Workshop on Essential Abstractions in GCC

Parallelization and Vectorization in GCC

GCC Resource Centre

(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

July 02 2013

Outline

- Transformation for parallel and vector execution
- Data dependence
- Auto-parallelization and auto-vectorization in gcc-4.7.2
- Conclusions

The Scope of This Lecture

- What this lecture addresses
  - Basic notion of dependence, vectorization and parallelization
  - GCC's approach of discovering and exploiting parallelism
  - Illustrated using carefully chosen examples
- What this lecture does not address
  - Details of algorithms, code and data structures used for parallelization and vectorization
  - Machine level issues related to parallelization and vectorization

Part 1

Transformations for Parallel and Vector Execution

Vectorization: SISD ⇒ SIMD

- Parallelism in executing same operation on multiple operands

  (8-bit, 16-bit, 32-bit operands)

- Existing 32 or 64-bit arithmetic units used to perform multiple operations in parallel

  A 64 bit word ≡ a vector of 2×(32 bits), 4×(16 bits), or 8×(8 bits)

- Vector registers are of length 128 bit and size of int is 32 bit

Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

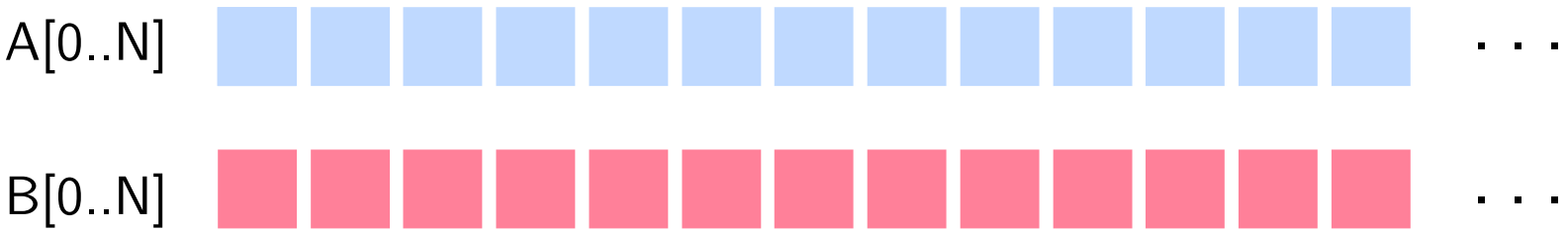
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



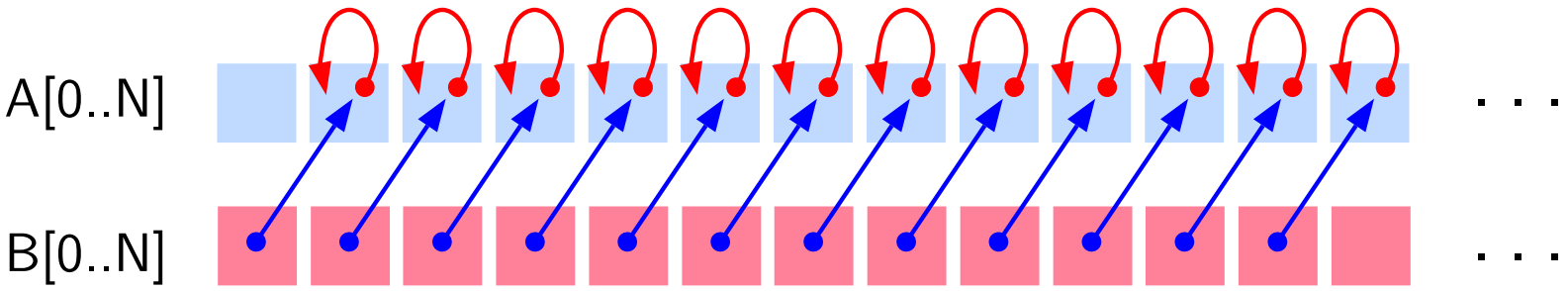
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



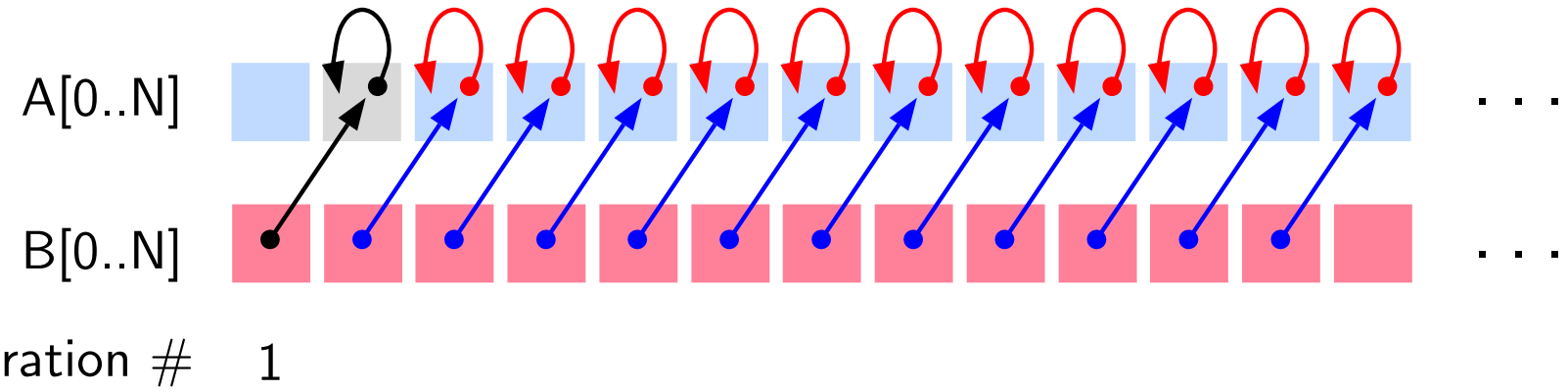
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



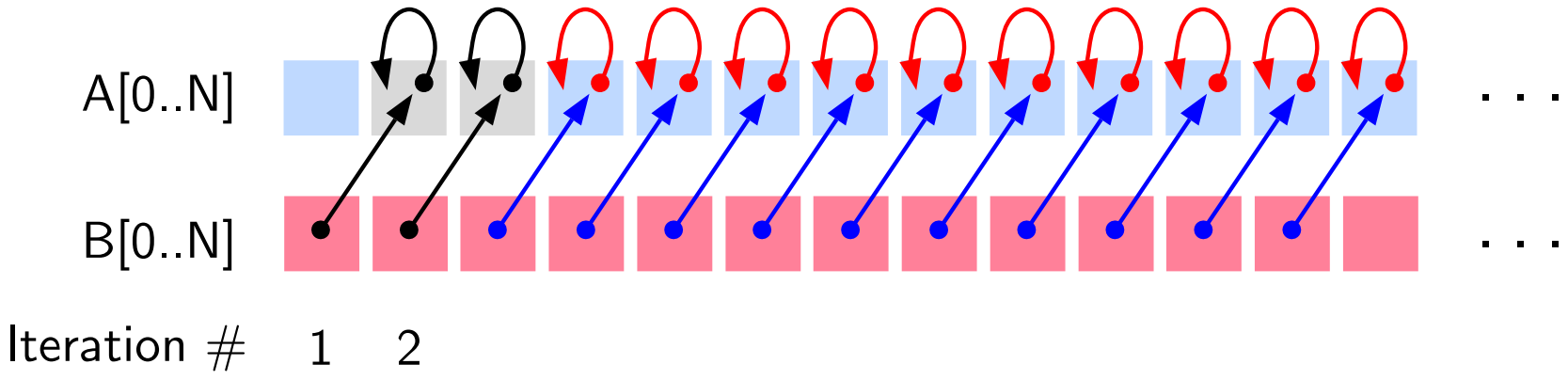
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



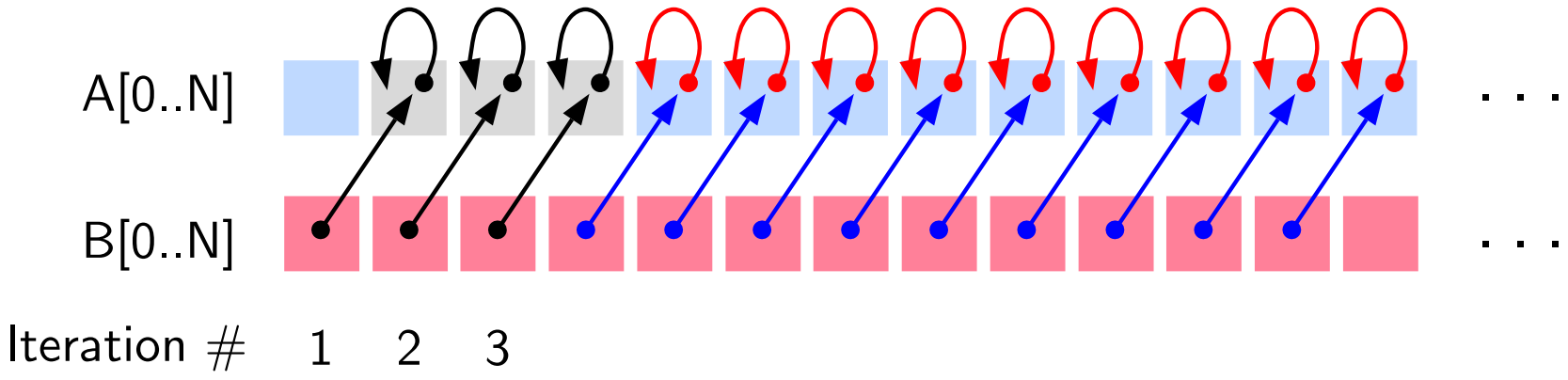
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



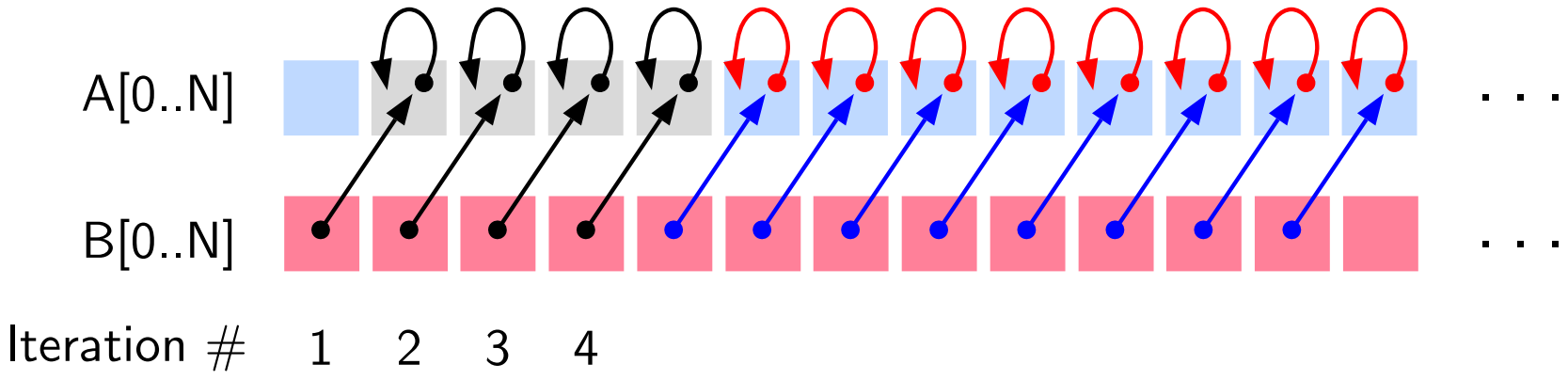
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



