional mappings per statement possible

Schedulers

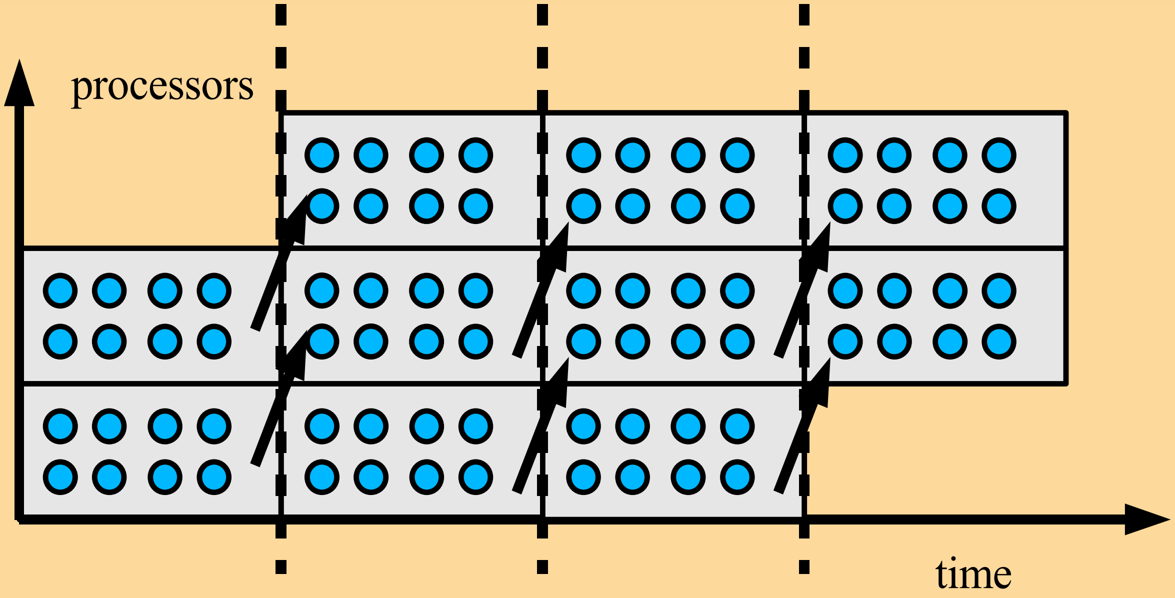
- hyperplane method by Lamport:
  - idea: construct one loop that carries all dependences
  - simple, but only uniform dependences
- Feautrier scheduler:
  - minimize latency: t(dest) – t(src)
  - can handle non-uniform dependences
  - more complex, but better results
- Darte-Vivien scheduler:
  - can handle non-uniform dependences
  - faster compromise, but (in theory) limited results

Allocators

- Feautrier allocator:
  - try to place source and destination on same processor
  - minimize communication cost, maximize parallelism
  - latest write access determines array element placement
- Dion-Robert allocator:
  - fast, practical
  - allows data placement: array elements on fixed processors (HPF)
- FCO:
  - restriction: ”forward communication only” (FCO):
    - positive direction vector of dependences in space dimensions
    - avoids deadlocks in time tiling

Tiling

- techniques deliver high degree of parallelism
- but: often too fine-grained
- idea: aggregate operations into larger chunks (tiles)
- communicate only between tiles



Space / Time Tiling

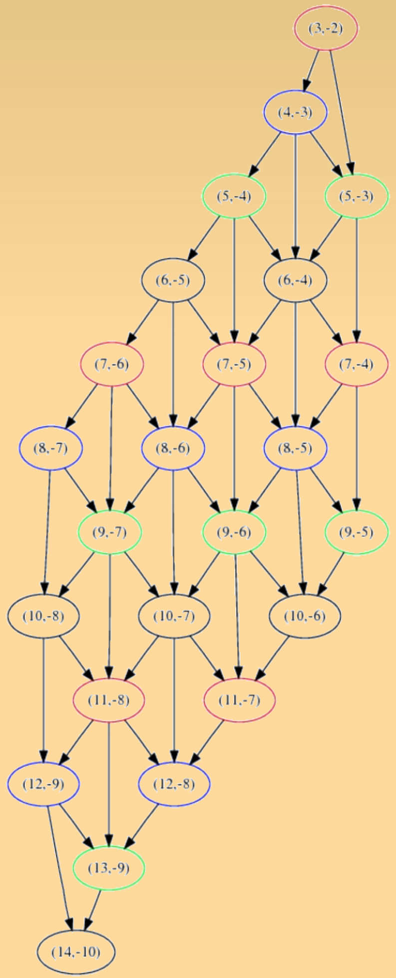
- aggregate virtual processors → space tiling
  - additional step: map space tiles to processors
- aggregate virtual execution steps → time tiling:
  - ”atomic” execution within time tiles
  - communication between time tiles

