Liveness Analysis in Explicitly-Parallel Programs

Alain Darte

With Alexandre Isoard, Paul Feautrier, Tomofumi Yuki

- Lattice-Based Memory Allocation. ☛ Polyhedral interferences, admissible lattices
- Register Allocation: What Does the NP-Completeness of Chaitin et al. Really Prove? ☛ Chordal interf.
- SSI Properties revisited. ☛ From SSA to SSI: intervals
- Exact & Approximated Data-Reuse Opt. for Tiling with Parametric Sizes. ☛ Pipelining, complex interf.
- Static Analysis of OpenStream Programs. ☛ More parallel specifications, deadlocks, polynomials
- Liveness Analysis in Explicitly-Parallel Programs. ☛ Generalization of interferences
- Extended Lattice-Based Memory Allocation. ☛ Generalization of memory mapping, union of polyhedra

Meeting in Discrete Structures 1, LIP
Villemanzy, December 17, 2015.

Parallel languages, runtime execution, and static analysis

Solution(s) for high-level parallel programming?

- Static or dynamic?
- Language constructs or libraries?
- Expressiveness: deterministic (no data races) or deadlock-free?
- How to represent communications and memories? Concurrency?
- Can static optimization help runtime optimizations?
  Worst-case, buffer sizes, granularity opt., mapping, locality, ...

Parallel languages, runtime execution, and static analysis

Solution(s) for high-level parallel programming?

- Static or dynamic?
- Language constructs or libraries?
- Expressiveness: deterministic (no data races) or deadlock-free?
- How to represent communications and memories? Concurrency?
- Can static optimization help runtime optimizations?
  Worst-case, buffer sizes, granularity opt., mapping, locality, ...

Many approaches:

- "Lower"-level: MPI, OpenCL, Lime, ...
- Runtime-based: Kaapi, StarPU (with task dep. as in OpenMP 4.0).
- (A)PGAS languages: Co-Array Fortran, UPC, Chapel, X10, ...
- "Dataflow" languages: KPN, SDF, CSDF, SigmaC, OpenStream, ...
- Automatic compilation schemes.

Discrete structures

- Polyhedral representations, Presburger formulas, integer sets.
- Linear programming, Farkas lemma, polynomial generalizations.
- Graph structures: chordal graphs, interval graphs, comparability graphs, serie-parallel graphs.
- Integer lattices, basis reduction, Hermite/Smith forms.
- NP-completeness, undecidability, Hilbert's 10th problem.

☛ Dependence analysis, liveness analysis, deadlock detection, while loop termination, upper/lower bounds for time & memory, scheduling, etc.

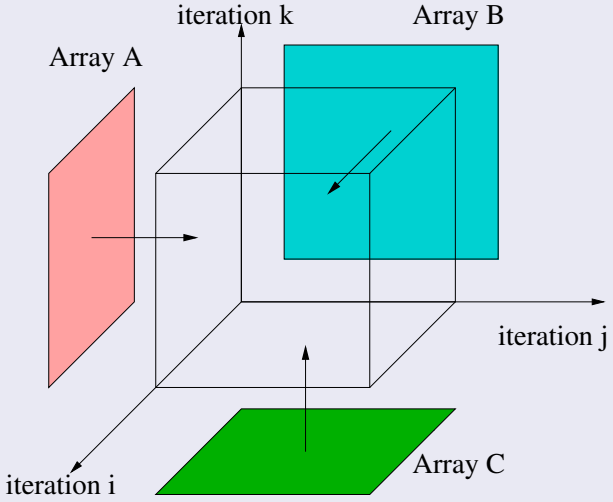
Outline

1. Polyhedral representation examples
   - Compilation for GPU, with shared-memory optimization
   - Tiling with automatic double-buffering, transfers and buffer sizes
2. Exploring different forms of parallelism
   - Analysis of a X10 subset
   - Analysis of an OpenStream subset
3. Liveness analysis
   - Chordal and interval graphs for SSA and SSI
   - Comparability graphs for partial orders
4. Lattice-based memory allocation
   - Polyhedral conflicts
   - Conflicts as union of polyhedra

Multi-dimensional affine representation of loops and arrays

Matrix Multiply

```
int i,j,k;
for(i = 0; i < n; i++) {
   for(j = 0; j < n; j++) {
S:     C[i][j] = 0;
       for(k = 0; k < n; k++) {
T:         C[i][j] += A[i][k] * B[k][j];
       }
   }
}
```



Polyhedral Description — Omega/ISCC syntax

```
Domain := [n]->{S[i][j]: 0<=i,j<n; T[i][j][k]: 0<=i,j,k<n};

Read := [n]->{T[i][j][k]->A[i][k]; T[i][j][k]->B[k][j];
              T[i][j][k]->C[i][j]};

Write := [n]->{S[i][j]->C[i][j]; T[i][j][k]->C[i][j]};

Order := [n]->{S[i][j]->[i][j][0]; T[i][j][k]->[i][j][1][k]};
```

PPCG code for CPU+GPU: GPU part

```
__global__ void kernel0(float *A, float *B, float *C, int n) /* n=12288 */
{
    int b0 = blockIdx.y, b1 = blockIdx.x; /* Grid: 192x192 blocks, each with 32x32 threads */
    int t0 = threadIdx.y, t1 = threadIdx.x; /* Loops: 384x384x768 tiles, each with 32x32x16 points */
    __shared__ float shared_A[32][16]; /* Thus 1 block = 2x2x768 tiles, 1 thread = 1x1x16 points */
    __shared__ float shared_B[16][32];
    float private_C[1][1];

    for (int g1 = 32 * b0; g1 <= 12256; g1 += 6144) /* 6144 = 32 (tile size) x 192 (number of blocks) */
      for (int g3 = 32 * b1; g3 <= 12256; g3 += 6144) { /* 32 is the tile size */
        private_C[0][0] = C[(t0 + g1) * 12288 + (t1 + g3)];
        for (int g9 = 0; g9 <= 12272; g9 += 16) { /* 16 consecutive points along k in a thread */
          if (t0 <= 15) /* 32x32 threads, only 16x32 do the transfer */
            shared_B[t0][t1] = B[(t0 + g9) * 12288 + (t1 + g3)];
          if (t1 <= 15) /* 32 threads, only 32x16 do the transfer */
            shared_A[t0][t1] = A[(t0 + g1) * 12288 + (t1 + g9)];
          __syncthreads();
          for (int c4 = 0; c4 <= 15; c4 += 1) /* compute the 16 consecutive points along k */
            private_C[0][0] += (shared_A[t0][c4] * shared_B[c4][t1]);
          __syncthreads();
        }
        C[(t0 + g1) * 12288 + (t1 + g3)] = private_C[0][0];
        __syncthreads();
      }
}
```

PPCG code for CPU+GPU: Verdoolaege, Cohen, etc.

PPCG code for CPU+GPU: GPU part (Volkov-like)