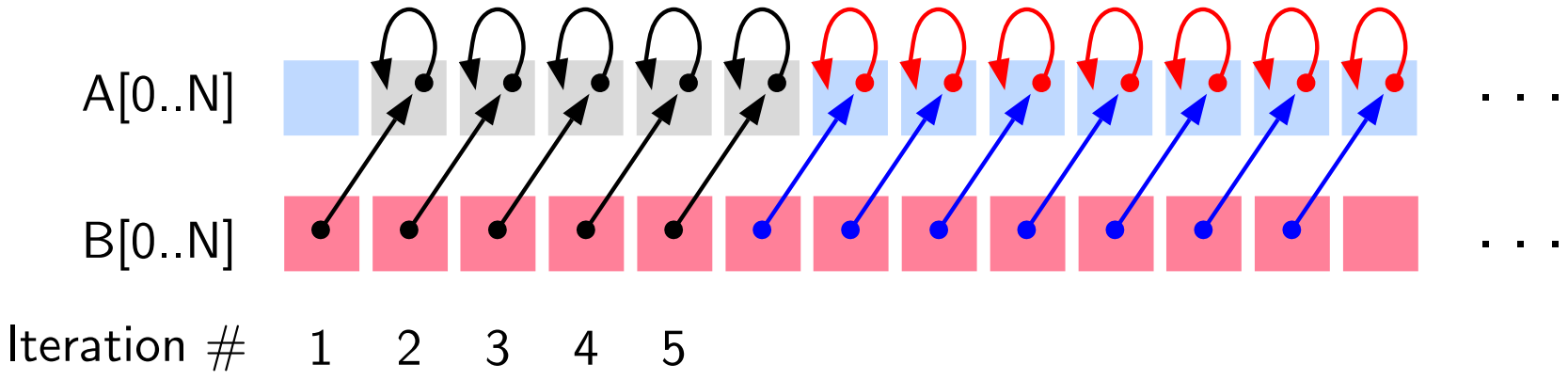
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



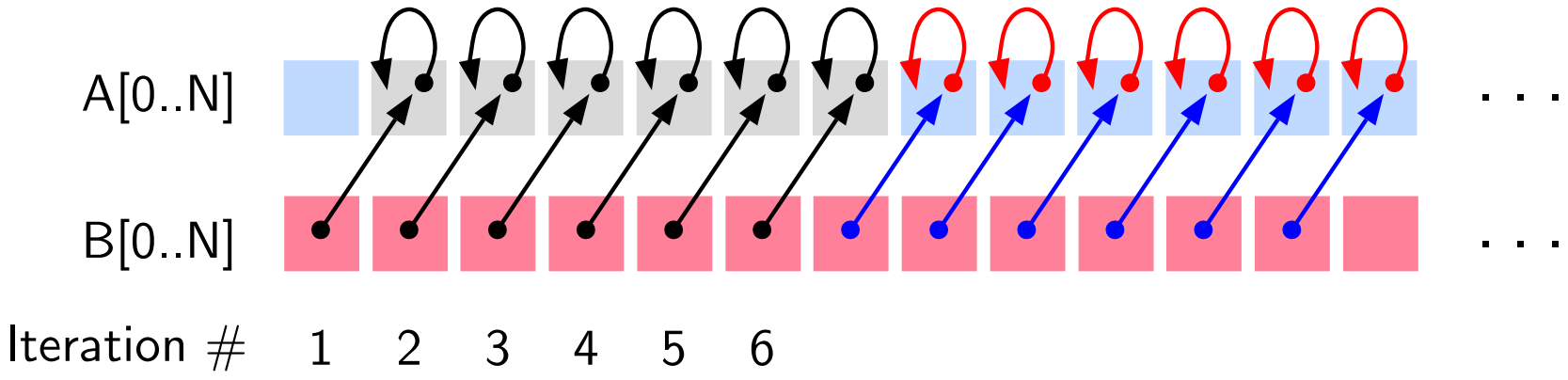
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



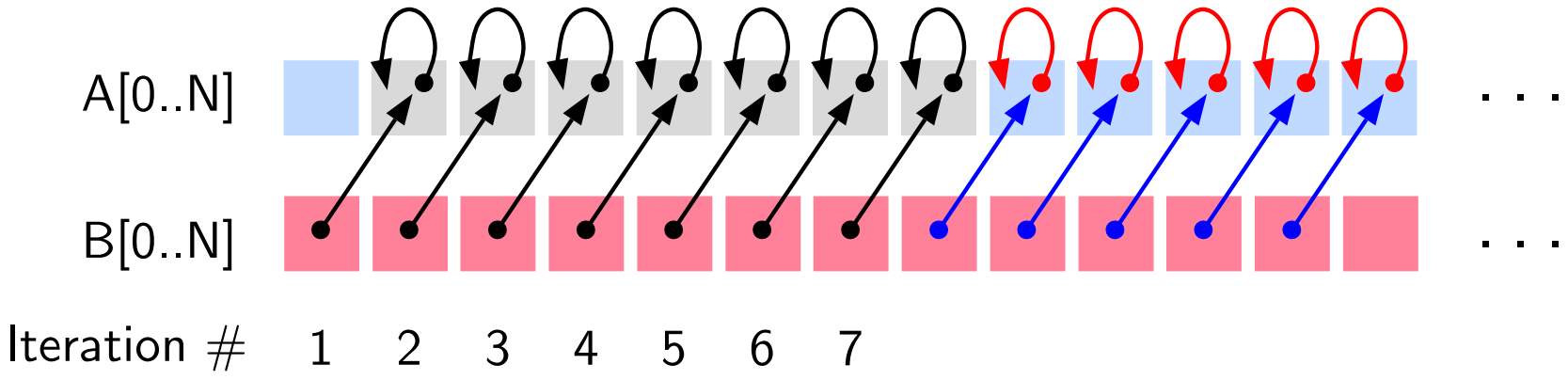
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



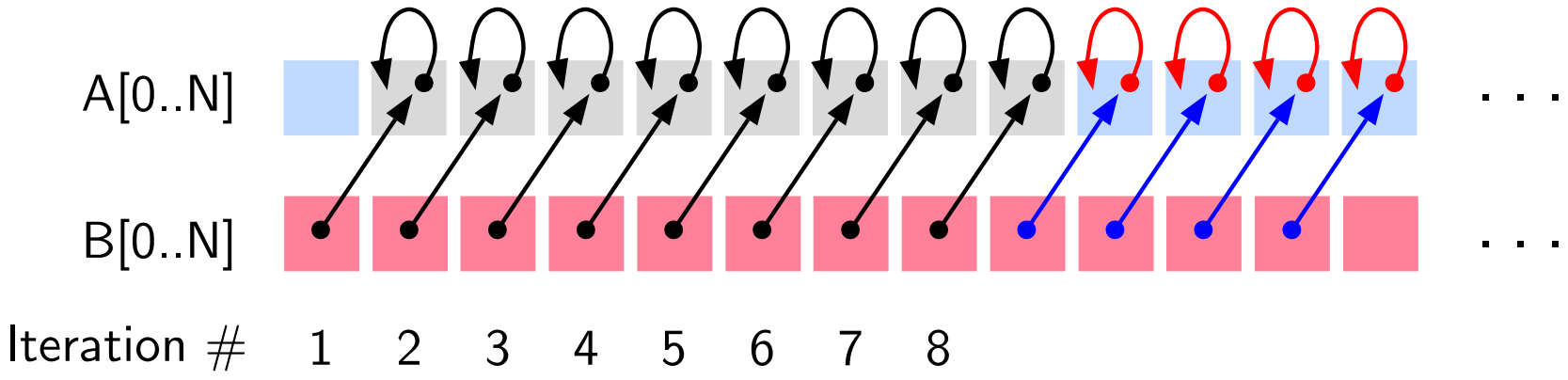
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



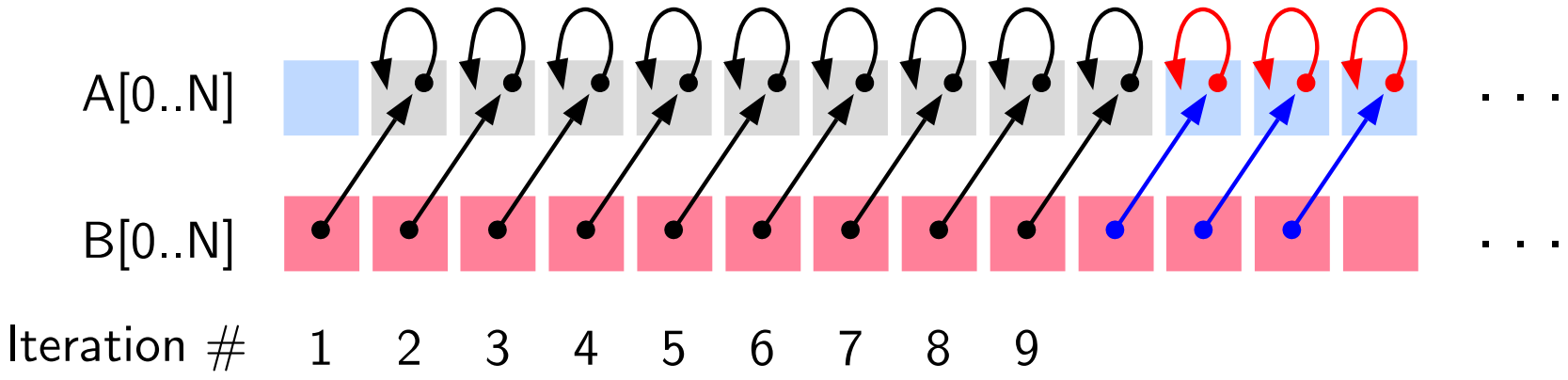
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



