(Automatic) Parallelization

Marc Moreno Maza

Ontario Research Center for Computer Algebra
Departments of Computer Science and Mathematics
University of Western Ontario, Canada

CS4402 - CS9635, February 28, 2024

Outline

Outline

First examples

- Can the loop on the right be run in parallel?

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

First examples

- Can the loop on the right be run in parallel?
- That is, can we replace `for` with `cilk_for`?

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

First examples

- Can the loop on the right be run in parallel?
- That is, can we replace `for` with `cilk_for`?
- No, since one cannot guarantee the executuion order of iterations when run concurrently.

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i] + b[i - 1];
}
```

First examples

- Can the loop on the right be run in parallel?
- That is, can we replace `for` with `cilk_for`?
- No, since one cannot guarantee the executuion order of iterations when run concurrently.

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

- What needs to be true for a loop to be parallelizable?

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i] + b[i - 1];
}
```

First examples

- Can the loop on the right be run in parallel?
- That is, can we replace `for` with `cilk_for`?
- No, since one cannot guarantee the executuion order of iterations when run concurrently.

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

- What needs to be true for a loop to be parallelizable?
- Iterations cannot interfere with each other

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i] + b[i - 1];
}
```

Detailed answer

- A *flow dependence* occurs when one iteration writes a location that a later iteration reads.

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

i = 1	i = 2	i = 3	i = 4	i = 5
W(a[1])	W(a[2])	W(a[3])	W(a[4])	W(a[5])
R(b[1])	R(b[2])	R(b[3])	R(b[4])	R(b[5])
W(c[1])	W(c[2])	W(c[3])	W(c[4])	W(c[5])
R(a[0])	R(a[1])	R(a[2])	R(a[3])	R(a[4])

Detailed answer

- A *flow dependence* occurs when one iteration writes a location that a later iteration reads.
- In the first example of the previous slide, there is a flow dependence from the first statement at iteration `i` to the second statement at iteration `i+1`.

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

i = 1	i = 2	i = 3	i = 4	i = 5
W(a[1])	W(a[2])	W(a[3])	W(a[4])	W(a[5])
R(b[1])	R(b[2])	R(b[3])	R(b[4])	R(b[5])
W(c[1])	W(c[2])	W(c[3])	W(c[4])	W(c[5])
R(a[0])	R(a[1])	R(a[2])	R(a[3])	R(a[4])

Other kinds of dependence

- Anti dependence: when an iteration reads a location that a later iteration writes

```
for (i = 1; i < N; i++) {
  a[i - 1] = b[i];
  c[i] = a[i];
}
```

Other kinds of dependence

- Anti dependence: when an iteration reads a location that a later iteration writes
- why considering that dependence?

```
for (i = 1; i < N; i++) {
  a[i - 1] = b[i];
  c[i] = a[i];
}
```

Other kinds of dependence

- Anti dependence: when an iteration reads a location that a later iteration writes
- why considering that dependence?
- Same reason as for flow dependence.

```
for (i = 1; i < N; i++) {
  a[i - 1] = b[i];
  c[i] = a[i];
}
```

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  a[i + 1] = c[i];
}
```

Other kinds of dependence

- Anti dependence: when an iteration reads a location that a later iteration writes
- why considering that dependence?
- Same reason as for flow dependence.

```
for (i = 1; i < N; i++) {
  a[i - 1] = b[i];
  c[i] = a[i];
}
```

- Output dependence: when an iteration writes a location that a later iteration writes

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  a[i + 1] = c[i];
}
```

Other kinds of dependence

- Anti dependence: when an iteration reads a location that a later iteration writes
- why considering that dependence?
- Same reason as for flow dependence.

```
for (i = 1; i < N; i++) {
  a[i - 1] = b[i];
  c[i] = a[i];
}
```

- Output dependence: when an iteration writes a location that a later iteration writes
- Same motivation as for flow dependence.

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  a[i + 1] = c[i];
}
```

Outline

Data dependence analysis

Data dependence

- A data dependency happens when, in a program, (i) a statement S_2 accesses a memory location that is accessed by a preceding statement S_1, (ii) one of these two accesses is a WRITE, and (iii) there is an execution path from S_1 to S_2.

Data dependence analysis

Data dependence

- A data dependency happens when, in a program, (i) a statement S_2 accesses a memory location that is accessed by a preceding statement S_1, (ii) one of these two accesses is a WRITE, and (iii) there is an execution path from S_1 to S_2.
- We denote by $I(S_i)$ (resp. $O(S_i)$) the set of the memory locations read (resp. written) by the statement S_i, for $1 \leq i \leq 2$.

Data dependence analysis

Data dependence

- A data dependency happens when, in a program, (i) a statement S_2 accesses a memory location that is accessed by a preceding statement S_1, (ii) one of these two accesses is a WRITE, and (iii) there is an execution path from S_1 to S_2.
- We denote by $I(S_i)$ (resp. $O(S_i)$) the set of the memory locations read (resp. written) by the statement S_i, for $1 \leq i \leq 2$.

There are three types of dependence:

Data dependence analysis

Data dependence

- A data dependency happens when, in a program, (i) a statement S_2 accesses a memory location that is accessed by a preceding statement S_1, (ii) one of these two accesses is a WRITE, and (iii) there is an execution path from S_1 to S_2.
- We denote by $I(S_i)$ (resp. $O(S_i)$) the set of the memory locations read (resp. written) by the statement S_i, for $1 \leq i \leq 2$.

There are three types of dependence:

1. Anti-dependence: $I(S_1) \cap O(S_2) \neq \emptyset$, that is, S_1 reads some memory location before S_2 overwrites it

Data dependence analysis

Data dependence

- A data dependency happens when, in a program, (i) a statement S_2 accesses a memory location that is accessed by a preceding statement S_1, (ii) one of these two accesses is a WRITE, and (iii) there is an execution path from S_1 to S_2.
- We denote by $I(S_i)$ (resp. $O(S_i)$) the set of the memory locations read (resp. written) by the statement S_i, for $1 \leq i \leq 2$.

There are three types of dependence:

1. Anti-dependence: $I(S_1) \cap O(S_2) \neq \emptyset$, that is, S_1 reads some memory location before S_2 overwrites it
2. True dependence (or Flow dependence): $O(S_1) \cap I(S_2) \neq \emptyset$, that is, S_1 writes to some memory location before S_2 reads from it

Data dependence analysis

Data dependence

- A data dependency happens when, in a program, (i) a statement S_2 accesses a memory location that is accessed by a preceding statement S_1, (ii) one of these two accesses is a WRITE, and (iii) there is an execution path from S_1 to S_2.
- We denote by $I(S_i)$ (resp. $O(S_i)$) the set of the memory locations read (resp. written) by the statement S_i, for $1 \leq i \leq 2$.