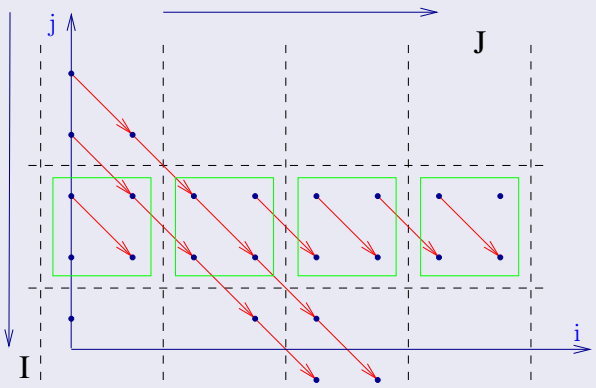
```
__global__ void kernel0(float *A, float *B, float *C, int n) /* n=12288 */
{
    int b0 = blockIdx.y, b1 = blockIdx.x; /* Grid: 192x192 blocks, each with 16x16 threads */
    int t0 = threadIdx.y, t1 = threadIdx.x; /* Loops: 384x384x768 tiles, each with 32x32x16 points */
    __shared__ float shared_A[32][16]; /* Thus 1 block = 2x2x768 tiles, 1 thread = 2x2x16 points */
    __shared__ float shared_B[16][32];
    float private_C[2][2];

    for (int g1 = 32 * b0; g1 <= 12256; g1 += 6144) /* 6144 = 32 (tile size) x 192 (number of blocks) */
      for (int g3 = 32 * b1; g3 <= 12256; g3 += 6144) { /* 32 is the tile size */
        private_C[0][0] = C[(t0 + g1) * 12288 + (t1 + g3)]; /* 2x2 points unrolled for register usage */
        private_C[0][1] = C[(t0 + g1) * 12288 + (t1 + g3 + 16)];
        private_C[1][0] = C[(t0 + g1 + 16) * 12288 + (t1 + g3)];
        private_C[1][1] = C[(t0 + g1 + 16) * 12288 + (t1 + g3 + 16)];
        for (int g9 = 0; g9 <= 12272; g9 += 16) { /* 16 consecutive points along k in a thread */
          for (int c1 = t1; c1 <= 31; c1 += 16) /* 16x32 to bring with 16x16 threads */
            shared_B[t0][c1] = B[(t0 + g9) * 12288 + (g3 + c1)];
          for (int c0 = t0; c0 <= 31; c0 += 16) /* 32x16 to bring with 16x16 threads */
            shared_A[c0][t1] = A[(g1 + c0) * 12288 + (t1 + g9)];
          __syncthreads();
          for (int c2 = 0; c2 <= 15; c2 += 1) { /* unrolled for register usage */
            private_C[0][0] += (shared_A[t0][c2] * shared_B[c2][t1]);
            private_C[0][1] += (shared_A[t0][c2] * shared_B[c2][t1 + 16]);
            private_C[1][0] += (shared_A[t0 + 16][c2] * shared_B[c2][t1]);
            private_C[1][1] += (shared_A[t0 + 16][c2] * shared_B[c2][t1 + 16]);
          }
          __syncthreads();
        }
        C[(t0 + g1) * 12288 + (t1 + g3)] = private_C[0][0];
        C[(t0 + g1) * 12288 + (t1 + g3 + 16)] = private_C[0][1];
        C[(t0 + g1 + 16) * 12288 + (t1 + g3)] = private_C[1][0];
        C[(t0 + g1 + 16) * 12288 + (t1 + g3 + 16)] = private_C[1][1];
        __syncthreads();
      }
```

PPCG code for CPU+GPU: GPU part with ILP (Volkov)

Parametric tiling with double buffering

Parameter n, tiles of size $b \times b$.

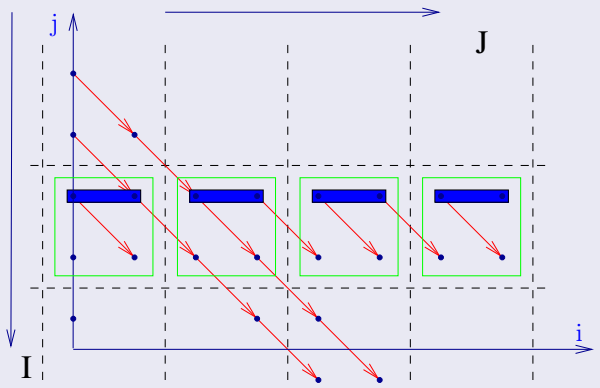



```
int i,j;
for(i = 0; i < n; i++) {
   for(j = 0; j < n; j++) {
     C[i+j] = C[i+j] + A[i]*B[j];
   }
}
```

Sets Load_A, Load_B, Load_C, Store_C?

Parametric tiling with double buffering

Parameter n, tiles of size $b \times b$.




```
int i,j;
for(i = 0; i < n; i++) {
   for(j = 0; j < n; j++) {
     C[i+j] = C[i+j] + A[i]*B[j];
   }
}
```

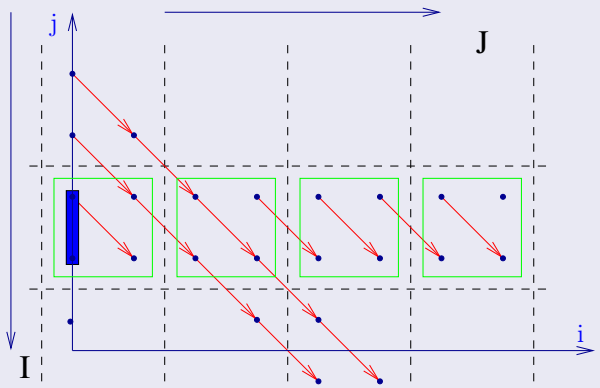
Sets Load_A, Load_B, Load_C, Store_C?

$\mathrm{Load}_A = \{m \mid 0 \leq m \leq n-1,\ J \leq m \leq J+b-1\}$

- size $2b$, when $n \geq 2b+1$: at least 2 tiles available.
- size n when $n \leq 2b$: less than 2 tiles.

Parametric tiling with double buffering

Parameter n, tiles of size $b \times b$.




```
int i,j;
for(i = 0; i < n; i++) {
   for(j = 0; j < n; j++) {
     C[i+j] = C[i+j] + A[i]*B[j];
   }
}
```

Sets Load_A, Load_B, Load_C, Store_C?

$\mathrm{Load}_A = \{m \mid 0 \leq m \leq n-1,\ J \leq m \leq J+b-1\}$

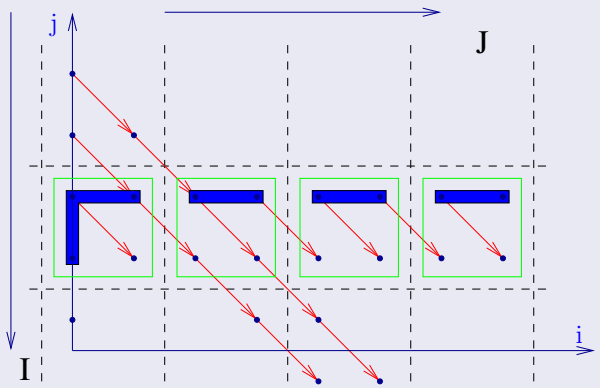
- size $2b$, when $n \geq 2b+1$: at least 2 tiles available.
- size n when $n \leq 2b$: less than 2 tiles.

$\mathrm{Load}_B = \{m \mid J = 0,\ 0 \leq m \leq n-1,\ n-I-b \leq m \leq n-I-1\}$

- size b when $n \geq b$: 1 full tile.
- size n when $n \leq b-1$: 1 partial tile.

Parametric tiling with double buffering

Parameter n, tiles of size $b \times b$.




```
int i,j;
for(i = 0; i < n; i++) {
   for(j = 0; j < n; j++) {
     C[i+j] = C[i+j] + A[i]*B[j];
   }
}
```

Sets Load_A, Load_B, Load_C, Store_C?

$\mathrm{Load}_A = \{m \mid 0 \leq m \leq n-1,\ J \leq m \leq J+b-1\}$

- size $2b$, when $n \geq 2b+1$: at least 2 tiles available.
- size n when $n \leq 2b$: less than 2 tiles.

$\mathrm{Load}_B = \{m \mid J = 0,\ 0 \leq m \leq n-1,\ n-I-b \leq m \leq n-I-1\}$

- size b when $n \geq b$: 1 full tile.
- size n when $n \leq b-1$: 1 partial tile.

$\mathrm{Load}_C = \{m \mid 0 \leq m,\ n-I-b \leq m \leq n-1-I,\ J = 0\}$
$\quad \cup \{m \mid \max(1, J) \leq m+I-n+1 \leq \min(n-1, J+b-1)\}$

- size $3b-1 = (2b-1)+b$ when $n \geq 2b+1$: 2 full tiles.
- size $b+n-1 = (2b-1)+(n-b)$ when $b \leq n \leq 2b$: 1 full tile, 1 partial tile.
- size $2n-1$ when $n \leq b-1$: 1 partial tile.

Outline

Sequential and parallel loops

```
for(i=0; i<n; i++) {
   for(j=0; j<n; j++) {
S:    ...
T:    ...
   }
}
```

- Total order $\prec$ defined by a sequential schedule σ and lexicographic order.
- $\sigma(S(i,j)) = (i,j,0)$, $\sigma(T(i,j)) = (i,j,1)$.
- $O \prec O'$ iff $\sigma(O) <_{\text{lex}} \sigma(O')$.
- $S(i,j) \prec T(i',j')$ iff $i < i'$ or $(i = i'$ and $j \leq j')$.