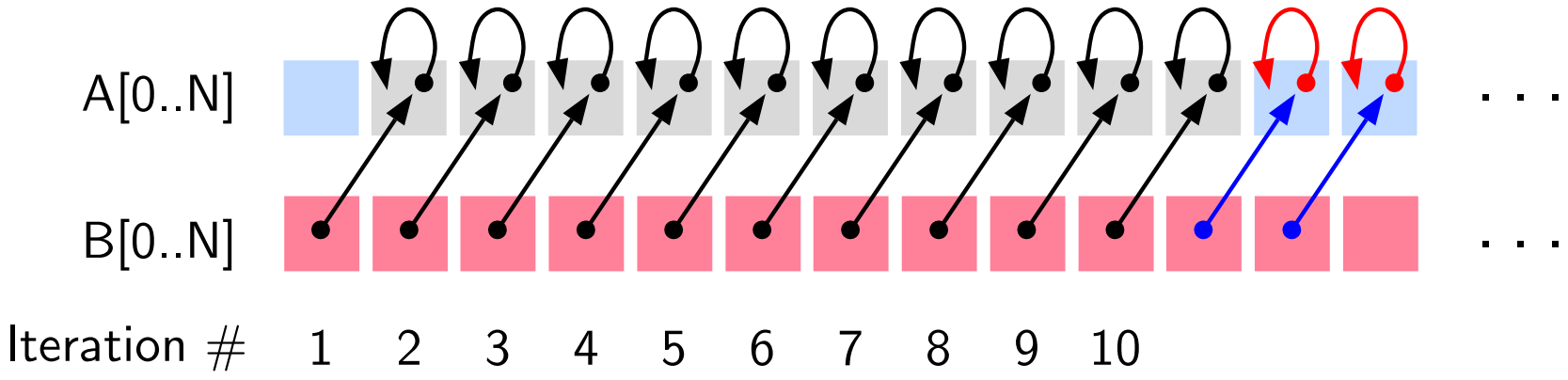
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



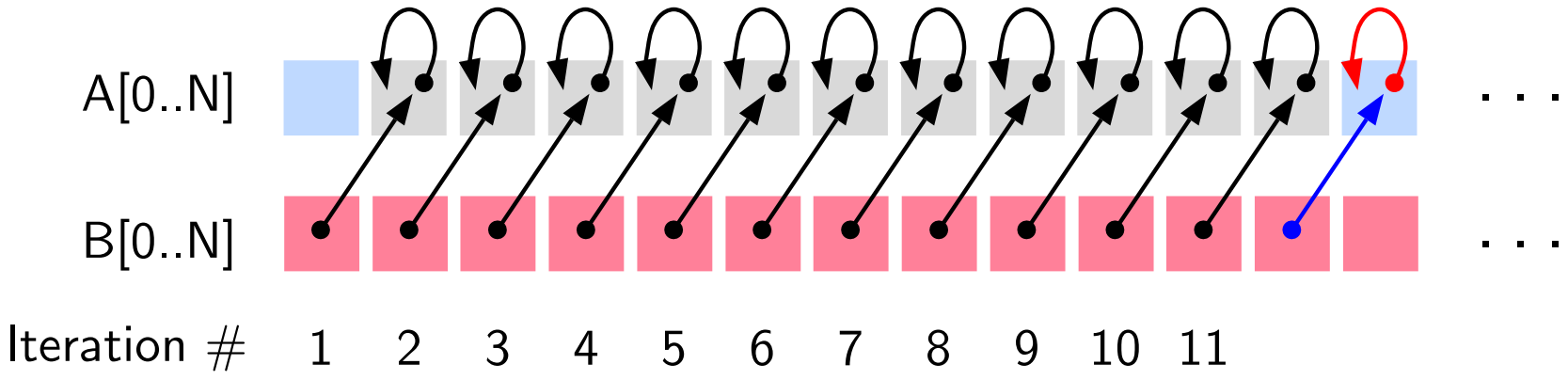
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