There are three types of dependence:

1. Anti-dependence: $I(S_1) \cap O(S_2) \neq \emptyset$, that is, S_1 reads some memory location before S_2 overwrites it
2. True dependence (or Flow dependence): $O(S_1) \cap I(S_2) \neq \emptyset$, that is, S_1 writes to some memory location before S_2 reads from it
3. Output dependence: $O(S_1) \cap O(S_2) \neq \emptyset$, that is, S_1 and S_2 write to the same memory location.

Data dependence analysis

- The *dependence source* is the earlier statement (the statement at the tail of the dependence arrow)

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

i = 1	i = 2	i = 3	i = 4	i = 5
W(a[1])	W(a[2])	W(a[3])	W(a[4])	W(a[5])
R(b[1])	R(b[2])	R(b[3])	R(b[4])	R(b[5])
W(c[1])	W(c[2])	W(c[3])	W(c[4])	W(c[5])
R(a[0])	R(a[1])	R(a[2])	R(a[3])	R(a[4])

Note that dependences can only go forward in time: always from an earlier iteration to a later iteration.

Data dependence analysis

- The *dependence source* is the earlier statement (the statement at the tail of the dependence arrow)
- The *dependence sink* is the later statement (the statement at the head of the dependence arrow)

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

i = 1	i = 2	i = 3	i = 4	i = 5
W(a[1])	W(a[2])	W(a[3])	W(a[4])	W(a[5])
R(b[1])	R(b[2])	R(b[3])	R(b[4])	R(b[5])
W(c[1])	W(c[2])	W(c[3])	W(c[4])	W(c[5])
R(a[0])	R(a[1])	R(a[2])	R(a[3])	R(a[4])

Note that dependences can only go forward in time: always from an earlier iteration to a later iteration.

Data dependence analysis

Loop dependence theorem

There exists a dependence from statement S_1 to statement S_2 in a common nest of loops if and only if there exist two index vectors $\mathbf{i}$ and $\mathbf{j}$ for the nest, such that

(1) $\mathbf{i} < \mathbf{j}$ or $\mathbf{i} = \mathbf{j}$ and there is a path from S_1 to S_2 in the body of the loop,
(2) statement S_1 accesses memory location M on index $\mathbf{i}$ and statement S_2 accesses location M on index $\mathbf{j}$, and
(3) one of these accesses is a WRITE.

When no such dependence exists within that loop nest, then all iterations can be executed in parallel.

Data dependence analysis

Loop dependence theorem

There exists a dependence from statement S_1 to statement S_2 in a common nest of loops if and only if there exist two index vectors $\mathbf{i}$ and $\mathbf{j}$ for the nest, such that

(1) $\mathbf{i} < \mathbf{j}$ or $\mathbf{i} = \mathbf{j}$ and there is a path from S_1 to S_2 in the body of the loop,

(2) statement S_1 accesses memory location M on index $\mathbf{i}$ and statement S_2 accesses location M on index $\mathbf{j}$, and

(3) one of these accesses is a WRITE.

When no such dependence exists within that loop nest, then all iterations can be executed in parallel.

Problems

- How do we represent dependences in loops?

Data dependence analysis

Loop dependence theorem

There exists a dependence from statement S_1 to statement S_2 in a common nest of loops if and only if there exist two index vectors $\mathbf{i}$ and $\mathbf{j}$ for the nest, such that

(1) $\mathbf{i} < \mathbf{j}$ or $\mathbf{i} = \mathbf{j}$ and there is a path from S_1 to S_2 in the body of the loop,
(2) statement S_1 accesses memory location M on index $\mathbf{i}$ and statement S_2 accesses location M on index $\mathbf{j}$, and
(3) one of these accesses is a WRITE.

When no such dependence exists within that loop nest, then all iterations can be executed in parallel.

Problems

- How do we represent dependences in loops?
- How do we determine if there are dependences?

Outline

Iteration space graphs

- Loop dependence analysis focuses on finding the dependencies within iterations of a loop or a loop-nest.

Iteration space graphs

- Loop dependence analysis focuses on finding the dependencies within iterations of a loop or a loop-nest.
- One way to show the dependence between the same statement in different iterations of the loop is to use the iteration space graphs

Iteration space graphs

- Loop dependence analysis focuses on finding the dependencies within iterations of a loop or a loop-nest.
- One way to show the dependence between the same statement in different iterations of the loop is to use the iteration space graphs

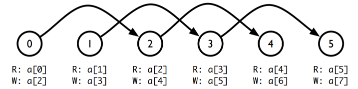
Iteration space graph

The iteration space graphs represent each execution point (a particular statement at a particular iteration) of a loop as a node in a graph. Then, one draws an arrow from one point P to a point Q whenever there is a flow dependence from P to Q.

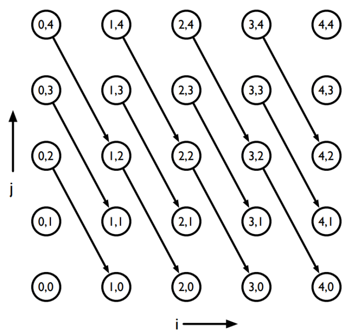
Iteration space graphs

Iteration space graphs

```
for (int i ...)
    a[i + 2] = a[i];
```




```
for (int i ...)
    for (int j ...)
        a[i + 1][j - 2] = a[i][j]
```



Iteration space graphs

Remarks

- Iteration space graphs can also be used to represent output and anti dependences

Iteration space graphs

Remarks

- Iteration space graphs can also be used to represent output and anti dependences
- To this end, one can use different kinds of arrows for clarity.

Iteration space graphs

Remarks

- Iteration space graphs can also be used to represent output and anti dependences
- To this end, one can use different kinds of arrows for clarity.

Limitations

- The number of arrows in an Iteration space graph can grow exponentially with the number of iterations.

Iteration space graphs

Remarks

- Iteration space graphs can also be used to represent output and anti dependences
- To this end, one can use different kinds of arrows for clarity.

Limitations

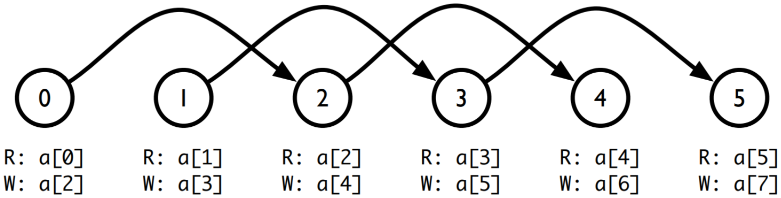
- The number of arrows in an Iteration space graph can grow exponentially with the number of iterations.
- Can we represent dependences in a more compact way?