Sequential and parallel loops

```
for(i=0; i<n; i++) {
   for(j=0; j<n; j++) {
S:    ...
T:    ...
   }
}
```

- Total order $\prec$ defined by a sequential schedule σ and lexicographic order.
- $\sigma(S(i,j)) = (i,j,0)$, $\sigma(T(i,j)) = (i,j,1)$.
- $O \prec O'$ iff $\sigma(O) <_{\text{lex}} \sigma(O')$.
- $S(i,j) \prec T(i',j')$ iff $i < i'$ or $(i = i'$ and $j \leq j')$.

```
for(i=0; i<n; i++) {
  forpar(j=0; j<n; j++) {
S:    ...
T:    ...
   }
}
```

- Partial order $\prec$, some form of lexicographic order.
- $S(i,j) \prec T(i',j')$ iff $i < i'$ or $(i = i'$ and $j = j')$.

Sequential and parallel loops

```
for(i=0; i<n; i++) {
   for(j=0; j<n; j++) {
S:    ...
T:    ...
   }
}
```

- Total order $\prec$ defined by a sequential schedule σ and lexicographic order.
- $\sigma(S(i,j)) = (i,j,0)$, $\sigma(T(i,j)) = (i,j,1)$.
- $O \prec O'$ iff $\sigma(O) <_{\text{lex}} \sigma(O')$.
- $S(i,j) \prec T(i',j')$ iff $i < i'$ or $(i = i'$ and $j \leq j')$.

```
for(i=0; i<n; i++) {
  forpar(j=0; j<n; j++) {
S:    ...
T:    ...
   }
}
```

- Partial order $\prec$, some form of lexicographic order.
- $S(i,j) \prec T(i',j')$ iff $i < i'$ or $(i = i'$ and $j = j')$.

```
forpar(i = 0; i < n; i++) {
   for(j = 0; j < n; j++) {
S:    ...
T:    ...
   }
}
```

- Partial order $\prec$, some form of lexicographic order.
- $S(i,j) \prec T(i',j')$ iff $(i = i'$ and $j \leq j')$.

Analyzing X10 through a polyhedral fragment

X10 language developed at IBM, variant at Rice (V. Sarkar)

- PGAS (partitioned global address space) memory principle.
- Parallelism of threads: in particular keywords finish, async, clock.
- No deadlocks by construction but non-determinism.

Polyhedral X10 Yuki, Feautrier, Rajopadhye, Saraswat (PPoPP 2013)

Can we analyze the code for data races?

```
finish {
  for(i in 0..n-1) {
    S1;
    async {
      S2;
    }
  }
}
```

```
clocked finish {
  for(i in 0..n-1) {
    S1; advance();
    clocked async {
      S2; advance();
    }
  }
}
```

Analyzing X10 through a polyhedral fragment

X10 language developed at IBM, variant at Rice (V. Sarkar)

- PGAS (partitioned global address space) memory principle.
- Parallelism of threads: in particular keywords finish, async, clock.
- No deadlocks by construction but non-determinism.

Polyhedral X10 Yuki, Feautrier, Rajopadhye, Saraswat (PPoPP 2013)

Can we analyze the code for data races?

```
finish {
  for(i in 0..n-1) {
    S1;
    async {
      S2;
    }
  }
}
```

```
clocked finish {
  for(i in 0..n-1) {
    S1; advance();
    clocked async {
      S2; advance();
    }
  }
}
```

Yes. Similar to data-flow analysis. Partial order $\prec$: incomplete lexicographic order.

Analyzing X10 through a polyhedral fragment

X10 language developed at IBM, variant at Rice (V. Sarkar)

- PGAS (partitioned global address space) memory principle.
- Parallelism of threads: in particular keywords finish, async, clock.
- No deadlocks by construction but non-determinism.

Polyhedral X10 Yuki, Feautrier, Rajopadhye, Saraswat (PPoPP 2013)

Can we analyze the code for data races?

```
finish {
  for(i in 0..n-1) {
    S1;
    async {
      S2;
    }
  }
}
```

Yes. Similar to data-flow analysis. Partial order $\prec$: incomplete lexicographic order.

```
clocked finish {
  for(i in 0..n-1) {
    S1; advance();
    clocked async {
      S2; advance();
    }
  }
}
```

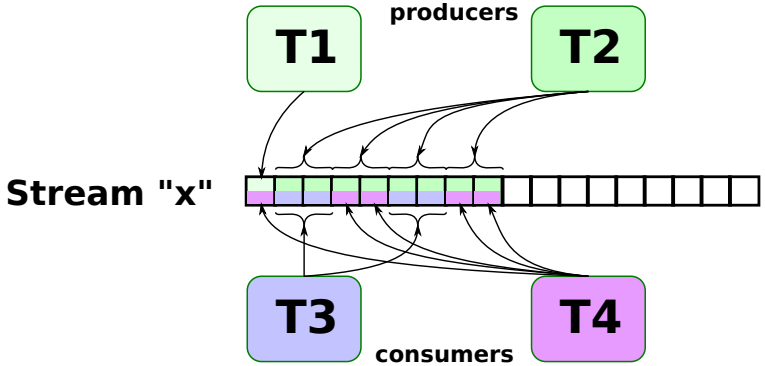
Undecidable. Partial order $\prec_c$ defined by $\vec{x} \prec_c \vec{y}$ iff $\vec{x} \prec \vec{y}$ or $\phi(\vec{x}) < \phi(\vec{y})$. $\phi(\vec{x}) = \#$ advances before (for $\prec$) $\vec{x}$.

Analyzing OpenStream through a polyhedral fragment

```
#pragma omp task output (x) // Task T1
x = ...;
for (i = 0; i < N; ++i) {
  int window_a[2], window_b[3];

  #pragma omp task output (x « window_a[2]) // Task T2
  window_a[0] = ...; window_a[1] = ...;
  if (i % 2) {
    #pragma omp task input (x » window_b[2]) // Task T3
    use (window_b[0], window_b[1]);
  }
  #pragma omp task input (x) // Task T4
  use (x);
}
```

(Pop, Cohen, 2011)



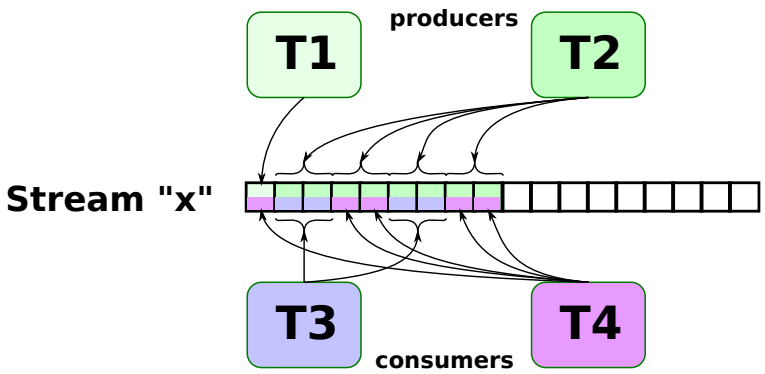
- Sequential control program for task activations.
- Reservation for reads/writes in streams with burst and horizon.
- Single assignment in streams (by construction) + dataflow semantics.

Analyzing OpenStream through a polyhedral fragment

```
#pragma omp task output (x) // Task T1
x = ...;
for (i = 0; i < N; ++i) {
  int window_a[2], window_b[3];

  #pragma omp task output (x « window_a[2]) // Task T2
  window_a[0] = ...; window_a[1] = ...;
  if (i % 2) {
    #pragma omp task input (x » window_b[2]) // Task T3
    use (window_b[0], window_b[1]);
  }
  #pragma omp task input (x) // Task T4
  use (x);
}
```

(Pop, Cohen, 2011)



- Sequential control program for task activations.
- Reservation for reads/writes in streams with burst and horizon.
- Single assignment in streams (by construction) + dataflow semantics.

- Unlike KPN, streams with multiple inputs/outputs (but deterministic).
- If a schedule exists with bounded streams, such sizes can be enforced by blocking R/W, without creating deadlocks at runtime.
- Deadlock detection is undecidable (encoding of polynomials again).