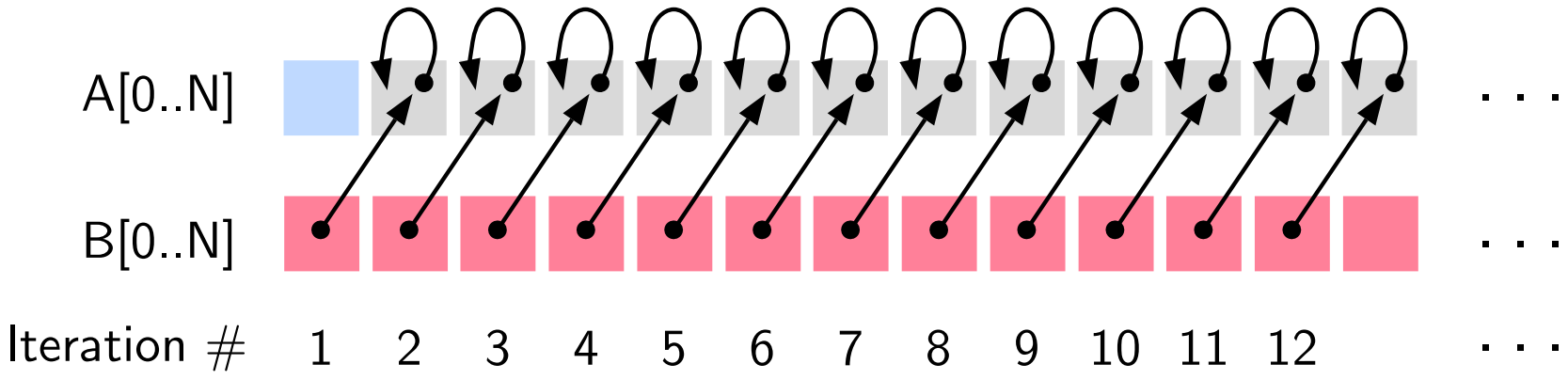
Example 1

Vectorization (SISD ⇒ SIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



Example 1

Vectorization (SISD ⇒ SIMD) : Yes

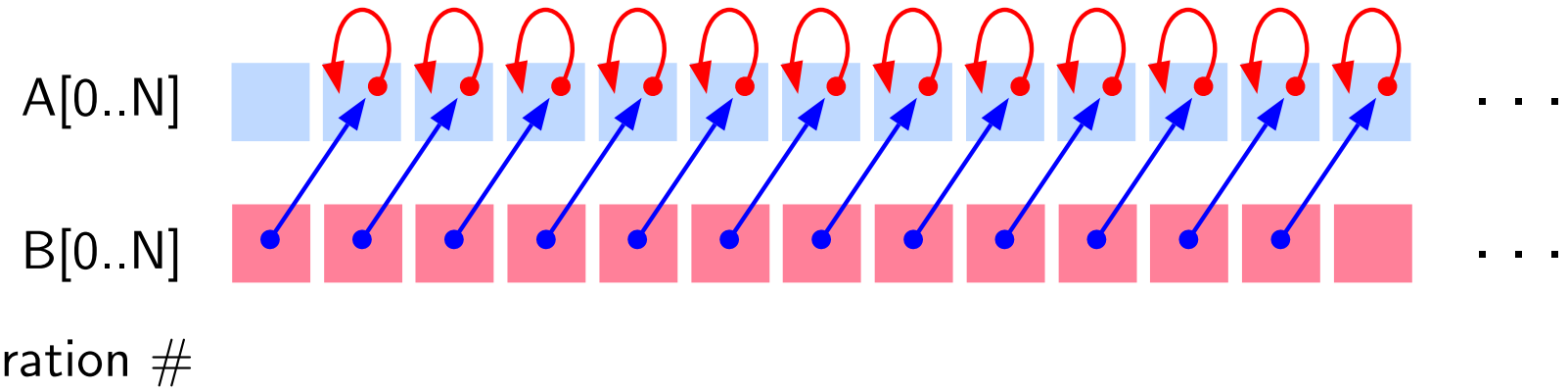
Vectorization Factor

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```



Example 1

Vectorization (SISD ⇒ SIMD) : Yes

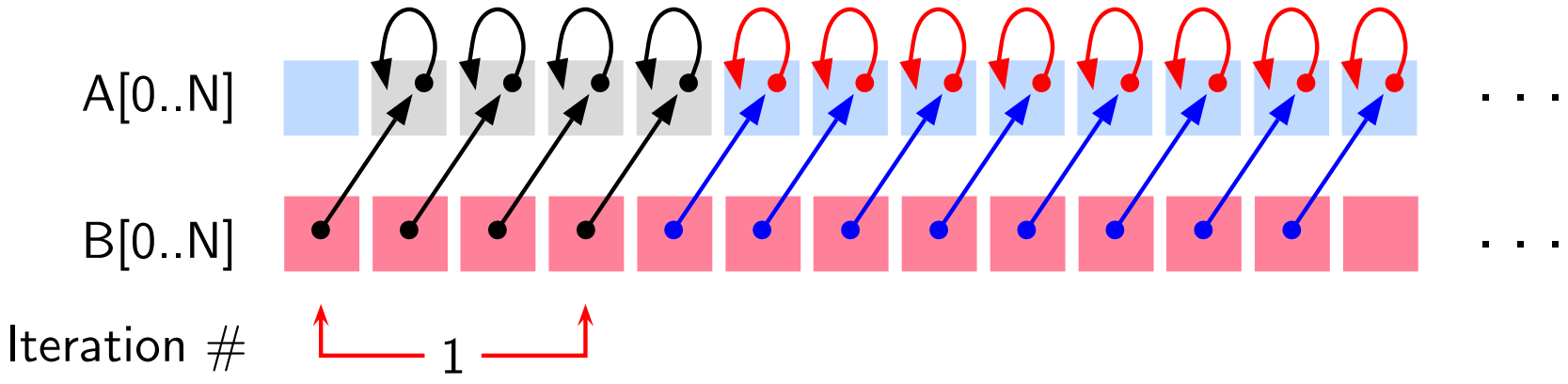
Vectorization Factor

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```



Example 1

Vectorization (SISD ⇒ SIMD) : Yes

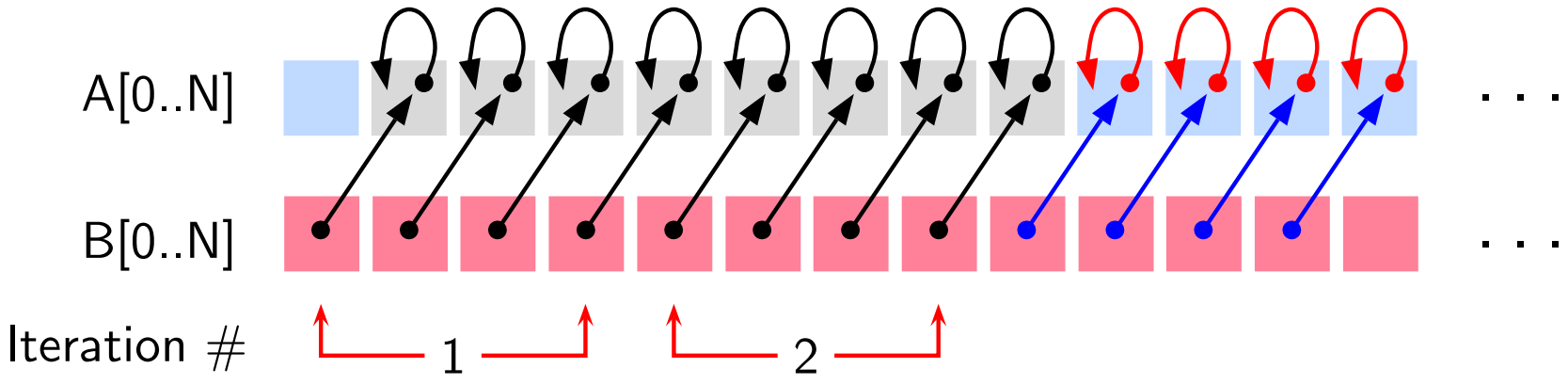
Vectorization Factor

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```



Example 1

Vectorization (SISD ⇒ SIMD) : Yes

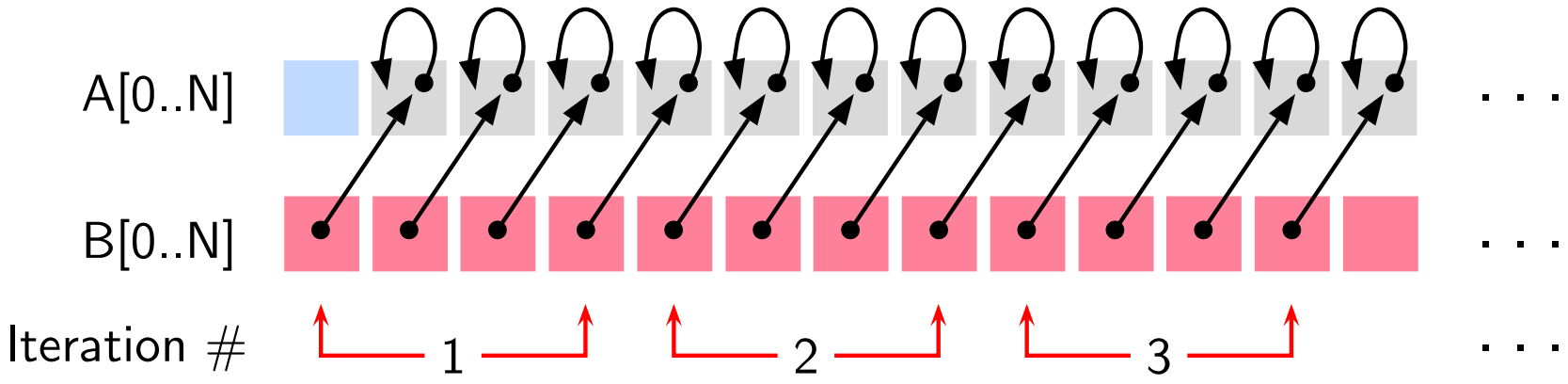
Vectorization Factor

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```



Example 1

Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

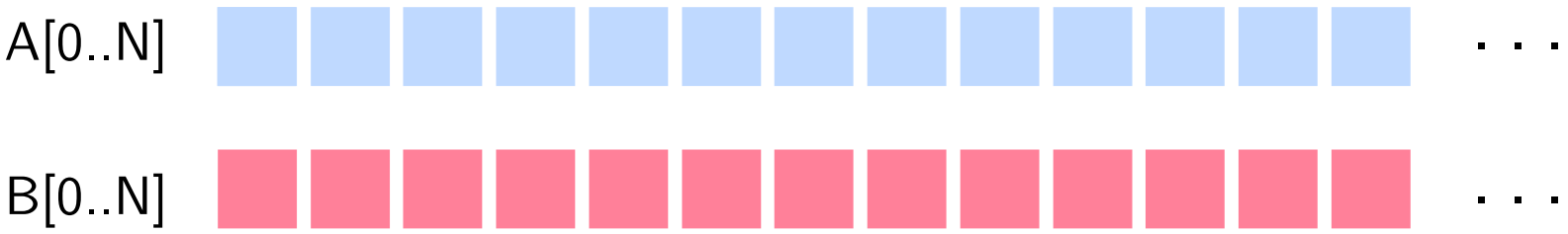
Example 1

Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



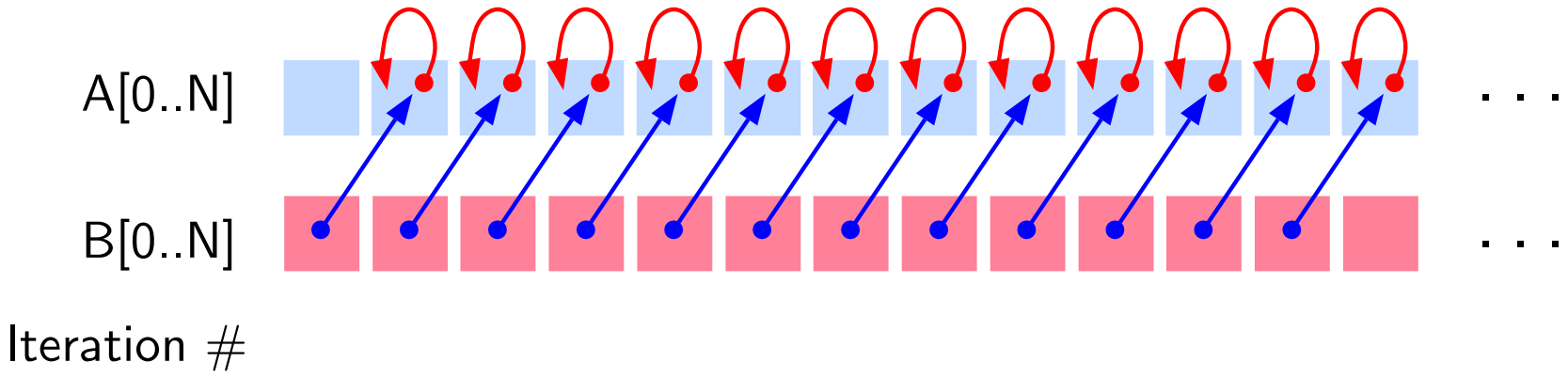
Example 1

Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



Example 1

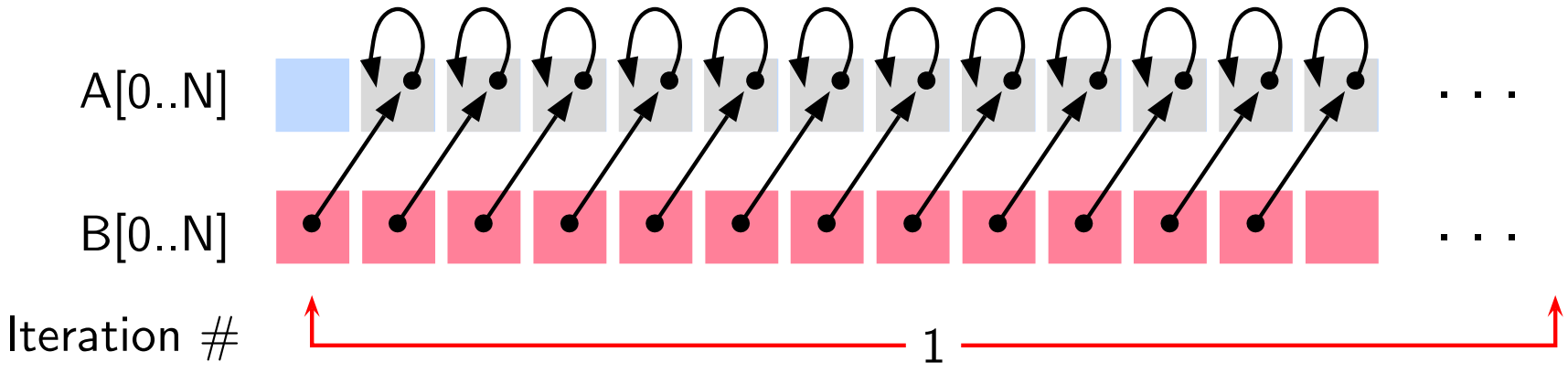
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Parallelized Code

```
int A[N], B[N], i;
for-all (i=1 to N)
  A[i] = A[i] + B[i-1];
```

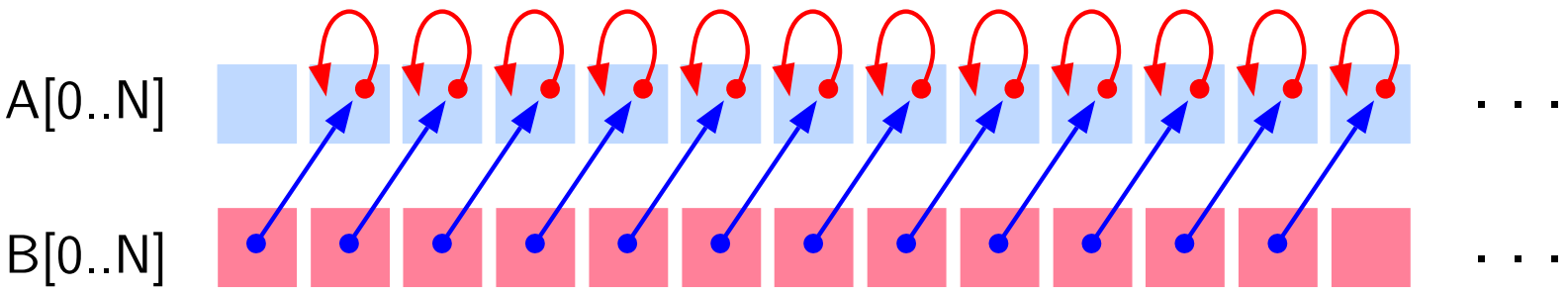


Example 1: The Moral of the Story

Vectorization	(SISD $\Rightarrow$ SIMD)	: Yes
Parallelization	(SISD $\Rightarrow$ MIMD)	: Yes

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



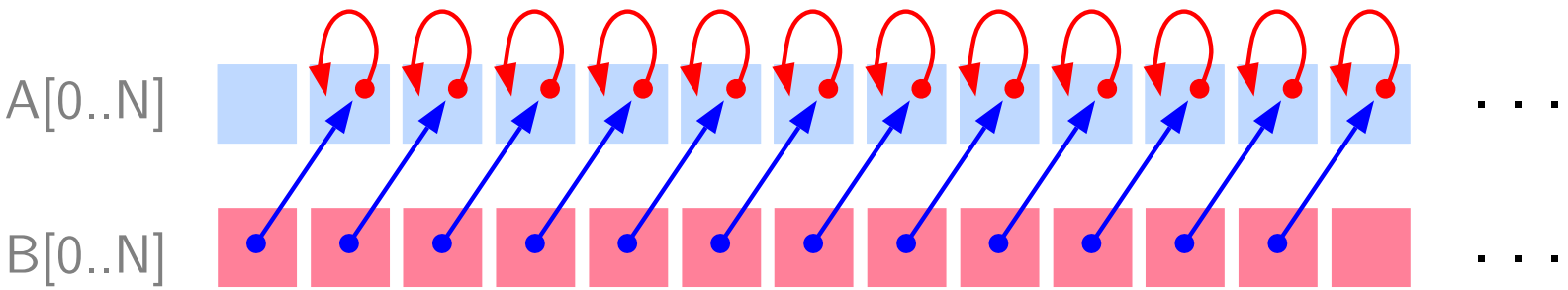
Example 1: The Moral of the Story

Vectorization	(SISD $\Rightarrow$ SIMD)	: Yes
Parallelization	(SISD $\Rightarrow$ MIMD)	: Yes