Outline

Distance and direction vectors

Distance vector: informal definition

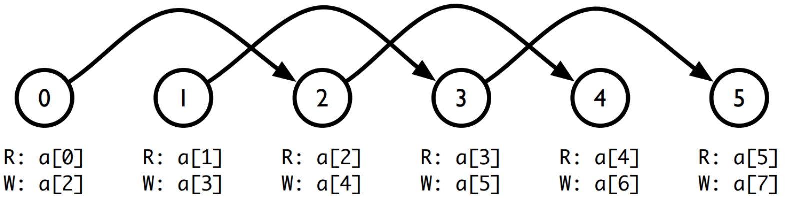
- The distance vector represents each dependence arrow in an iteration space graph as a vector



Distance and direction vectors

Distance vector: informal definition

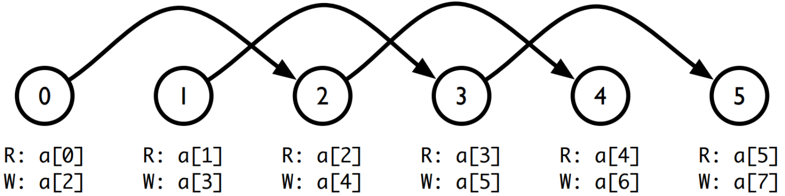
- The distance vector represents each dependence arrow in an iteration space graph as a vector
- That is, it captures the "shape" of the dependence, but loses where the dependence originates



Distance and direction vectors

Distance vector: informal definition

- The distance vector represents each dependence arrow in an iteration space graph as a vector
- That is, it captures the “shape” of the dependence, but loses where the dependence originates

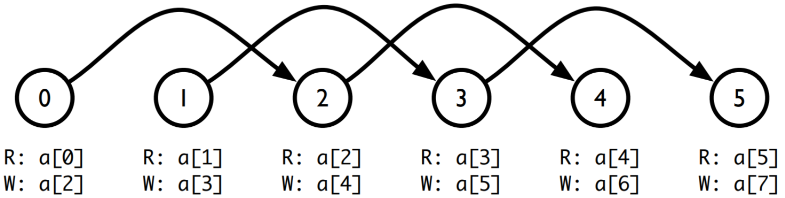


- For the above 1D iteration space graph, the distance vector is (2)

Distance and direction vectors

Distance vector: informal definition

- The distance vector represents each dependence arrow in an iteration space graph as a vector
- That is, it captures the "shape" of the dependence, but loses where the dependence originates



- For the above 1D iteration space graph, the distance vector is (2)
- Indeed, each dependence is 2 iterations forward

Distance and direction vectors

Distance vector

For an n-loop nest, let $\vec{I} = (i_1, \ldots, i_n)$ and $\vec{I'} = (i'_1, \ldots, i'_n)$ be two iterations. The distance vector $d(\vec{I}, \vec{I'})$ from $\vec{I}$ to $\vec{I'}$ is a vector of length n and with $i'_k - i_k$ as k-th coordinate.

Distance and direction vectors

Distance vector

For an n-loop nest, let $\vec{I} = (i_1, \ldots, i_n)$ and $\vec{I'} = (i'_1, \ldots, i'_n)$ be two iterations. The distance vector $d(\vec{I}, \vec{I'})$ from $\vec{I}$ to $\vec{I'}$ is a vector of length n and with $i'_k - i_k$ as k-th coordinate.

Direction vector

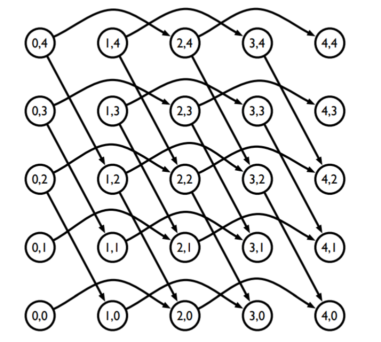
With the same notations as above, the direction vector $D(\vec{I}, \vec{I'})$ is defined as a vector of length n such that

$$D(\vec{I}, \vec{I'})_k = \begin{cases} \text{“} < \text{”} \text{ if } d(\vec{I}, \vec{I'})_k > 0 \\ \text{“} = \text{”} \text{ if } d(\vec{I}, \vec{I'})_k = 0 \\ \text{“} > \text{”} \text{ if } d(\vec{I}, \vec{I'})_k < 0 \end{cases}$$

Distance and direction vectors

Iteration space graphs

```
for (int i ...)
   for (int j ...)
      a[i + 1][j - 2] = a[i][j] +
        a[i - 1][j - 2];
```

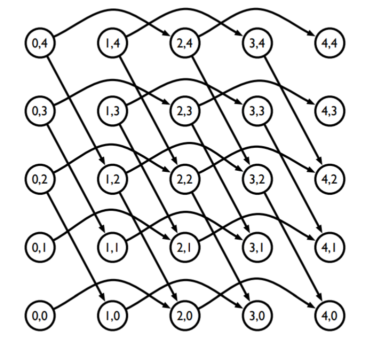


Distance and direction vectors

Distance and direction vectors

Iteration space graphs

```
for (int i ...)
   for (int j ...)
      a[i + 1][j - 2] = a[i][j] +
        a[i - 1][j - 2];
```



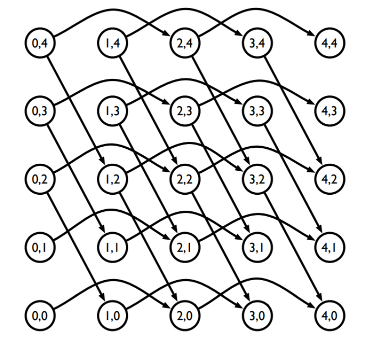
Distance and direction vectors

The distance vectors are: $(1, -2), (2, 0)$

Distance and direction vectors

Iteration space graphs

```
for (int i ...)
   for (int j ...)
      a[i + 1][j - 2] = a[i][j] +
        a[i - 1][j - 2];
```



Distance and direction vectors

The distance vectors are: $(1, -2), (2, 0)$

The direction vectors are: $(<, >), (<, =)$

Distance and direction vectors

Problems with distance vectors

Cannot always summarize as easily:

```
for (i = 0; i < N; i++)
a[2*i] = a[i];
```

Nevertheless:

- Direction vectors lose a lot of information, but do capture some useful information: Which dimension and direction the dependence is in.

Distance and direction vectors

Problems with distance vectors

Cannot always summarize as easily:

```
for (i = 0; i < N; i++)
a[2*i] = a[i];
```

Nevertheless:

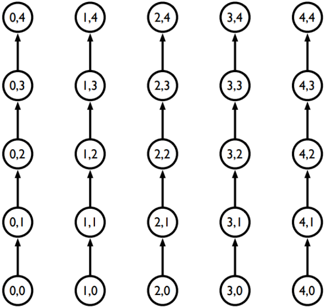
- Direction vectors lose a lot of information, but do capture some useful information: Which dimension and direction the dependence is in.
- Often, the only information we need to determine (in order to decide whether an optimization is legal) is captured by direction vectors.