Outline

1. Polyhedral representation examples
   - Compilation for GPU, with shared-memory optimization
   - Tiling with automatic double-buffering, transfers and buffer sizes
2. Exploring different forms of parallelism
   - Analysis of a X10 subset
   - Analysis of an OpenStream subset
3. Liveness analysis
   - Chordal and interval graphs for SSA and SSI
   - Comparability graphs for partial orders
4. Lattice-based memory allocation
   - Polyhedral conflicts
   - Conflicts as union of polyhedra

Liveness analysis

Uses of liveness analysis:

- Necessary for memory reuse:
  - Register allocation: interference graph.
  - Array contraction: conflicting relations.
  - Even wire usage: bitwidth analysis.
- Important information for:
  - Communication: live-in/live-out sets (inlining, offloading)
  - Memory footprint (e.g., for cache prediction)
  - Lower/upper bounds on memory usage.

Liveness analysis

Uses of liveness analysis:

- Necessary for memory reuse:
  - Register allocation: interference graph.
  - Array contraction: conflicting relations.
  - Even wire usage: bitwidth analysis.
- Important information for:
  - Communication: live-in/live-out sets (inlining, offloading)
  - Memory footprint (e.g., for cache prediction)
  - Lower/upper bounds on memory usage.

Several variants:

- Value-based or memory-based analysis.
- Liveness sets or interference graphs.
- Control flow graphs (CFG): basic blocks, SSA, SSI, etc.
- Task graphs, parallel specifications: not really explored so far.

Array contraction: symbolic unrolling, analysis, mapping

```
x = ...;
y = x + ...;
... = y;
```

$\Rightarrow$

```
x = ...;
x = x + ...;
... = x;
```

Array contraction: symbolic unrolling, analysis, mapping

```
c[0] = 0;
for(i=0; i<n; i++) {
  c[i+1] = c[i] + ...;
}
```

$\Rightarrow$

```
c = 0;
for(i=0; i<n; i++) {
  c = c + ...;
}
```

Array contraction: symbolic unrolling, analysis, mapping

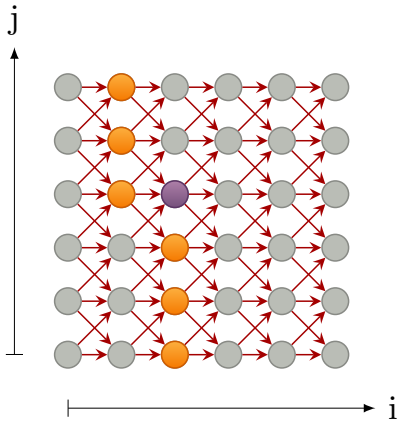
```
c[0] = 0;
for(i=0; i<n; i++) {
  c[i+1] = c[i] + ...;
}
```

$\Rightarrow$

```
c = 0;
for(i=0; i<n; i++) {
  c = c + ...;
}
```

```
for(i=0; i<n; ++i) {
  for(j=0; j<n; ++j) {
     A[i][j] = A[i-1][j-1] +
        A[i-1,j] + A[i-1,j+1];
  }
}
```

Mapping: `a[i][j]` $\mapsto$ `a[(j-i)%(n+1)]`



Array contraction: symbolic unrolling, analysis, mapping

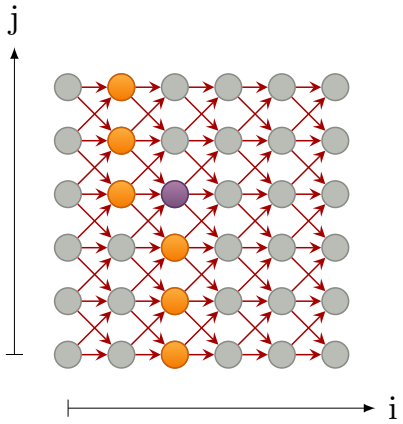
```
c[0] = 0;
for(i=0; i<n; i++) {
  c[i+1] = c[i] + ...;
}
```

$\Rightarrow$

```
c = 0;
for(i=0; i<n; i++) {
  c = c + ...;
}
```

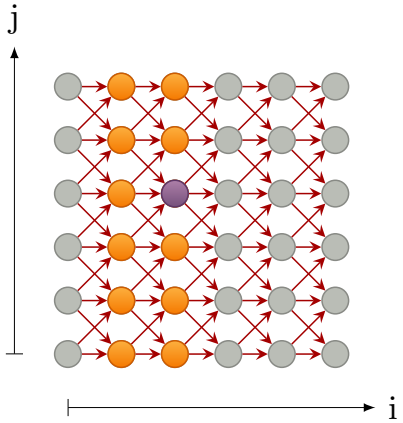
```
for(i=0; i<n; ++i) {
  for(j=0; j<n; ++j) {
     A[i][j] = A[i-1][j-1] +
        A[i-1,j] + A[i-1,j+1];
  }
}
```

Mapping: a[i][j] $\mapsto$ a[(j-i)%(n+1)]




```
for(i=0; i<n; ++i) {
  forpar(j=0; j<n; ++j) {
     A[i][j] = A[i-1][j-1] +
        A[i-1,j] + A[i-1,j+1];
  }
}
```

Mapping: a[i][j] $\mapsto$ a[i%2][j]



Control-flow graphs and interferences

Basic blocks, no hole or single write

- Interference graph = interval graph.
- Linear cliques = live sets at a program point, maxlive.
- Linear-time allocation.

Control-flow graphs and interferences

Basic blocks, no hole or single write

- Interference graph = interval graph.
- Linear cliques = live sets at a program point, maxlive.
- Linear-time allocation.

General control-flow graph

- Chaitin coloring NP-completeness.
- Fixed-point computations for liveness sets.
- Special cases for reducible graphs (backwards).
- Bounded tree-width for some languages.

Control-flow graphs and interferences

Basic blocks, no hole or single write

- Interference graph = interval graph.
- Linear cliques = live sets at a program point, maxlive.
- Linear-time allocation.

General control-flow graph

- Chaitin coloring NP-completeness.
- Fixed-point computations for liveness sets.
- Special cases for reducible graphs (backwards).
- Bounded tree-width for some languages.

Static single assignment (SSA) with dominance

- Interference graph = chordal graph.
- Clique max = live set at a control point.
- Liveness sets computation without fix-point (2 passes).
- Linear-time algorithms for coloring.

Control-flow graphs and interferences

Basic blocks, no hole or single write

- Interference graph = interval graph.
- Linear cliques = live sets at a program point, maxlive.
- Linear-time allocation.

General control-flow graph

- Chaitin coloring NP-completeness.
- Fixed-point computations for liveness sets.
- Special cases for reducible graphs (backwards).
- Bounded tree-width for some languages.

Static single assignment (SSA) with dominance

- Interference graph = chordal graph.
- Clique max = live set at a control point.
- Liveness sets computation without fix-point (2 passes).
- Linear-time algorithms for coloring.

Static single information (SSI) with dominance

- Interference graph = interval graph, proof is not obvious.
- Liveness sets computation = one linear-time pass.

Liveness at a given "step" with iscc

```
# Inputs
Params := [n] -> { : n >= 0 };
Domain := [n] -> { S[i,j] : 0 <= i, j < n };
Read := [n] -> { S[i,j] -> A[i-1,j-1]; S[i,j] -> A[i-1,j];
                 S[i,j] -> A[i-1,j+1] } * Domain;
Write := [n] -> { S[i,j] -> A[i,j] } * Domain;
Sched := [n] -> { S[i,j] -> [i,j] };

# Operators
Prev := { [i,j]->[k,l]: i<k or (i=k and j<l) };
Preveq := { [i,j]->[k,l]: i<k or (i=k and j<=l) };
WriteBeforeTStep := (Prev^-1).(Sched^-1).Write;
ReadAfterTStep := Preveq.(Sched^-1).Read;

# Liveness and conflicts
Live := WriteBeforeTStep * ReadAfterTStep;
Conflict := (Live^-1).Live;
Delta := deltas Conflict;
```

☛

$$\begin{aligned}\mathrm{Delta}(n) = \quad & \{(1, i_1) \mid i_1 \leq 0,\ n \geq 3,\ i_1 \geq 1 - n\} \cup \\ & \{(0, i_1) \mid i_1 \geq 1 - n,\ n \geq 2,\ i_1 \leq -1 + n\} \cup \\ & \{(-1, i_1) \mid i_1 \geq 0,\ n \geq 3,\ i_1 \leq -1 + n\}\end{aligned}$$

Generalizations? Liveness sets not the right concept

Inner parallelism Almost the same.