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	
Write	Read	
Write	Write	
Read	Read	



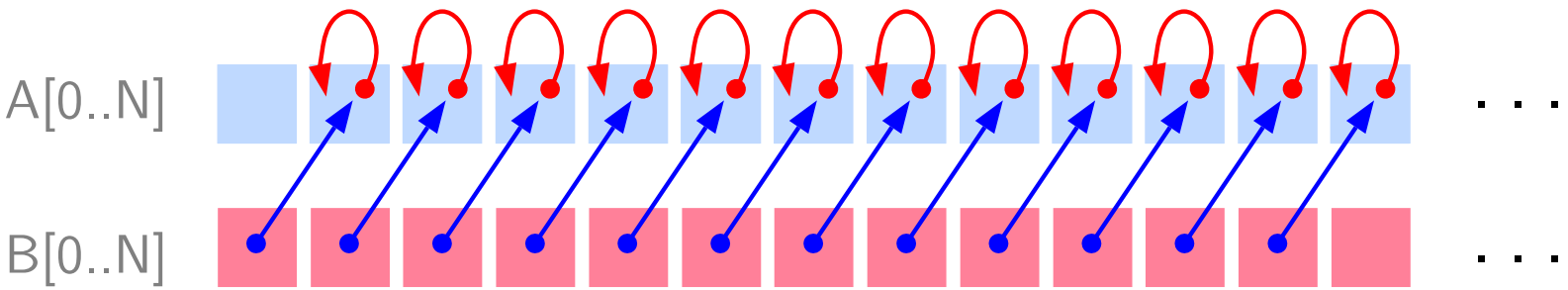
Example 1: The Moral of the Story

Vectorization	(SISD $\Rightarrow$ SIMD)	: Yes
Parallelization	(SISD $\Rightarrow$ MIMD)	: Yes

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	No
Write	Read	
Write	Write	
Read	Read	



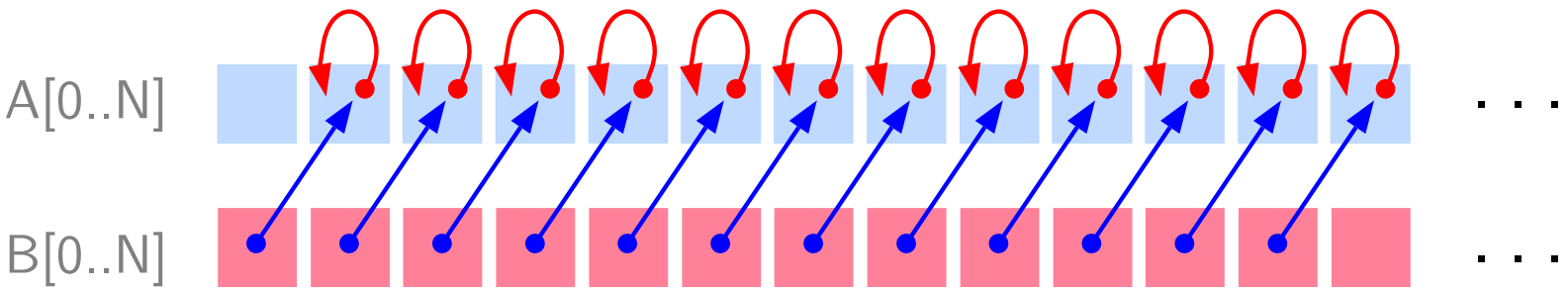
Example 1: The Moral of the Story

Vectorization	(SISD $\Rightarrow$ SIMD)	: Yes
Parallelization	(SISD $\Rightarrow$ MIMD)	: Yes

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	No
Write	Read	No
Write	Write	
Read	Read	



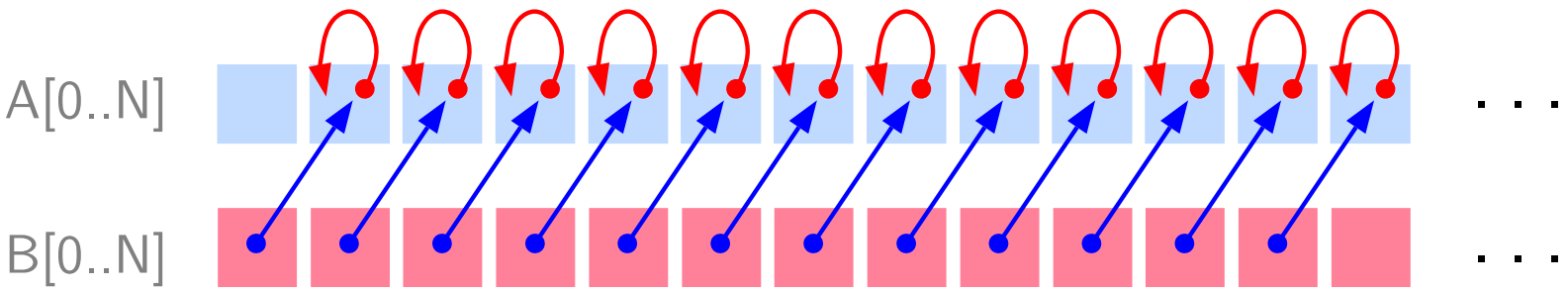
Example 1: The Moral of the Story

Vectorization	(SISD $\Rightarrow$ SIMD)	: Yes
Parallelization	(SISD $\Rightarrow$ MIMD)	: Yes

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	No
Write	Read	No
Write	Write	No
Read	Read	



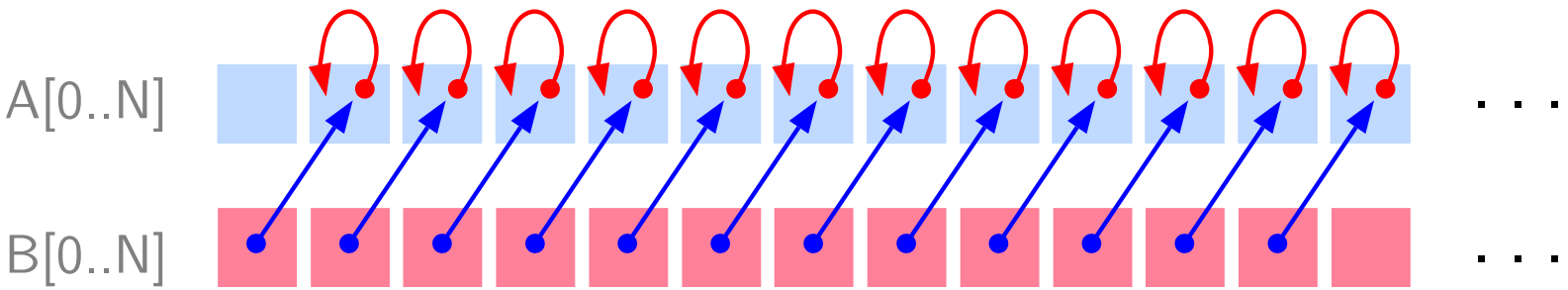
Example 1: The Moral of the Story

Vectorization	(SISD $\Rightarrow$ SIMD)	: Yes
Parallelization	(SISD $\Rightarrow$ MIMD)	: Yes

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	No
Write	Read	No
Write	Write	No
Read	Read	Does not matter



Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

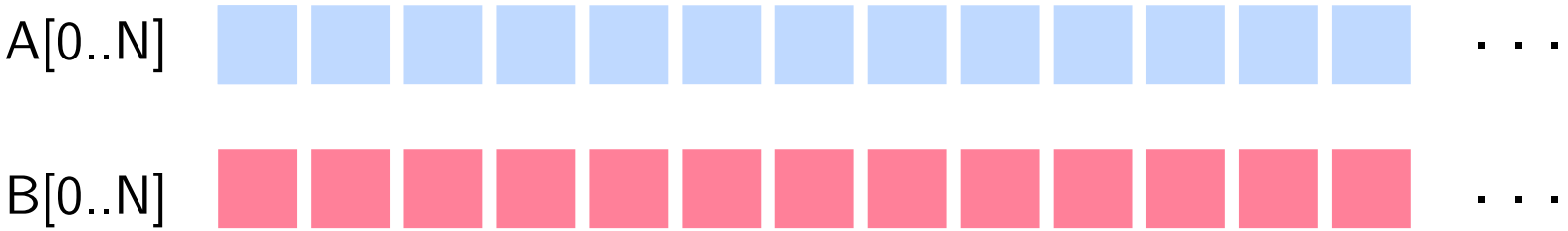
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



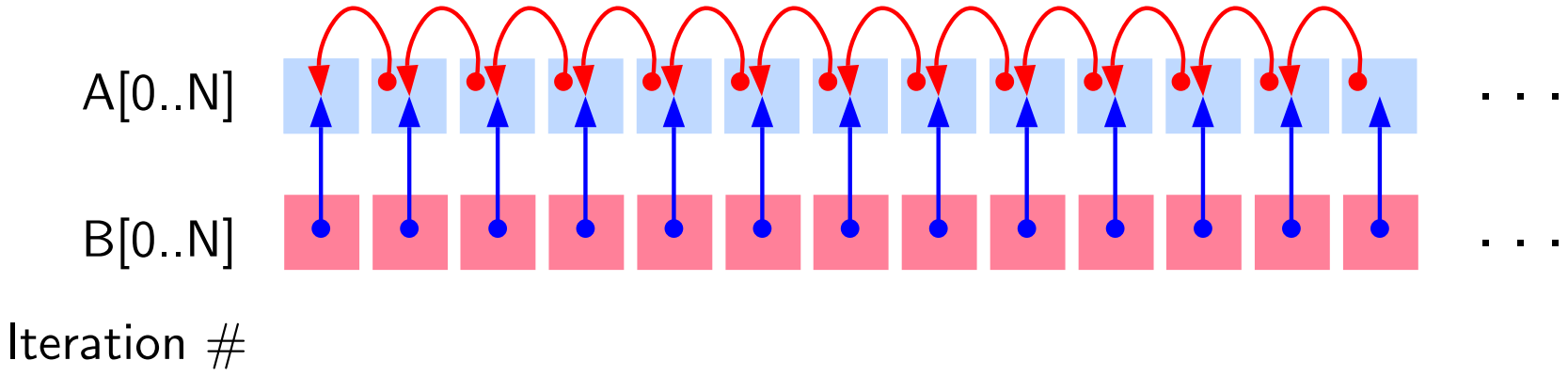
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