More examples (1/3)

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

More examples (1/3)

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



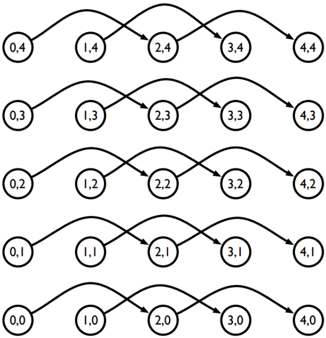
Can parallelize i loop but not j loop

More examples (2/3)

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

More examples (2/3)

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



Can parallelize j loop but not i loop

More examples (3/3)

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
       for (int k = 0; j < n; j++)
           a[i+1][j][k] = a[i][j][k+1] + B
```

More examples (3/3)

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
       for (int k = 0; j < n; j++)
           a[i+1][j][k] = a[i][j][k+1] + B
```

- Two iterations (i_0, j_0, k_0) and $(i_0 + \Delta i, j_0 + \Delta j, k_0 + \Delta k)$ access the same memory location, whenever we have:

$$i_0 + 1 = i_0 + \Delta i;\ j_0 = j_0 + \Delta j;\ k_0 = k_0 + \Delta k + 1.$$

More examples (3/3)

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
       for (int k = 0; j < n; j++)
           a[i+1][j][k] = a[i][j][k+1] + B
```

- Two iterations (i_0, j_0, k_0) and $(i_0 + \Delta i, j_0 + \Delta j, k_0 + \Delta k)$ access the same memory location, whenever we have:

$$i_0 + 1 = i_0 + \Delta i;\ j_0 = j_0 + \Delta j;\ k_0 = k_0 + \Delta k + 1.$$

- That is:

$$\Delta i = 1;\ \Delta j = 0;\ \Delta k = -1.$$

More examples (3/3)

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
       for (int k = 0; j < n; j++)
           a[i+1][j][k] = a[i][j][k+1] + B
```

- Two iterations (i_0, j_0, k_0) and $(i_0 + \Delta i, j_0 + \Delta j, k_0 + \Delta k)$ access the same memory location, whenever we have:

$$i_0 + 1 = i_0 + \Delta i;\ j_0 = j_0 + \Delta j;\ k_0 = k_0 + \Delta k + 1.$$

- That is:

$$\Delta i = 1;\ \Delta j = 0;\ \Delta k = -1.$$

- The corresponding direction vector is: $(<, =, >)$.
- The j-loop can be vectorized.

Outline

Outline

Delinearization

Linearized multi-dimensional array

```
for (int i = 0; i < n; i ++)
    for (int j = i + 1; j < n; j ++)
        A[i * n + j] =
                A[n * n - n + j - 1];
```

$$\begin{cases} 0 \le i_1 < n \\ i_1 + 1 \le j_1 < n \\ 0 \le i_2 < n \\ i_2 + 1 \le j_2 < n \\ i_1 * n + j_1 = n^2 - n + j_2 - 1 \end{cases} \tag{1}$$

Delinearized multi-dimensional array

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

$$\begin{cases} 0 \le i_1 < n \\ i_1 + 1 \le j_1 < n \\ 0 \le i_2 < n \\ i_2 + 1 \le j_2 < n \\ i_1 = n - 1 \\ j_1 = j_2 - 1 \end{cases} \tag{2}$$

Type of array references

- We assume that arrays are multi-dimensional, thus delinearization has been applied.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
       for (int k = 0; j < n; j++)
            a [5][i+1][j] = a[N][i][k] + B
```

In the above loop body, the pair `[5]`, `[N]` is ZIV, the pair `[i+1]`,`[i]` is SIV and the pair `[j]`,`[k]` is MIV.

Type of array references

- We assume that arrays are multi-dimensional, thus delinearization has been applied.
- A pair of array references (or subscripts) in the same dimension is

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
       for (int k = 0; j < n; j++)
            a [5][i+1][j] = a[N][i][k] + B
```

In the above loop body, the pair `[5]`, `[N]` is ZIV, the pair `[i+1]`,`[i]` is SIV and the pair `[j]`,`[k]` is MIV.

Type of array references

- We assume that arrays are multi-dimensional, thus delinearization has been applied.
- A pair of array references (or subscripts) in the same dimension is
  - ↳ zero index variable (ZIV) if it does not involve any loop counter,

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; j < n; j++)
            a [5][i+1][j] = a[N][i][k] + B
```

In the above loop body, the pair `[5]`, `[N]` is ZIV, the pair `[i+1]`,`[i]` is SIV and the pair `[j]`,`[k]` is MIV.

Type of array references

- We assume that arrays are multi-dimensional, thus delinearization has been applied.
- A pair of array references (or subscripts) in the same dimension is
  - ↳ zero index variable (ZIV) if it does not involve any loop counter,
  - ↳ single index variable (SIV) if it involves only one loop counter

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
       for (int k = 0; j < n; j++)
            a [5][i+1][j] = a[N][i][k] + B
```

In the above loop body, the pair `[5]`, `[N]` is ZIV, the pair `[i+1]`,`[i]` is SIV and the pair `[j]`,`[k]` is MIV.