Generalizations? Liveness sets not the right concept

Inner parallelism Almost the same.

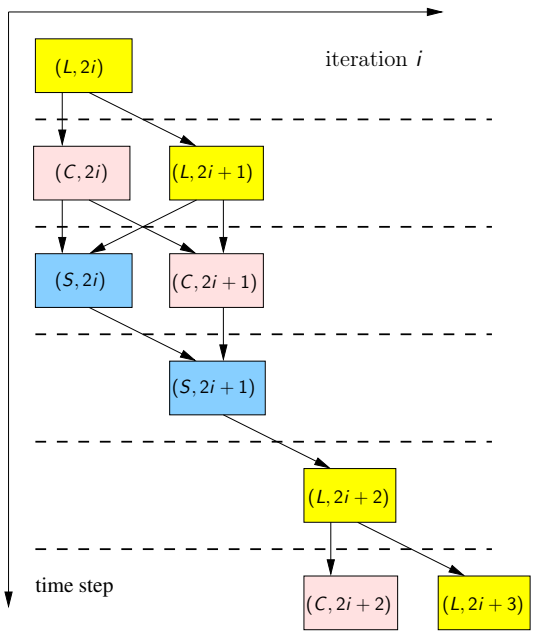
Seq/Par nested loops Can use a careful hierarchical approach.

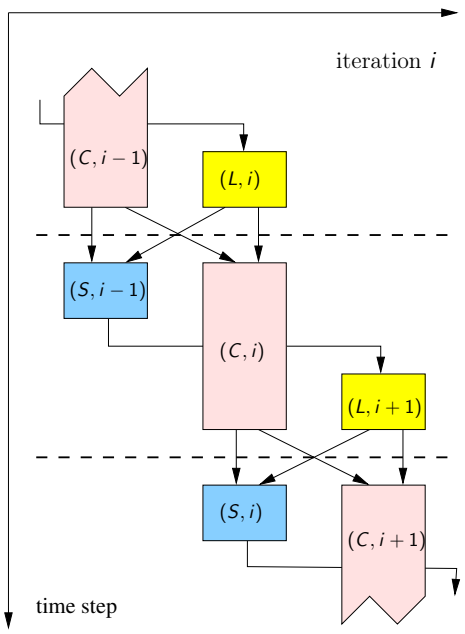
Generalizations? Liveness sets not the right concept

Inner parallelism Almost the same.

Seq/Par nested loops Can use a careful hierarchical approach.

Software pipelining Harder to get a concept of "time".





On the right, values computed in $S(i-1)$ and $L(i+1)$ both conflict with those in (C, i), but not with each other. Not a clique.

Reasoning at the level of traces

Define:

- $a \in t$ iff a is executed in a trace t;
- $a \prec_t b$ iff $a \in t$, $b \in t$ and a is executed before b in t;
- $S_\exists(a, b)$ iff there is a trace t such that $a \prec_t b$.
- $R_\forall(a, b) = \neg S_\exists(b, a)$ iff, for all traces t, $a, b \in t$ implies $a \prec_t b$.

Then, a and b conflict ($a \bowtie b$) if, for some trace t, $W_a \prec_t W_b \prec_t R_a$.

Reasoning at the level of traces

Define:

- $a \in t$ iff a is executed in a trace t;
- $a \prec_t b$ iff $a \in t$, $b \in t$ and a is executed before b in t;
- $S_\exists(a, b)$ iff there is a trace t such that $a \prec_t b$.
- $R_\forall(a, b) = \neg S_\exists(b, a)$ iff, for all traces t, $a, b \in t$ implies $a \prec_t b$.

Then, a and b conflict ($a \bowtie b$) if, for some trace t, $W_a \prec_t W_b \prec_t R_a$.

Conservative approximations for $a \bowtie b$:

- iff $S_\exists(W_a, R_a)$, $S_\exists(W_a, W_b)$, $S_\exists(W_b, R_a)$ iff $\neg R_\forall(R_a, W_a)$, $\neg R_\forall(W_b, W_a)$, $\neg R_\forall(R_a, W_b)$.
- with an under-approximation $\underline{R}_\forall \subseteq R_\forall$.

Reasoning at the level of traces

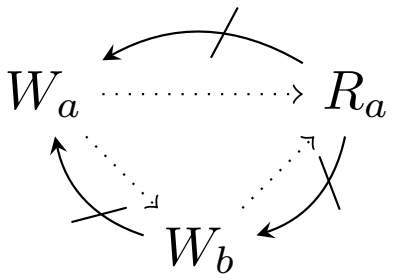
Define:

- $a \in t$ iff a is executed in a trace t;
- $a \prec_t b$ iff $a \in t$, $b \in t$ and a is executed before b in t;
- $S_\exists(a, b)$ iff there is a trace t such that $a \prec_t b$.
- $R_\forall(a, b) = \neg S_\exists(b, a)$ iff, for all traces t, $a, b \in t$ implies $a \prec_t b$.

Then, a and b conflict ($a \bowtie b$) if, for some trace t, $W_a \prec_t W_b \prec_t R_a$.

Conservative approximations for $a \bowtie b$:

- iff $S_\exists(W_a, R_a)$, $S_\exists(W_a, W_b)$, $S_\exists(W_b, R_a)$ iff $\neg R_\forall(R_a, W_a)$, $\neg R_\forall(W_b, W_a)$, $\neg R_\forall(R_a, W_b)$.
- with an under-approximation $\underline{R}_\forall \subseteq R_\forall$.



When $\underline{R}_\forall$ is a partial order $\preceq$, $a \bowtie b$ iff $R_a \not\preceq W_a$, $W_b \not\preceq W_a$, $R_a \not\preceq W_b$.

☛ Covers sequential code, OpenMP-like loop parallelism, OpenMP-4.0 task parallelism, X10, OpenStream, even some form of if conditions, etc.

Partial orders, user-defined data races, comparability graphs

Mapping: if allocation respects $\bowtie$, it is valid for any execution expressed by the parallel specification ☛ form of schedule-independent mapping.

Partial orders, user-defined data races, comparability graphs

Mapping: if allocation respects $\bowtie$, it is valid for any execution expressed by the parallel specification ☛ form of schedule-independent mapping.

Partial order: quite general, but cannot take critical sections into account. Theory can handle if conditions, but not a partial order anymore.

Partial orders, user-defined data races, comparability graphs

Mapping: if allocation respects $\bowtie$, it is valid for any execution expressed by the parallel specification ☛ form of schedule-independent mapping.

Partial order: quite general, but cannot take critical sections into account. Theory can handle if conditions, but not a partial order anymore.

Interference graph: if no dead code, no undefined read, but possibly races, it is the complement of a comparability graph.

Optimality: size = max clique, polynomially computable (Dilworth) if graph is given in extension (unlike polyhedral optimization). Note: different than finding the minimum size for any execution (NP-complete).

Partial orders, user-defined data races, comparability graphs

Mapping: if allocation respects $\bowtie$, it is valid for any execution expressed by the parallel specification ☛ form of schedule-independent mapping.

Partial order: quite general, but cannot take critical sections into account. Theory can handle if conditions, but not a partial order anymore.

Interference graph: if no dead code, no undefined read, but possibly races, it is the complement of a comparability graph.

Optimality: size = max clique, polynomially computable (Dilworth) if graph is given in extension (unlike polyhedral optimization). Note: different than finding the minimum size for any execution (NP-complete).

Source-to-source transformation: contraction can be expressed in the same specification model, without constraining parallelism further.

Still many open questions on how to exploit these properties further.