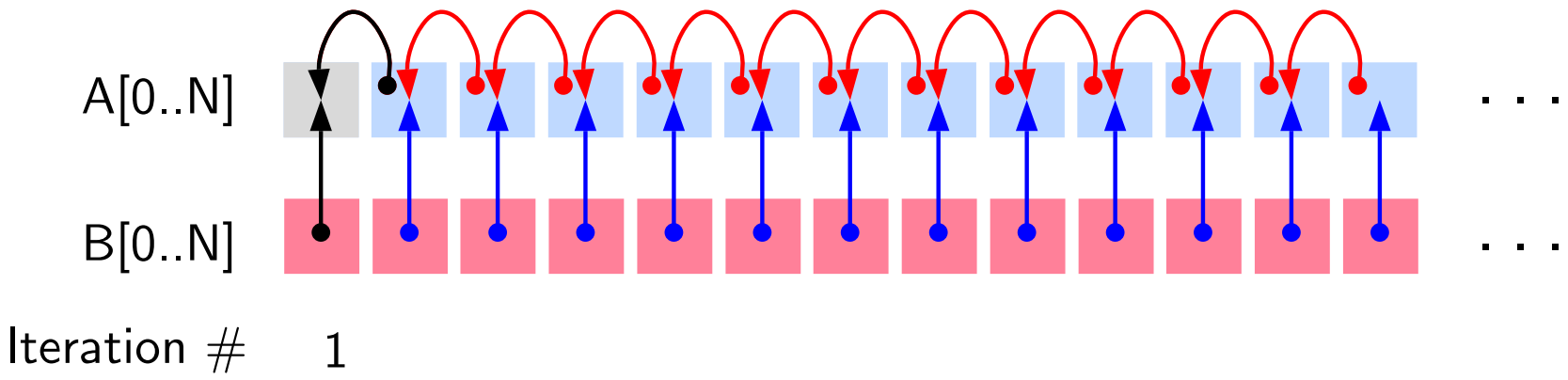
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



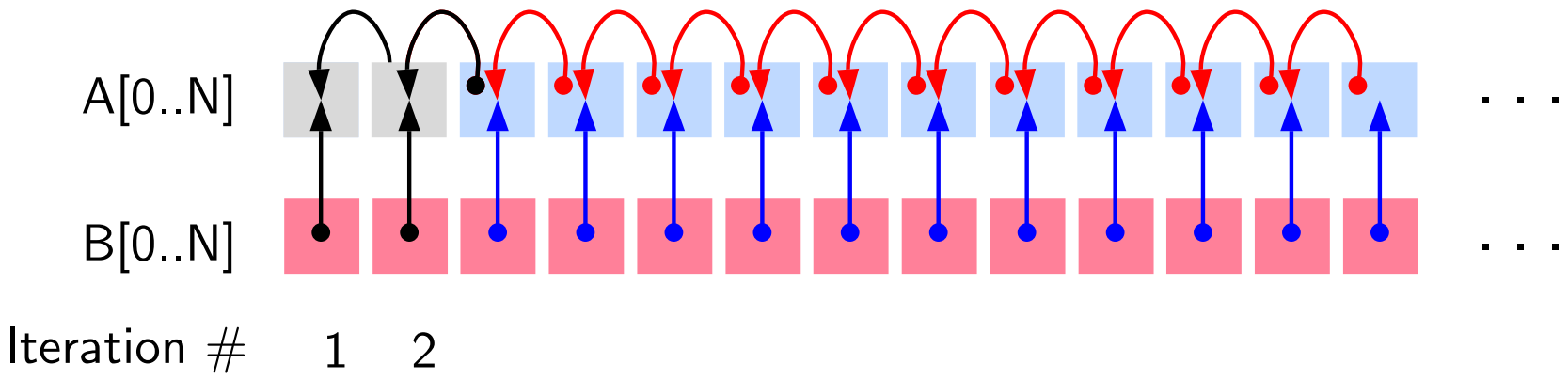
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



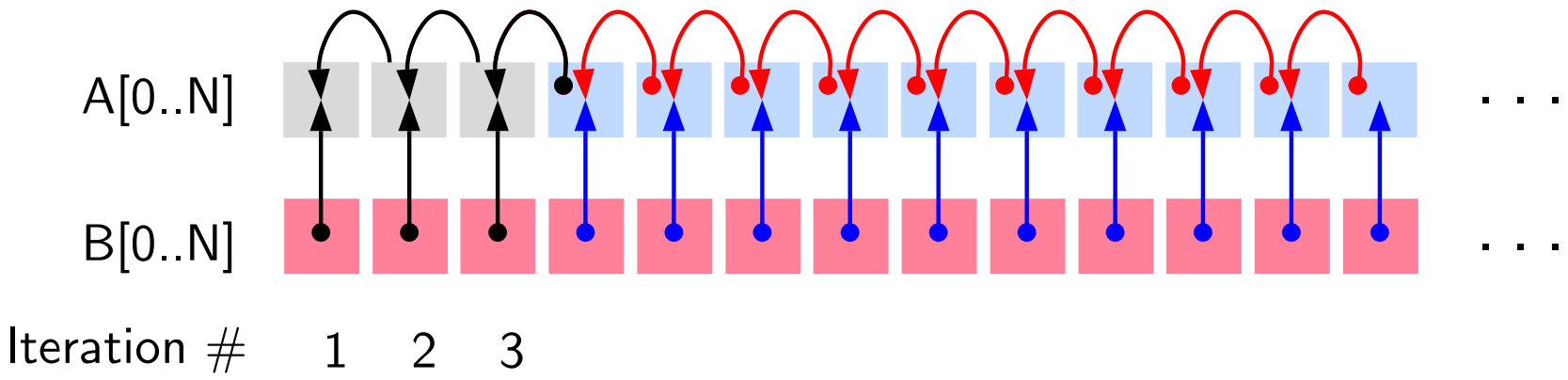
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



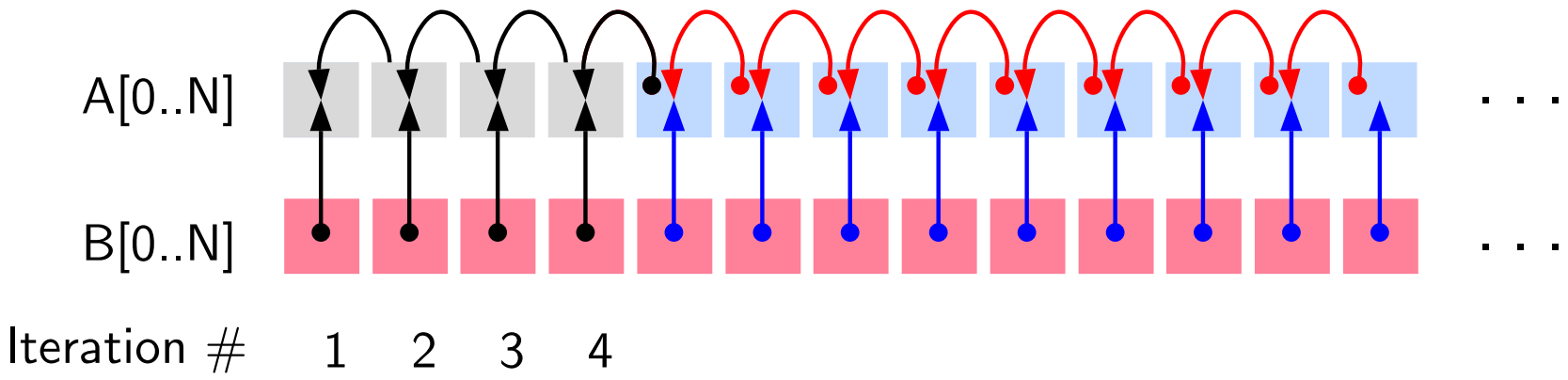
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



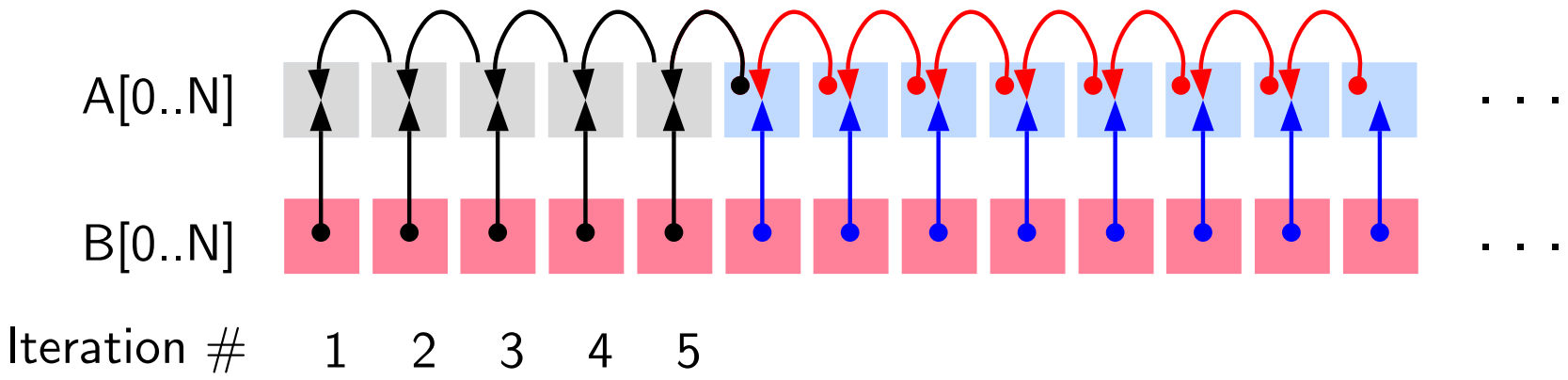
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



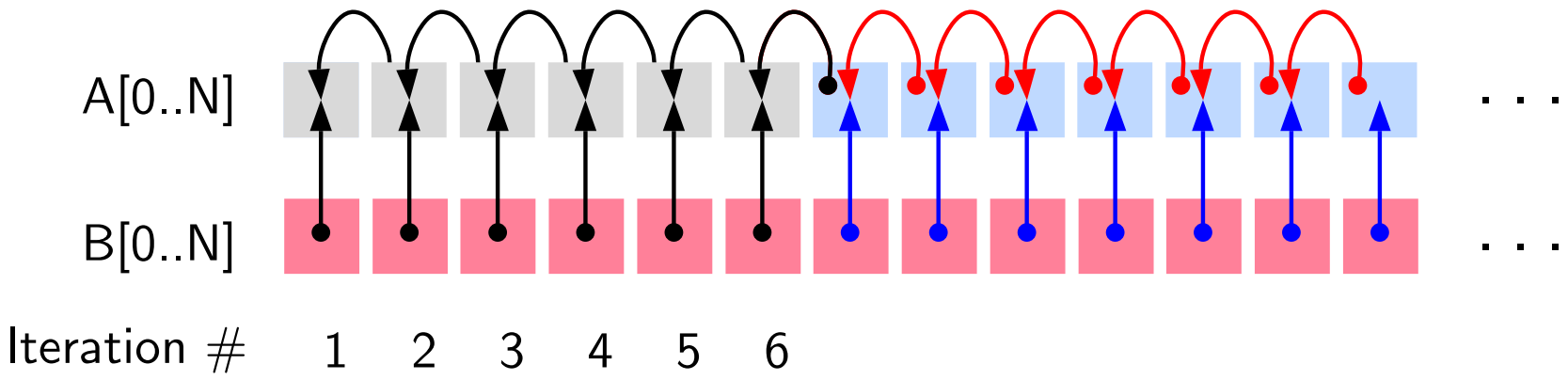
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