Code Generation

- generate loops (→ CLooG)
- include parallelization constructs
- targets:
  - distributed memory: requires communication code!
    - PC clusters
    - Grid
  - shared memory:
    - multi-core processors (OpenMP)
  - general-purpose computing on graphics processing units:
    - shared and distributed memory aspects

Distributed Memory

- inter-processor communication
- for clusters:
  - map virtual processors / tiles on processor nodes
  - use MPI send / receive or collective operations for exchanging data
- for Grid:
  - use HOC-SA middleware
  - adapt a taskfarming component for communication between task
  - create task graph: tiles + inter-task dependences
  - challenges: scalability, memory usage!



Shared Memory

- generate (synchronous) loop nest
- using OpenMP
- annotate code: parallel for loops are marked
- example (SOR):

```
for (glT1=-1;glT1<=floord(3*M+N-5,100);glT1++) {
  #pragma omp parallel for private(vT1,vP1,vP2,tileT1,tileP1,other1,other2,K,I)
  for (rp1=max(max(ceild(100*glT1-N-193,300),0),ceild(100*glT1-2*M-N+5,100));
       rp1<=min(min(floord(M,100),floord(100*glT1+101,300)),
                floord(100*glT1+99,100));
       rp1++) {
    for (vT1=...;vT1<=...;vT1++) {
      for (vP1=...;vP1<=...;vP1++) {
        ...
          K = other1; I = other2; A[I]=(A[1+I]+A[I-1])/2;
      }
    ...
}
```

Computing on GPUs

- using graphic cards for HPC
- interface for general-purpose computing:
  - CUDA (Compute Unified Device Architecture)
- generating loops using LooPo
- memory aspects:
  - shared global memory, slow
  - local scratchpad memory, fast but small (16 KB)
  - challenge: optimize usage of scratchpad memory

Example: SOR

- successive over-relaxation (SOR):
- used in algorithms to speed up convergence
- typically used in Gauss-Seidel method

```
DO K=1,M
   DO I=2,N-1
       A(I)=(A(I-1)+A(I+1))/2.0
   END DO
END DO
```

SOR

- only uniform dependences
- Lamport scheduler possible
- also possible: Feautrier, Darte-Vivien
- different tiling choices: rectangular, parallelogram
- speedup: up to 3.9 on 4 cores (97.4% efficiency)
- target code: inner loop optimized to 8 assembler instructions

LU backward substitution

- second part of LU decomposition

```
DO k1=0, n-1
   sum[(n-k1-1)] = U[(n-k1-1)];
END DO
DO k=0, n-1
   DO l=0, k-1
      sum[(n-k-1)]=sum[(n-k-1)]-a[(n-k-1)][(n-l-1)]*U[(n-l-1)];
   END DO
   U[(n-k-1)]=sum[(n-k-1)]/a[(n-k-1)][(n-k-1)];
END DO
```

LU backward substitution

- non-uniform dependences
- allocator results:
  - Feautrier: fully dimensional placement for all statements
  - Dion-Robert: constant placement for statement #3

Cholesky decomposition

- a symmetric positive-definite matrix is decomposed into:
  - lower triangular matrix and
  - transpose of the lower triangular matrix
- used to solve systems of linear equations

```
do k=1,n
   do j=1,k-1
      a(k,k) = a(k,k) - a(k,j)*a(k,j)
   end do
   a(k,k) = sqrt(a(k,k))
   do i=k+1,n
      do j=1,k-1
         a(i,k) = a(i,k) - a(i,j)*a(k,j)
      end do
      a(i,k)=a(i,k)/a(k,k)
   end do
end do
```

Cholesky decomposition

- complicating factors:
  - imperfectly nested loop nest
  - non-uniform dependences (→ no Lamport)
  - use of function sqrt:

```
float FUNCTION sqrt(x)

  FLOAT, INTENT(IN) :: x
END FUNCTION sqrt
```

Kernel19 Livermore

- Livermore fortran kernels
- kernel19: general linear recurrence equations

```
DO k= 1,n
  B5[k]= SA[k] +STB5*SB[k]
  STB5= B5[k] -STB5
END DO
DO i= 1,n
  B5[n-i+1]= SA[n-i+1] +STB5*SB[n-i+1]
  STB5= B5[n-i+1] -STB5
END DO
```