Outline

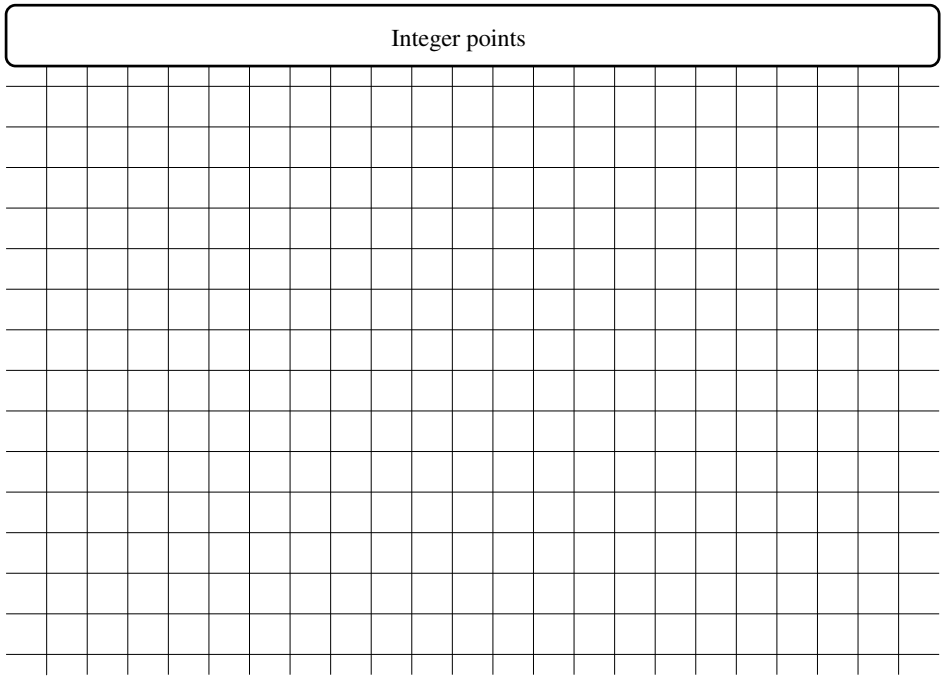
1. Polyhedral representation examples
   - Compilation for GPU, with shared-memory optimization
   - Tiling with automatic double-buffering, transfers and buffer sizes
2. Exploring different forms of parallelism
   - Analysis of a X10 subset
   - Analysis of an OpenStream subset
3. Liveness analysis
   - Chordal and interval graphs for SSA and SSI
   - Comparability graphs for partial orders
4. Lattice-based memory allocation
   - Polyhedral conflicts
   - Conflicts as union of polyhedra

Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



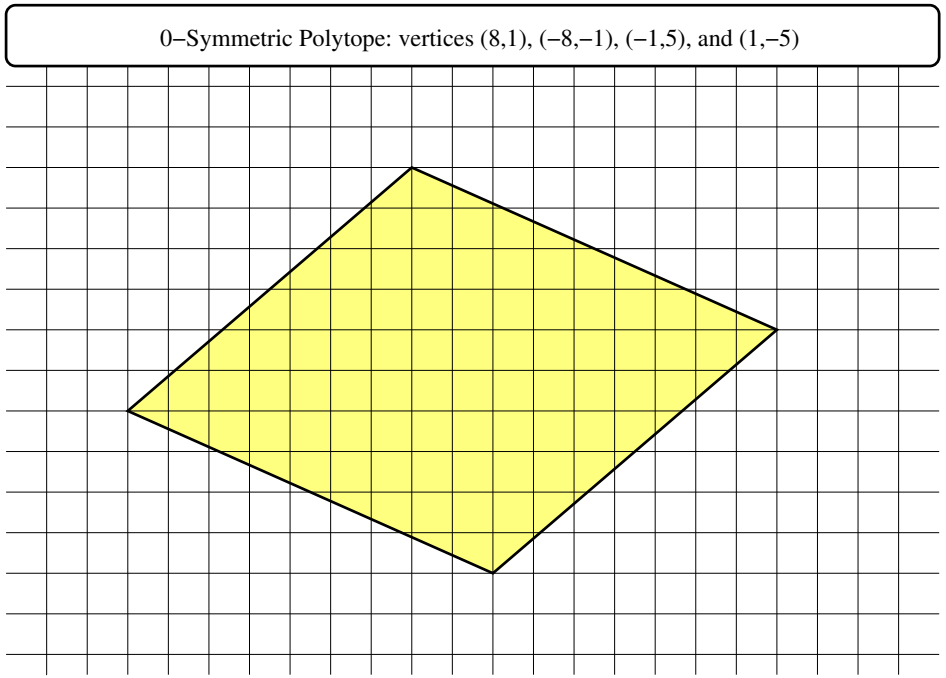
- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



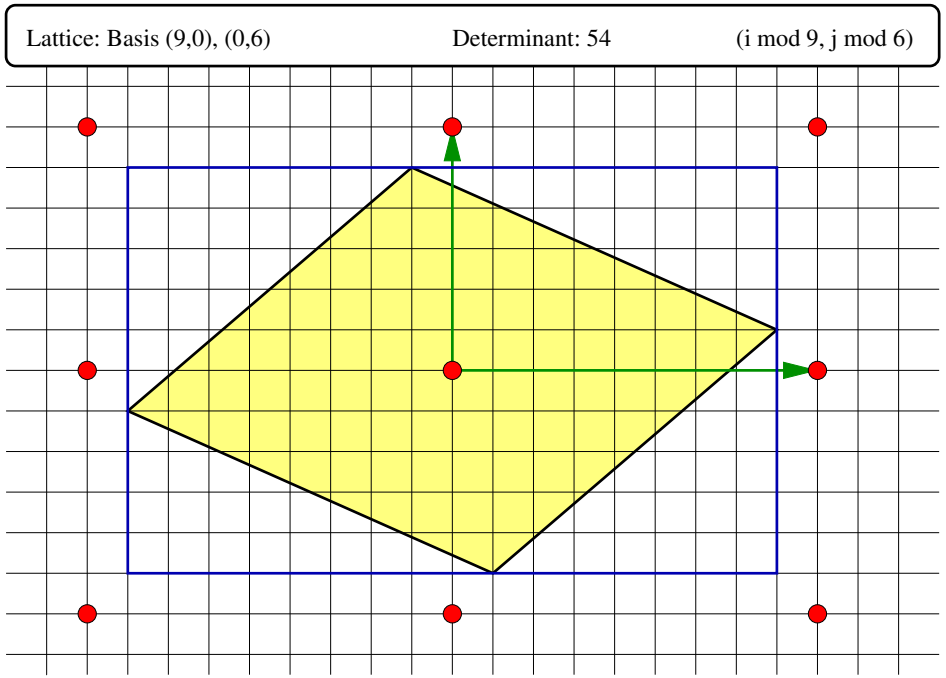
- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



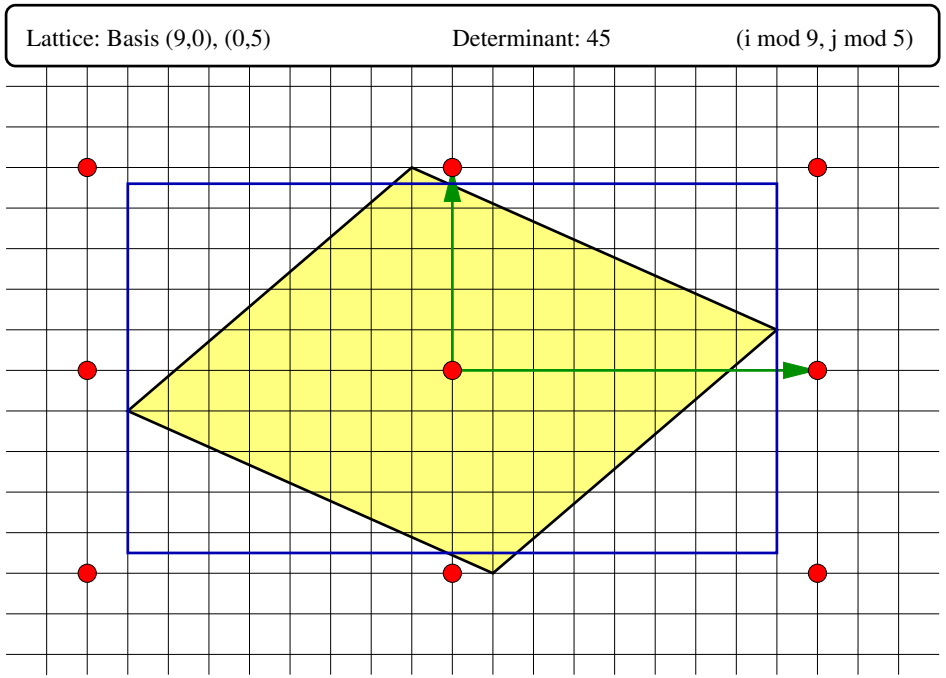
- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