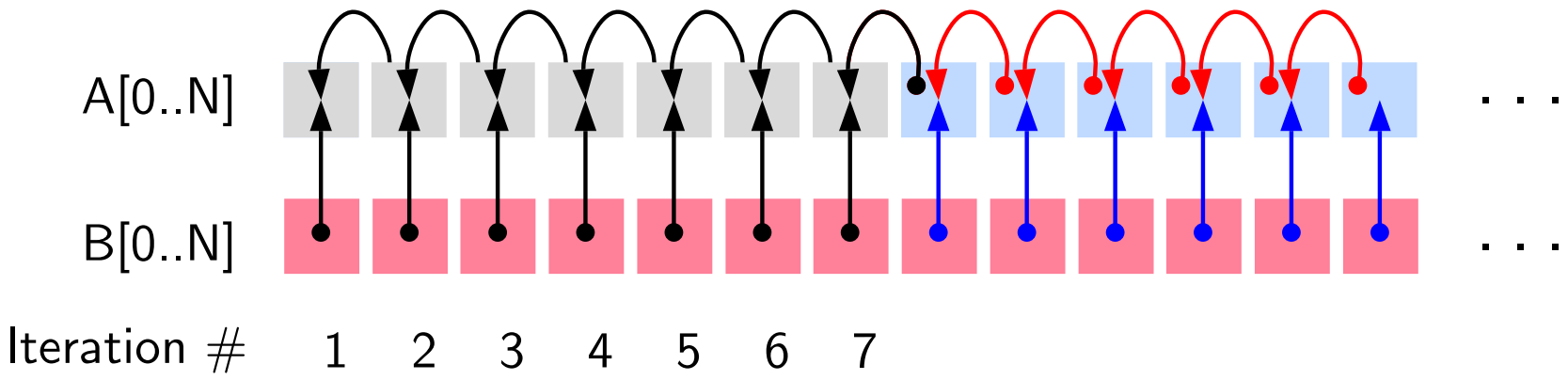
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



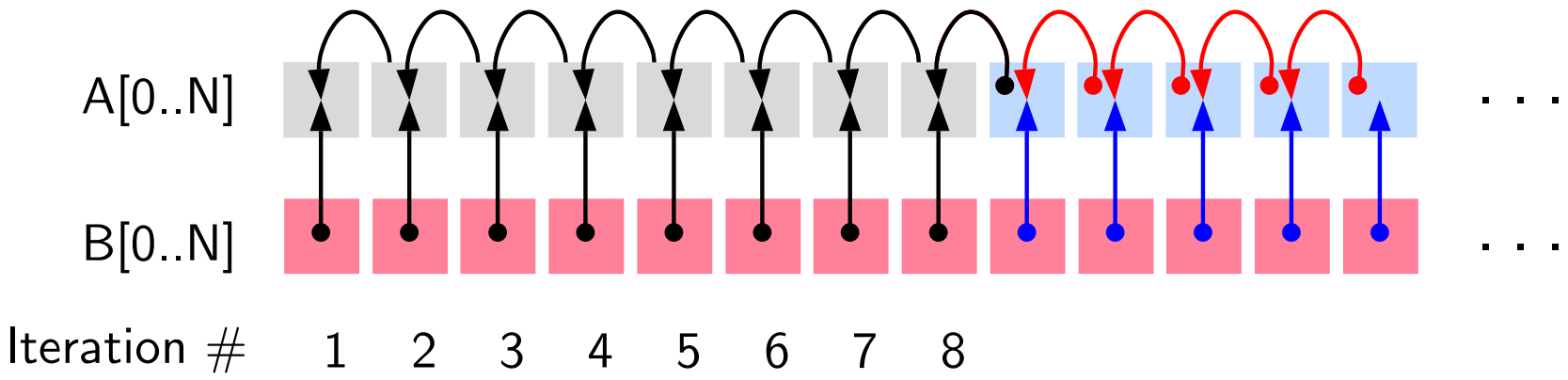
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



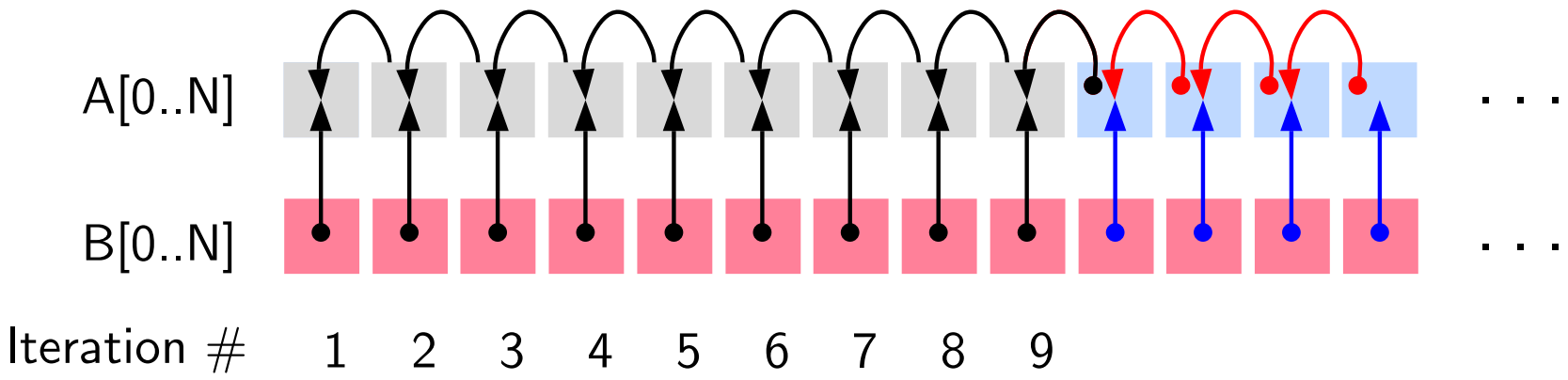
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



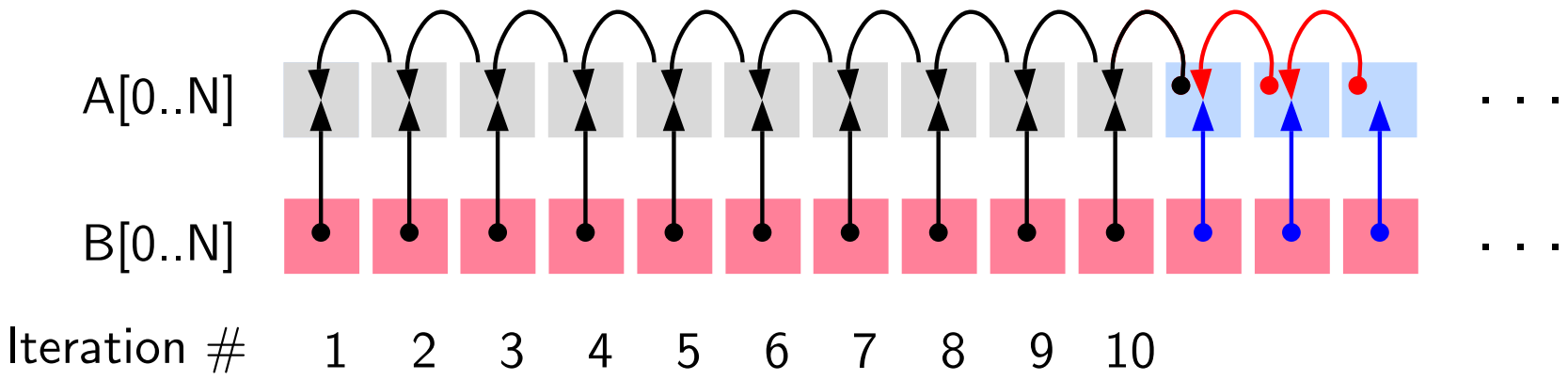
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



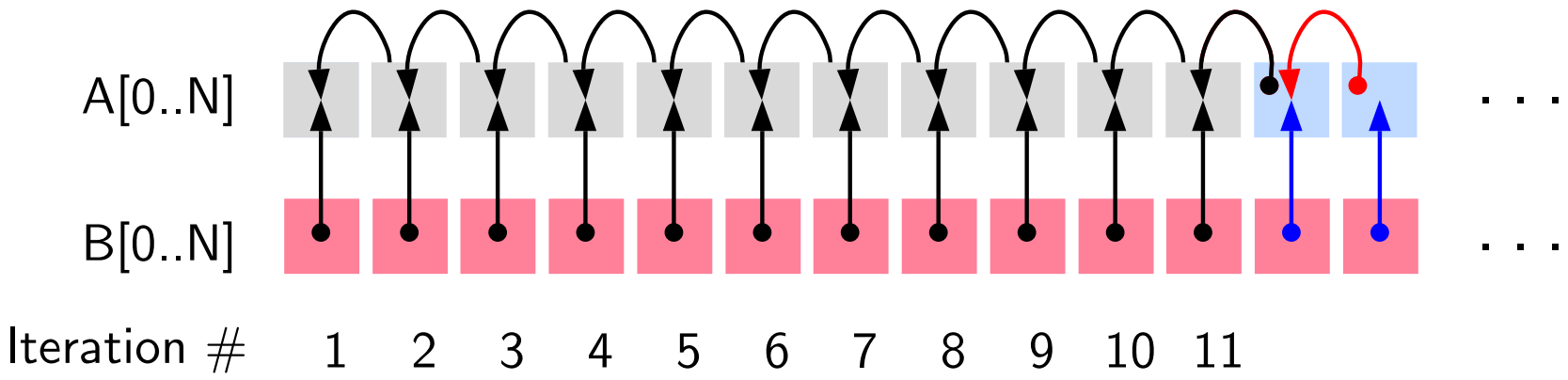
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