Type of array references

- We assume that arrays are multi-dimensional, thus delinearization has been applied.
- A pair of array references (or subscripts) in the same dimension is
  - zero index variable (ZIV) if it does not involve any loop counter,
  - single index variable (SIV) if it involves only one loop counter
  - multiple index variable (MIV) if it involves more than one one loop counter.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
       for (int k = 0; j < n; j++)
            a [5][i+1][j] = a[N][i][k] + B
```

In the above loop body, the pair `[5]`, `[N]` is ZIV, the pair `[i+1]`,`[i]` is SIV and the pair `[j]`,`[k]` is MIV.

ZIV test

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.

ZIV test

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for that pair, we have a ZIV

ZIV test

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for that pair, we have a ZIV
- If the two index expressions are different, then the corresponding array references are independent.

ZIV test

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for that pair, we have a ZIV
- If the two index expressions are different, then the corresponding array references are independent.
- If the two index expressions are the same, further dependence analysis is needed.

Strong SIV

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.

Strong SIV Test

Strong SIV

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for both, a subscript is SIV and involves the index i.

Strong SIV Test

Strong SIV

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for both, a subscript is SIV and involves the index i.
- We say that this SIV subscript is **strong** if the two index expressions are of the form $a\,i + c_1$, $a\,i' + c_2$ respectively, where a, c_1, c_2 are integers with $a \neq 0$.

Strong SIV Test

Strong SIV

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for both, a subscript is SIV and involves the index i.
- We say that this SIV subscript is **strong** if the two index expressions are of the form $a\,i + c_1$, $a\,i' + c_2$ respectively, where a, c_1, c_2 are integers with $a \neq 0$.
- The corresponding dependence distance is calculated by

$$d = i' - i = \frac{c_1 - c_2}{a}.$$

Strong SIV Test

Strong SIV

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for both, a subscript is SIV and involves the index i.
- We say that this SIV subscript is **strong** if the two index expressions are of the form $a\,i + c_1$, $a\,i' + c_2$ respectively, where a, c_1, c_2 are integers with $a \neq 0$.
- The corresponding dependence distance is calculated by

$$d = i' - i = \frac{c_1 - c_2}{a}.$$

Strong SIV Test

- A dependence exists between two references only if $|d| \leq U - L$, where U and L are the loop upper and lower bounds for the index i.

Strong SIV

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for both, a subscript is SIV and involves the index i.
- We say that this SIV subscript is **strong** if the two index expressions are of the form $a\,i + c_1$, $a\,i' + c_2$ respectively, where a, c_1, c_2 are integers with $a \neq 0$.
- The corresponding dependence distance is calculated by

$$d = i' - i = \frac{c_1 - c_2}{a}.$$

Strong SIV Test

- A dependence exists between two references only if $|d| \leq U - L$, where U and L are the loop upper and lower bounds for the index i.
- Otherwise, no dependence exists.

Strong SIV test example

do $I = 1, N$

$S_1 :\quad A(I + 2 * N) = A(I + N) + B$

enddo

Strong SIV test example

do $I = 1, N$
$S_1 : \quad A(I + 2 * N) = A(I + N) + B$
enddo

- The dependence distance: $d = 2 * N - N = N$;

Strong SIV test example

do $I = 1, N$

$S_1 : \quad A(I + 2 * N) = A(I + N) + B$

enddo

- The dependence distance: $d = 2 * N - N = N$;
- $U - L = N - 1$;

Strong SIV test example

do $I = 1, N$
$S_1 : \quad A(I + 2 * N) = A(I + N) + B$
enddo

- The dependence distance: $d = 2 * N - N = N$;
- $U - L = N - 1$;
- Thus, since $|d| > U - L$, no dependence exists.

Strong SIV test example

do $I = 1, N$
$S_1 : \quad A(I + 2 * N) = A(I + N) + B$
enddo

- The dependence distance: $d = 2 * N - N = N$;
- $U - L = N - 1$;
- Thus, since $|d| > U - L$, no dependence exists.

parallel do $I = 1, N$
$S_1 : \quad A(I + 2 * N) = A(I + N) + B$
enddo

Weak SIV test

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.

Weak SIV test

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for both, a subscript is SIV and involves the index i.

Weak SIV test

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for both, a subscript is SIV and involves the index i.
- We say that this SIV subscript is **weak** if the two index expressions are of the form $a_1\, i + c_1, a_2\, i' + c_2$, where a_1, a_2, c_1, c_2 are integers with $a_1 \neq a_2$ and non-zero.

Weak SIV test

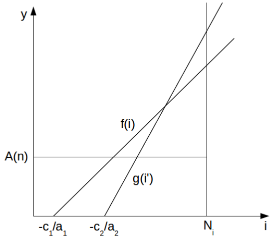
- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for both, a subscript is SIV and involves the index i.
- We say that this SIV subscript is **weak** if the two index expressions are of the form $a_1\, i + c_1, a_2\, i' + c_2$, where a_1, a_2, c_1, c_2 are integers with $a_1 \neq a_2$ and non-zero.
- The dependence equation is

$$a_1\, i + c_1 = a_2\, i' + c_2.$$

Weak SIV test

- Consider two statements S_1 and S_2 accessing an array A, one of them for writing.
- Suppose that for both, a subscript is SIV and involves the index i.
- We say that this SIV subscript is **weak** if the two index expressions are of the form $a_1\, i + c_1, a_2\, i' + c_2$, where a_1, a_2, c_1, c_2 are integers with $a_1 \neq a_2$ and non-zero.
- The dependence equation is

$$a_1\, i + c_1 = a_2\, i' + c_2.$$



Weak SIV test example (1/2)

- When $a_2 = -a_1$, the dependence equation becomes

$$i + i' = \frac{c_2 - c_1}{a_1}.$$

Weak SIV test example (1/2)

- When $a_2 = -a_1$, the dependence equation becomes

$$i + i' = \frac{c_2 - c_1}{a_1}.$$

- Thus a_1 must divide $c_2 - c_1$ in order to have a solution.

Weak SIV test example (1/2)

- When $a_2 = -a_1$, the dependence equation becomes

$$i + i' = \frac{c_2 - c_1}{a_1}.$$

- Thus a_1 must divide $c_2 - c_1$ in order to have a solution.
- With the example on the next slide, we have $i = n - i' + 1$ with $1 \leq i, i' \leq n$.

Weak SIV test example (1/2)

- When $a_2 = -a_1$, the dependence equation becomes

$$i + i' = \frac{c_2 - c_1}{a_1}.$$

- Thus a_1 must divide $c_2 - c_1$ in order to have a solution.