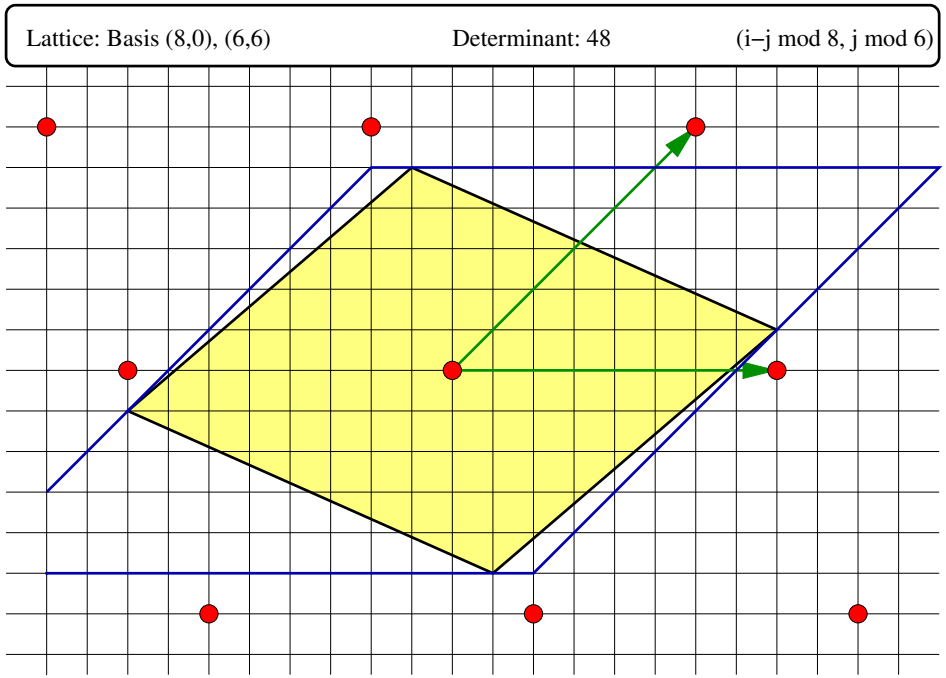
- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



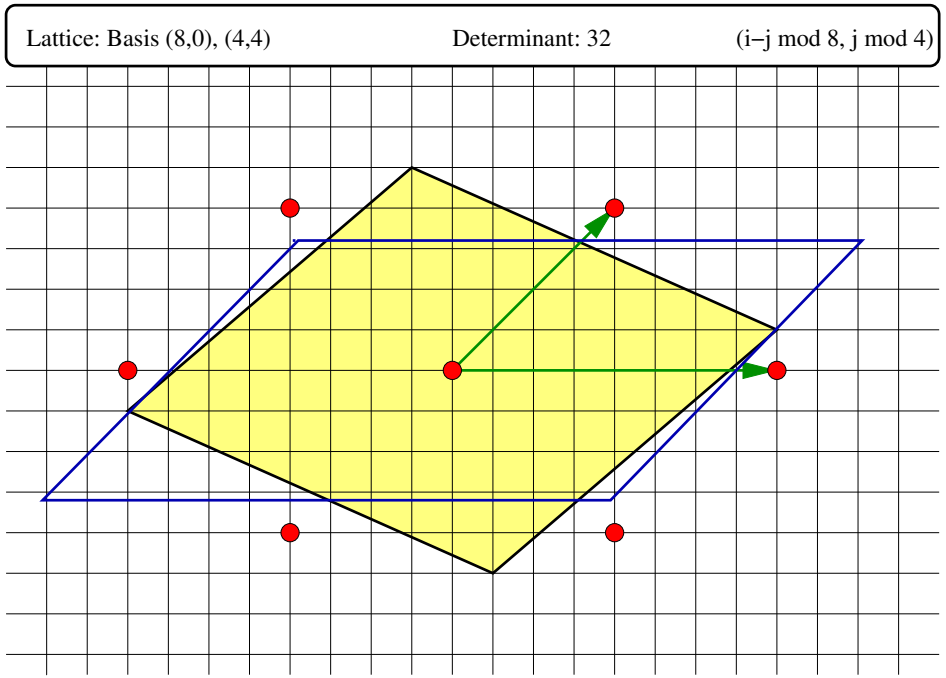
- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



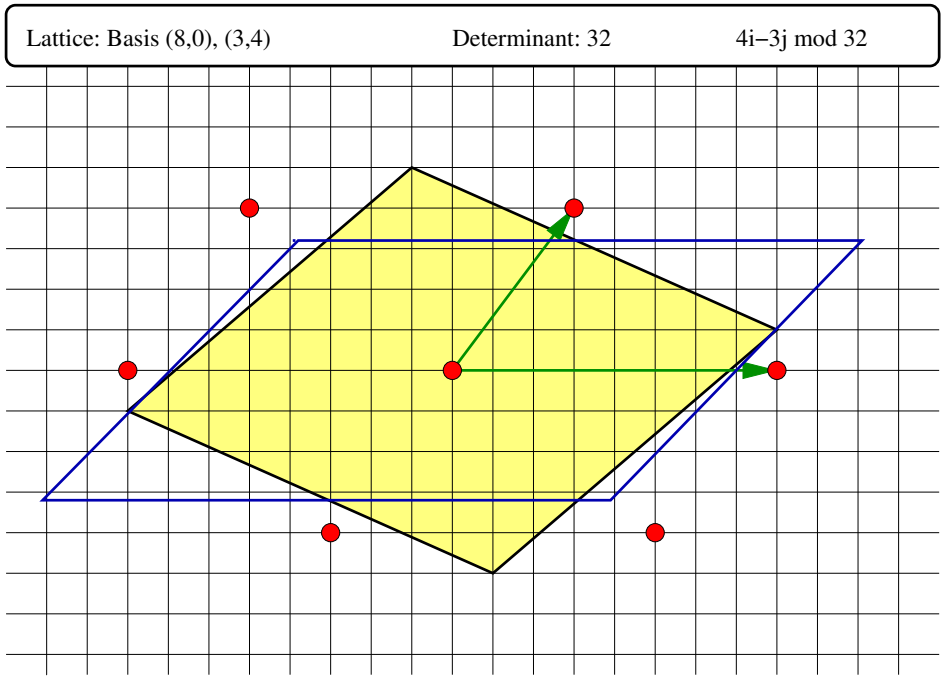
- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



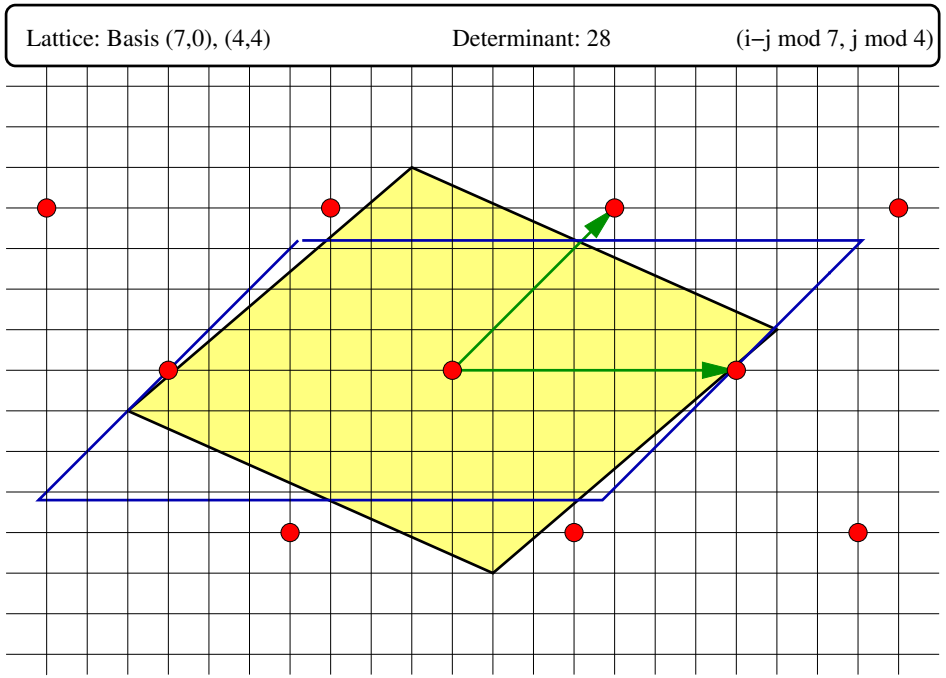
- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