- With the example on the next slide, we have $i = n - i' + 1$ with $1 \leq i, i' \leq n$.
- This leads to

$$i + i' = n + 1$$

Weak SIV test example (1/2)

- When $a_2 = -a_1$, the dependence equation becomes
$$i + i' = \frac{c_2 - c_1}{a_1}.$$
- Thus a_1 must divide $c_2 - c_1$ in order to have a solution.
- With the example on the next slide, we have $i = n - i' + 1$ with $1 \leq i, i' \leq n$.
- This leads to
$$i + i' = n + 1$$
- To break this dependence (assuming n is even for simplicity) we split the loop into two: $1 \leq i \leq n/2$ and $n/2 + 1 \leq i \leq n$.

Weak SIV test example (2/2)

do $I = 1, N$
S_1 : $\quad A(I) = A(N - I + 1) + B$
enddo

Weak SIV test example (2/2)

do $I = 1, N$
$S_1 : \quad A(I) = A(N - I + 1) + B$
enddo

parallel do $I = 1, N/2$
$S_1 : \quad A(I) = A(N - I + 1) + B$
enddo parallel do $I = N/2 + 1, N$
$S_2 : \quad A(I) = A(N - I + 1) + B$
enddo

Symbolic SIV test

do $I = L_1, U_1$
$S_1 : \quad A(a_1 * I + c_1) = \dots$
enddo
do $J = L_2, U_2$
$S_2 : \quad A(a_2 * J + c_2) = \dots$
enddo

Symbolic SIV test

do $I = L_1, U_1$
$S_1 : \quad A(a_1 * I + c_1) = \ldots$
enddo
do $J = L_2, U_2$
$S_2 : \quad A(a_2 * J + c_2) = \ldots$
enddo

- A dependence exists if the following dependence equation is satisfied

$$a_1\, i - a_2\, j = c_2 - c_1,$$

for some index value of i, s.t. $L_1 \leq i \leq U_1$, and j, s.t. $L_2 \leq j \leq U_2$.

Symbolic SIV test

do $I = L_1, U_1$
$S_1 : \quad A(a_1 * I + c_1) = \dots$
enddo
do $J = L_2, U_2$
$S_2 : \quad A(a_2 * J + c_2) = \dots$
enddo

- A dependence exists if the following dependence equation is satisfied

$$a_1\, i - a_2\, j = c_2 - c_1,$$

  for some index value of i, s.t. $L_1 \leq i \leq U_1$, and j, s.t. $L_2 \leq j \leq U_2$.

- Hence, for $a_1 > 0$, there is a dependence only if

$$\begin{array}{ll} a_1\, L_1 - a_2\, U_2 \leq c_2 - c_1 \leq a_1\, U_1 - a_2\, L_2, & \text{if} \quad a_2 > 0; \\ a_1\, L_1 - a_2\, L_2 \leq c_2 - c_1 \leq a_1\, U_1 - a_2\, U_2, & \text{if} \quad a_2 < 0. \end{array}$$

Symbolic SIV test: idea of the proof

- The solutions of the **linear Diophantine** equation

$$a_1\, x - b_1\, y = b_0 - a_0$$

provide the values i, j of the index variable of the previous symbolic SIV test.

Symbolic SIV test: idea of the proof

- The solutions of the **linear Diophantine** equation

$$a_1\, x - b_1\, y = b_0 - a_0$$

  provide the values i, j of the index variable of the previous symbolic SIV test.
- Let e_a and e_b be two values such that $e_a\, a_1 + e_b\, b_1 = \gcd(a_1, b_1)$.

Symbolic SIV test: idea of the proof

- The solutions of the **linear Diophantine** equation

$$a_1\, x - b_1\, y = b_0 - a_0$$

  provide the values i, j of the index variable of the previous symbolic SIV test.
- Let e_a and e_b be two values such that $e_a\, a_1 + e_b\, b_1 = \gcd(a_1, b_1)$.
- The solutions of the Diophantine equation are given by

$$\begin{array}{rcl} x_k & = & e_a \left(\frac{b_0 - a_0}{g}\right) + k\, \frac{b_1}{g} \\ y_k & = & e_b \left(\frac{b_0 - a_0}{g}\right) + k\, \frac{a_1}{g} \end{array}$$

Symbolic SIV test: idea of the proof

- The solutions of the **linear Diophantine** equation

$$a_1\, x - b_1\, y = b_0 - a_0$$

  provide the values i, j of the index variable of the previous symbolic SIV test.
- Let e_a and e_b be two values such that $e_a\, a_1 + e_b\, b_1 = \gcd(a_1, b_1)$.
- The solutions of the Diophantine equation are given by

$$\begin{array}{rcl} x_k & = & e_a\left(\frac{b_0 - a_0}{g}\right) + k\,\frac{b_1}{g} \\ y_k & = & e_b\left(\frac{b_0 - a_0}{g}\right) + k\,\frac{a_1}{g} \end{array}$$

- For dependence to exist, these solutions must occur in the region defined by the loop bounds.

Outline

Key notions (1/2)

- The *polyhedral model* is mathematical framework for analyzing, scheduling and optimizing for-loop nests.

do $i = 0, N-1$
  do $j = 0, N-1$
S_1 : $A[i,j] += u[i] * v[i]$
  enddo
enddo
do $k = 0, N-1$
  do $\ell = 0, N-1$
S_2 : $x[k] += A[\ell, k] * y[\ell]$
  enddo
enddo

Iteration domain

$$0 \leq i < N$$
$$0 \leq j < N$$
$$0 \leq k < N$$
$$0 \leq \ell < N$$

Dependence equation

$$i = \ell$$
$$j = k$$

Key notions (1/2)

- The *polyhedral model* is mathematical framework for analyzing, scheduling and optimizing for-loop nests.
- It views the iterations of a for-loop nest as the integer points of a polyhedral set.

do $i = 0, N-1$
  do $j = 0, N-1$
S_1 : $A[i,j] += u[i] * v[i]$
  enddo
enddo
do $k = 0, N-1$
  do $\ell = 0, N-1$
S_2 : $x[k] += A[\ell, k] * y[\ell]$
  enddo
enddo

Iteration domain

$$0 \le i < N$$
$$0 \le j < N$$
$$0 \le k < N$$
$$0 \le \ell < N$$

Dependence equation

$$i = \ell$$
$$j = k$$

Key notions (1/2)

- The *polyhedral model* is mathematical framework for analyzing, scheduling and optimizing for-loop nests.
- It views the iterations of a for-loop nest as the integer points of a polyhedral set.
- It makes a number of natural assumptions, in particular: every *array reference* is an affine expression in the loop counters, loop bounds, array dimension sizes and possibly other constants.

do $i = 0, N-1$
  do $j = 0, N-1$
S_1 : $A[i,j] += u[i] * v[i]$
  enddo
enddo
do $k = 0, N-1$
  do $\ell = 0, N-1$
S_2 : $x[k] += A[\ell,k] * y[\ell]$
  enddo
enddo

Iteration domain

$$0 \leq i < N$$
$$0 \leq j < N$$
$$0 \leq k < N$$
$$0 \leq \ell < N$$

Dependence equation

$$i = \ell$$
$$j = k$$

Key notions (2/2)

- The *iteration domain* is defined by the value ranges of the loop counters.

Iteration domain

$$0 \le i < N$$
$$0 \le j < N$$
$$0 \le k < N$$
$$0 \le \ell < N$$

Dependence equation

$$i = \ell$$
$$j = k$$

Dependence polyhedron for $S_1 \to S_2$

$$\left(\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 \\ \hline 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \end{array}\right) \begin{bmatrix} i \\ j \\ k \\ \ell \\ N \\ 1 \end{bmatrix} \begin{array}{c} \\ \\ \ge 0 \\ \\ \hline = 0 \\ \\ \end{array}$$

Key notions (2/2)

- The *iteration domain* is defined by the value ranges of the loop counters.
- The *dependence equations* are deduced from the array references of the pair of statements under study.

Iteration domain

$$
\begin{aligned}
0 \le i < N \\
0 \le j < N \\
0 \le k < N \\
0 \le \ell < N
\end{aligned}
$$

Dependence equation

$$
\begin{aligned}
i = \ell \\
j = k
\end{aligned}
$$

Dependence polyhedron for $S_1 \to S_2$

$$
\left(\begin{array}{cccccc}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 1 & -1 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 1 & -1 \\
\hline
1 & 0 & 0 & -1 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0
\end{array}\right)
\begin{bmatrix} i \\ j \\ k \\ \ell \\ N \\ 1 \end{bmatrix}
\begin{array}{c} \\ \\ \ge 0 \\ \\ \hline = 0 \\ \\ \end{array}
$$

Key notions (2/2)

- The *iteration domain* is defined by the value ranges of the loop counters.
- The *dependence equations* are deduced from the array references of the pair of statements under study.
- The *dependence polyhedron* collects the equality and inequality constraints from the iteration domain and the dependence equations. Obviously, this polyhedron has integer points, this there is an output dependence.

Iteration domain

$$0 \le i < N$$
$$0 \le j < N$$
$$0 \le k < N$$
$$0 \le \ell < N$$

Dependence equation

$$i = \ell$$
$$j = k$$

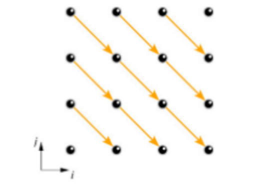
Dependence polyhedron for $S_1 \to S_2$

$$\left(\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 \\ \hline 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \end{array}\right) \begin{bmatrix} i \\ j \\ k \\ \ell \\ N \\ 1 \end{bmatrix} \begin{array}{c} \\ \\ \ge 0 \\ \\ \hline = 0 \\ \\ \end{array}$$

Automatic parallelization: plain multiplication (1/2)

Serial dense univariate polynomial multiplication

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

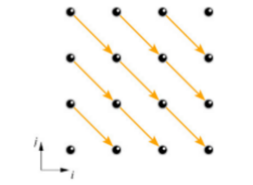


- Can 2 different iterations access the same memory location at least one for writing?

Automatic parallelization: plain multiplication (1/2)

Serial dense univariate polynomial multiplication

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

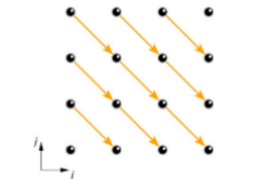


- Can 2 different iterations access the same memory location at least one for writing?
- Given an integer $p \geq 2$, every (i, j) satisfying $i + j = p$ will access $c[p]$ in writing.

Automatic parallelization: plain multiplication (1/2)

Serial dense univariate polynomial multiplication

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

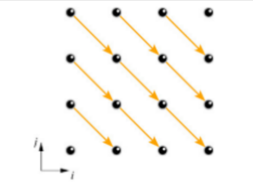


- Can 2 different iterations access the same memory location at least one for writing?
- Given an integer $p \geq 2$, every (i, j) satisfying $i + j = p$ will access $c[p]$ in writing.
- However, for 2 integers $q > p \geq 2$, an iteration (i, j) satisfying $i + j = p$ and an iteration (k, ℓ) satisfying $k + \ell = q$ won't access the same location in the array c.

Automatic parallelization: plain multiplication (1/2)

Serial dense univariate polynomial multiplication

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

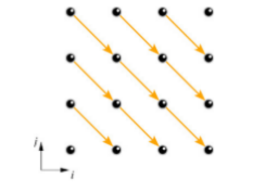


- Can 2 different iterations access the same memory location at least one for writing?
- Given an integer $p \geq 2$, every (i, j) satisfying $i + j = p$ will access $c[p]$ in writing.
- However, for 2 integers $q > p \geq 2$, an iteration (i, j) satisfying $i + j = p$ and an iteration (k, ℓ) satisfying $k + \ell = q$ won't access the same location in the array c.
- Hence, this suggests a change of coordinates with one new coordinate being $p = i + j$, where p stands for *processor*. What can be the second coordinate?

Automatic parallelization: plain multiplication (1/2)

Serial dense univariate polynomial multiplication

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

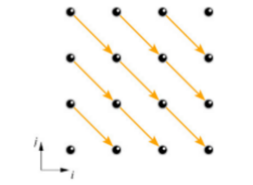


- Can 2 different iterations access the same memory location at least one for writing?
- Given an integer $p \geq 2$, every (i, j) satisfying $i + j = p$ will access $c[p]$ in writing.
- However, for 2 integers $q > p \geq 2$, an iteration (i, j) satisfying $i + j = p$ and an iteration (k, ℓ) satisfying $k + \ell = q$ won't access the same location in the array c.
- Hence, this suggests a change of coordinates with one new coordinate being $p = i + j$, where p stands for *processor*. What can be the second coordinate?
- Call it t for *time*. The map $M : (i, j) \longmapsto (p, t)$ must be linear, given by a unimodular matrix (thus with determinant equal to 1 or -1).

Automatic parallelization: plain multiplication (1/2)

Serial dense univariate polynomial multiplication

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

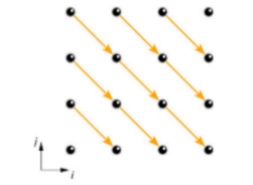


- Can 2 different iterations access the same memory location at least one for writing?
- Given an integer $p \geq 2$, every (i, j) satisfying $i + j = p$ will access $c[p]$ in writing.
- However, for 2 integers $q > p \geq 2$, an iteration (i, j) satisfying $i + j = p$ and an iteration (k, ℓ) satisfying $k + \ell = q$ won't access the same location in the array c.
- Hence, this suggests a change of coordinates with one new coordinate being $p = i + j$, where p stands for *processor*. What can be the second coordinate?
- Call it t for *time*. The map $M : (i, j) \longmapsto (p, t)$ must be linear, given by a unimodular matrix (thus with determinant equal to 1 or -1).
- In general, such a map should also preserve order between iterations, that is, $(i, j) < (k, \ell)$ must imply $M(i, j) < M(k, \ell)$.

Automatic parallelization: plain multiplication (1/2)

Serial dense univariate polynomial multiplication

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



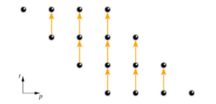
- Can 2 different iterations access the same memory location at least one for writing?
- Given an integer $p \geq 2$, every (i, j) satisfying $i + j = p$ will access $c[p]$ in writing.
- However, for 2 integers $q > p \geq 2$, an iteration (i, j) satisfying $i + j = p$ and an iteration (k, ℓ) satisfying $k + \ell = q$ won't access the same location in the array c.
- Hence, this suggests a change of coordinates with one new coordinate being $p = i + j$, where p stands for *processor*. What can be the second coordinate?
- Call it t for *time*. The map $M : (i, j) \longmapsto (p, t)$ must be linear, given by a unimodular matrix (thus with determinant equal to 1 or -1).
- In general, such a map should also preserve order between iterations, that is, $(i, j) < (k, \ell)$ must imply $M(i, j) < M(k, \ell)$.
- One possible choice is $t(i, j) = n - j$.

Automatic parallelization: plain multiplication (2/2)

- What should be the loop bounds in the new system of coordinates?

Asynchronous parallel dense univariate polynomial multiplication