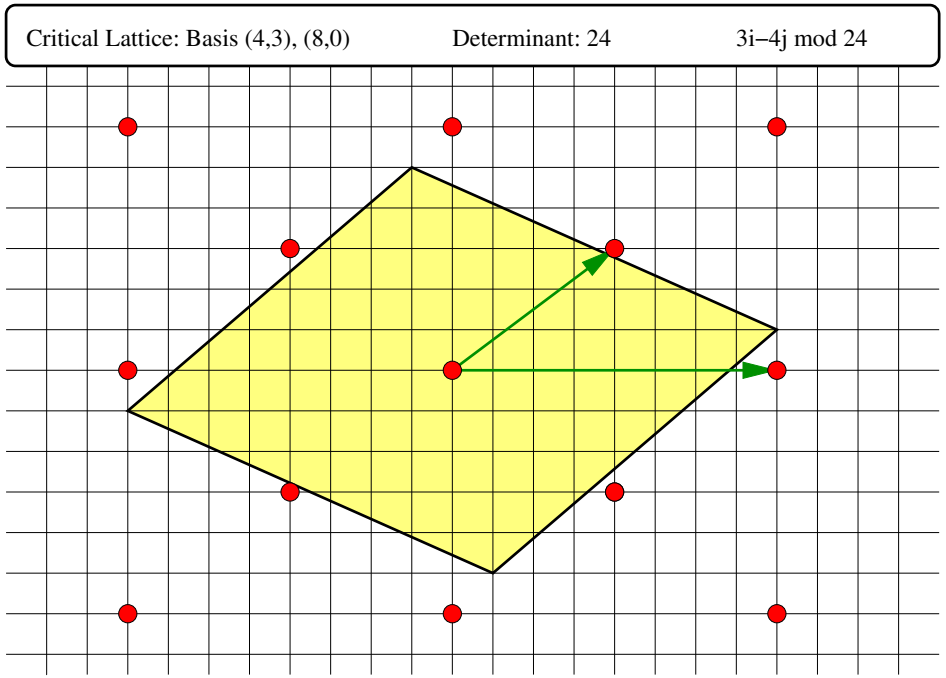
- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

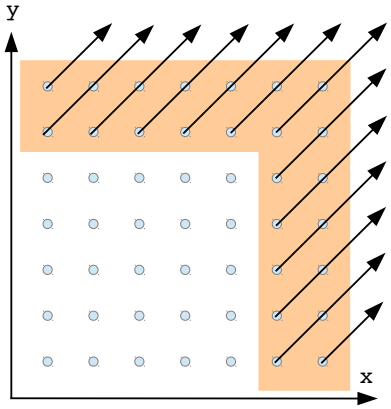
Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

Dealing with union of polyhedra: new theory

Live-out set of a tiled code:

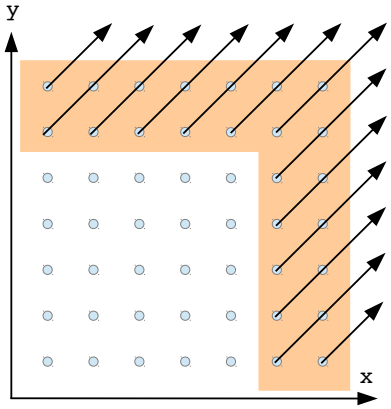


- Successive modulo:
  $(x, y) \mapsto (x \bmod N, y \bmod N)$.
- Skewed mapping:
  $(x, y) \mapsto (x - y \bmod (2N - 1), y \bmod 2)$.

☛ How to find the second one?

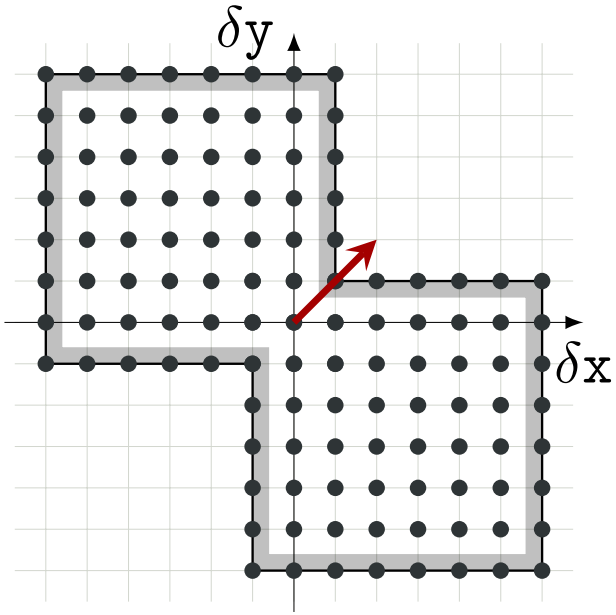
Dealing with union of polyhedra: new theory

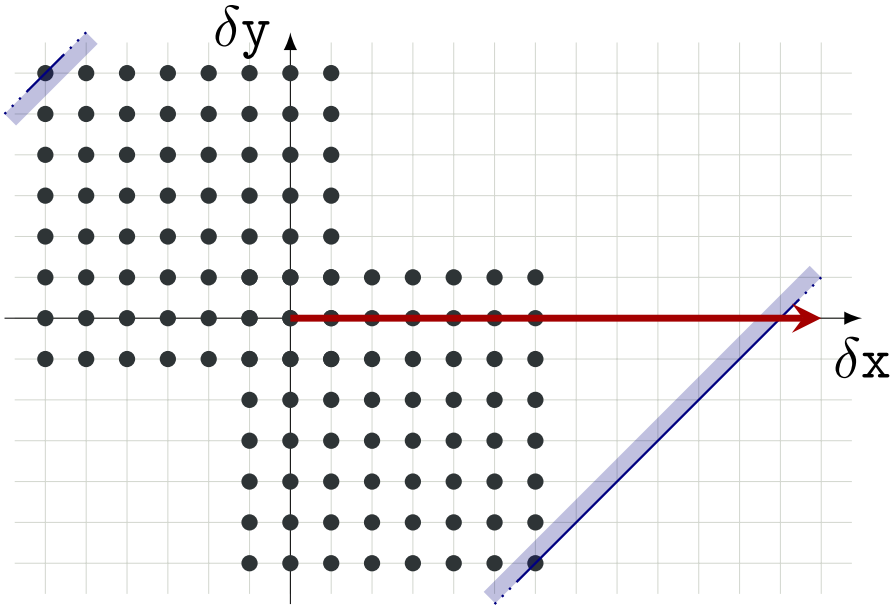
Live-out set of a tiled code:



- Successive modulo:
  $(x, y) \mapsto (x \bmod N, y \bmod N)$.
- Skewed mapping:
  $(x, y) \mapsto (x - y \bmod (2N-1), y \bmod 2)$.

☛ How to find the second one?





Conclusion: many pieces of the puzzle get together

New generalizations and links with previous approaches.

- Liveness analysis for parallel specifications.
- Interference graph structure analysis and exploitation.
- Lattice-based memory allocation extensions.

☛ Towards a better understanding of parallel languages: semantics, static analysis, and links with the runtime.

Still some problems or applications to explore, possibly with MC2 (graph structures), Avalon (OpenMP 4.0/StarPU dependent tasks), Roma (memory optimization), Aric (lattice theory).