```
parallel_for (p=0; p<=2*n; p++){
  c [ p ] =0;
  for (t=max(0,n-p); t<= min(n,2*n-p)
    C [ p ]  = C [ p ]
     + A [ t+p-n ]  * B [ n-t ] ;
}
```

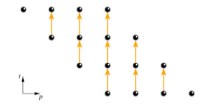


Automatic parallelization: plain multiplication (2/2)

- What should be the loop bounds in the new system of coordinates?
- On the left, see our input iteration domain with the coordinate change

Asynchronous parallel dense univariate polynomial multiplication

```
parallel_for (p=0; p<=2*n; p++){
  c [ p ] =0;
  for (t=max(0,n-p); t<= min(n,2*n-p)
    C [ p ]  = C [ p ]
     + A [ t+p-n ]  * B [ n-t ] ;
}
```



Automatic parallelization: plain multiplication (2/2)

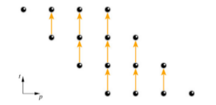
- What should be the loop bounds in the new system of coordinates?
- On the left, see our input iteration domain with the coordinate change
- Some linear transformations produces the new iteration domain with the inverse of our change of coordinates

$$\begin{cases} i & \geq & 0 \\ i & \leq & n \\ j & \geq & 0 \\ j & \leq & n \\ t & = & n-j \\ p & = & i+j, \end{cases}$$

$$\begin{cases} i & = & p+t-n \\ j & = & -t+n \\ t & \geq & -p+n \\ t & \leq & -p+2n \\ n & \geq & t \\ 0 & \leq & t \\ p & \geq & 0 \\ p & \leq & 2n. \end{cases}$$

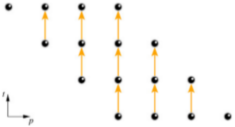
Asynchronous parallel dense univariate polynomial multiplication

```
parallel_for (p=0; p<=2*n; p++){
  c [ p ] =0;
  for (t=max(0,n-p); t<= min(n,2*n-p)
    C [ p ]  = C [ p ]
     + A [ t+p-n ]  * B [ n-t ] ;
}
```



Generating parametric code & use of tiling techniques

```
parallel_for (p=0; p<=2*n; p++){
  c [ p ] =0;
  for (t=max(0,n-p); t<= min(n,2*n-p);t++)
    C [ p ]  = C [ p ]
      + A [ t+p-n ]  * B [ n-t ] ;
}
```

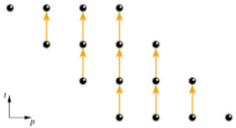


Improving the parallelization

- The above generated code is not practical for multicore implementation: the number of processors is in $\Theta(n)$. (Not to mention poor locality!) and the work is unevenly distributed among the workers.

Generating parametric code & use of tiling techniques

```
parallel_for (p=0; p<=2*n; p++){
  c [ p ] =0;
  for (t=max(0,n-p); t<= min(n,2*n-p);t++)
    C [ p ]  = C [ p ]
      + A [ t+p-n ]  * B [ n-t ] ;
}
```

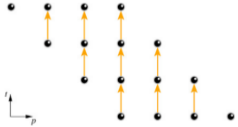


Improving the parallelization

- The above generated code is not practical for multicore implementation: the number of processors is in $\Theta(n)$. (Not to mention poor locality!) and the work is unevenly distributed among the workers.
- We group the virtual processors (or threads) into 1D blocks, each of size B. Each thread is known by its block number b and a local coordinate u in its block.

Generating parametric code & use of tiling techniques

```
parallel_for (p=0; p<=2*n; p++){
  c [ p ] =0;
  for (t=max(0,n-p); t<= min(n,2*n-p);t++)
    C [ p ]  = C [ p ]
      + A [ t+p-n ]  * B [ n-t ] ;
}
```

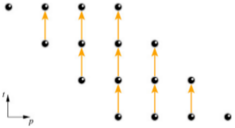


Improving the parallelization

- The above generated code is not practical for multicore implementation: the number of processors is in $\Theta(n)$. (Not to mention poor locality!) and the work is unevenly distributed among the workers.
- We group the virtual processors (or threads) into 1D blocks, each of size B. Each thread is known by its block number b and a local coordinate u in its block.
- Blocks represent good units of work which have good locality property.

Generating parametric code & use of tiling techniques

```
parallel_for (p=0; p<=2*n; p++){
  c [ p ] =0;
  for (t=max(0,n-p); t<= min(n,2*n-p);t++)
    C [ p ]  = C [ p ]
      + A [ t+p-n ]  * B [ n-t ] ;
}
```



Improving the parallelization

- The above generated code is not practical for multicore implementation: the number of processors is in $\Theta(n)$. (Not to mention poor locality!) and the work is unevenly distributed among the workers.
- We group the virtual processors (or threads) into 1D blocks, each of size B. Each thread is known by its block number b and a local coordinate u in its block.
- Blocks represent good units of work which have good locality property.
- This yields the following constraints: $0 \le u < B$, $p = bB + u$.

Generating parametric code: using tiles

We apply `RegularChains:-QuantifierElimination` on the left system (in order to get rid off i, j) leading to the relations on the right:

$$\left\{\begin{array}{c} o < n \\ 0 \le i \le n \\ 0 \le j \le n \\ t = n - j \\ p = i + j \\ 0 \le b \\ o \le u < B \\ p = bB + u, \end{array}\right. \qquad \left\{\begin{array}{c} B > 0 \\ n > 0 \\ 0 \le b \le 2n/B \\ 0 \le u < B \\ 0 \le u \le 2n - Bb \\ p = bB + u, \end{array}\right. \tag{3}$$

From where we derive the following program:

```
for (p=0; p<=2*n; p++) c [ p ] = 0;
parallel_for (b=0; b<= 2 n / B; b++) {
    parallel_for (u=0; u<=min(B-1, 2*n - B * b); u++) {
       p = b * B + u;
       for (t=max(0,n-p); t<=min(n,2*n-p) ;t++)
           c [ p ] = c [ p ] + a [ t+p-n ] * b [ n-t ];
   }
}
```

References

[1] C. Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. PACT '04. IEEE Computer Society, 2004, pp. 7–16.

[2] A. Größlinger, M. Griebl, and C. Lengauer. "Quantifier elimination in automatic loop parallelization". In: *J. Symb. Comput.* 41.11 (2006), pp. 1206–1221.