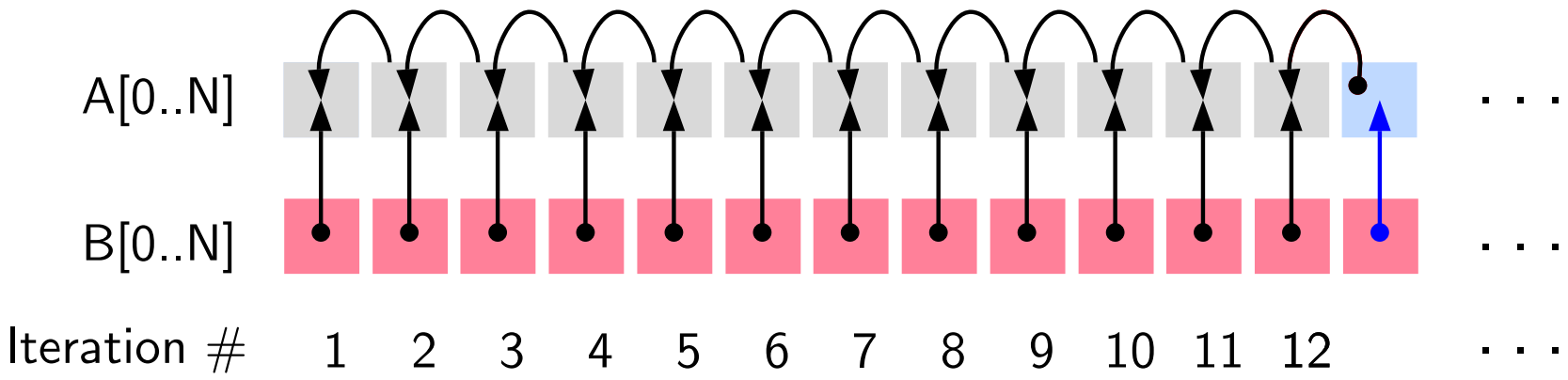
Example 2

Vectorization (SISD ⇒ SIMD) : ??
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



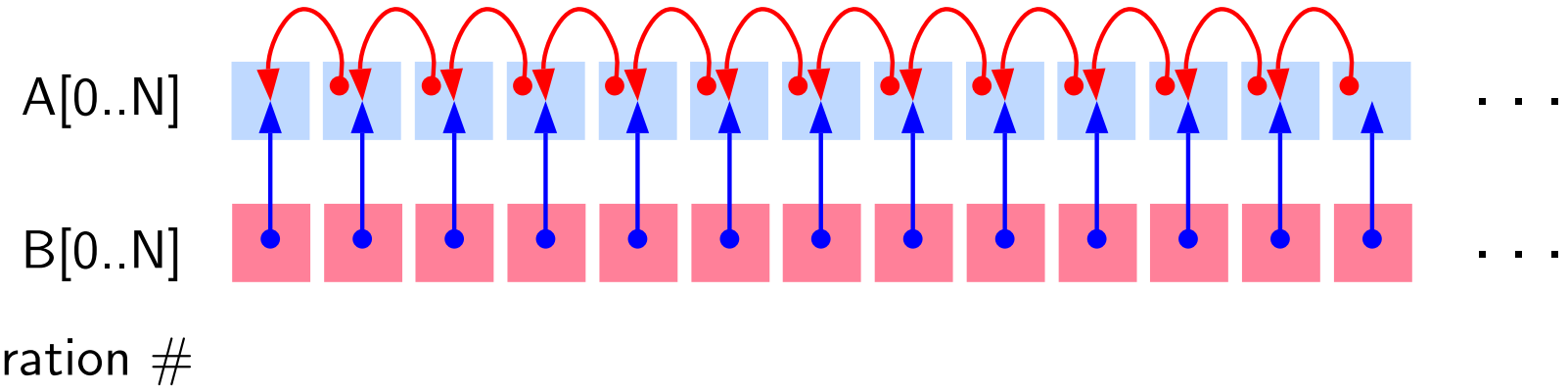
Example 2

Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction



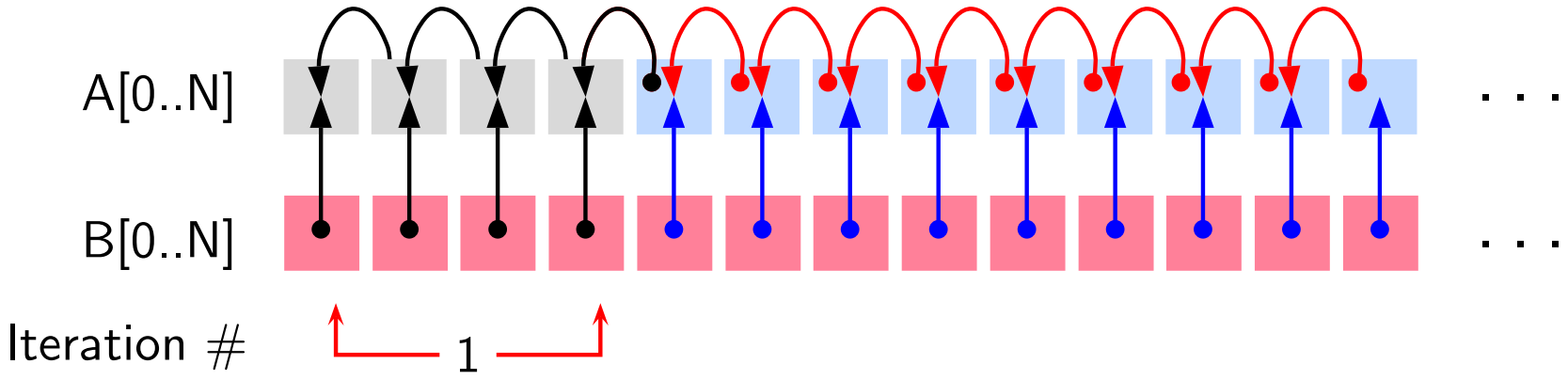
Example 2

Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction



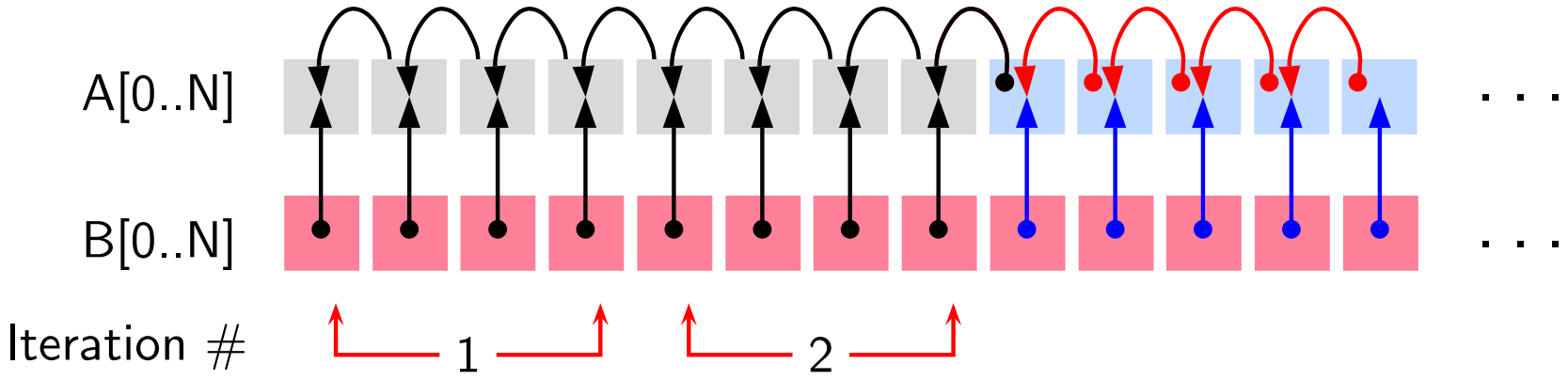
Example 2

Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction



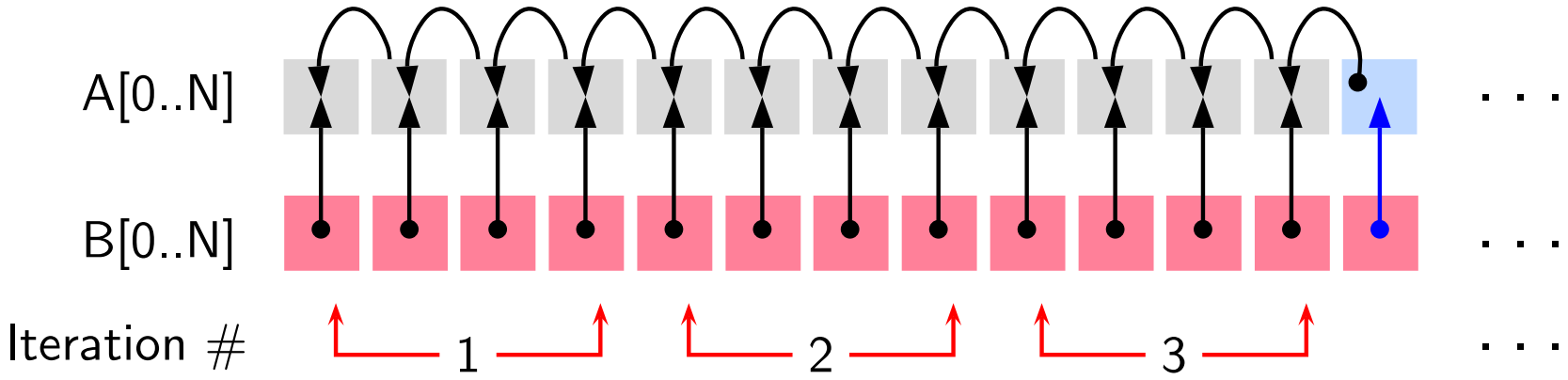
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : ??

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction



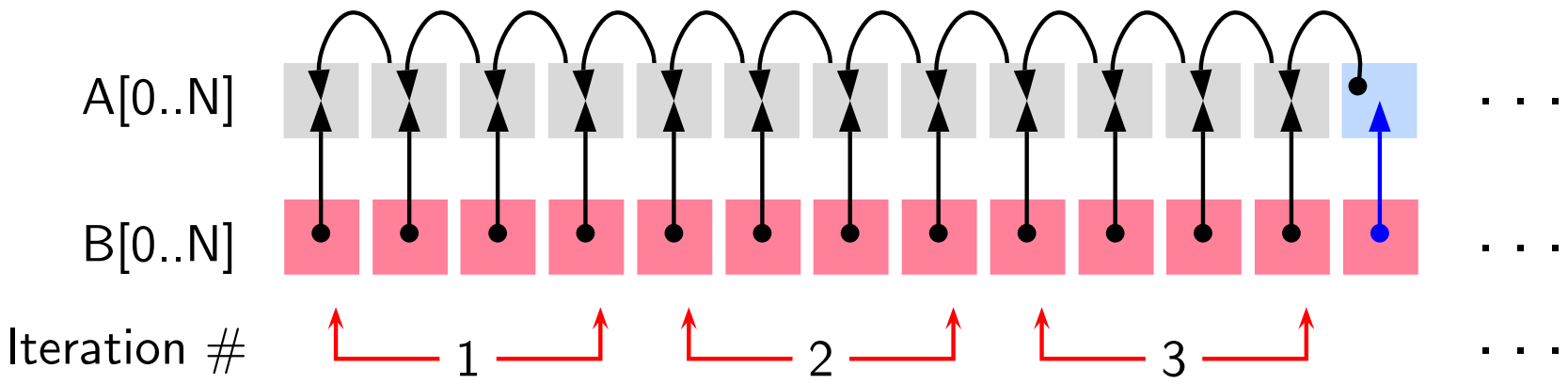
Example 2

Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction
- Read-writes across multiple instructions executing in parallel may not be synchronized



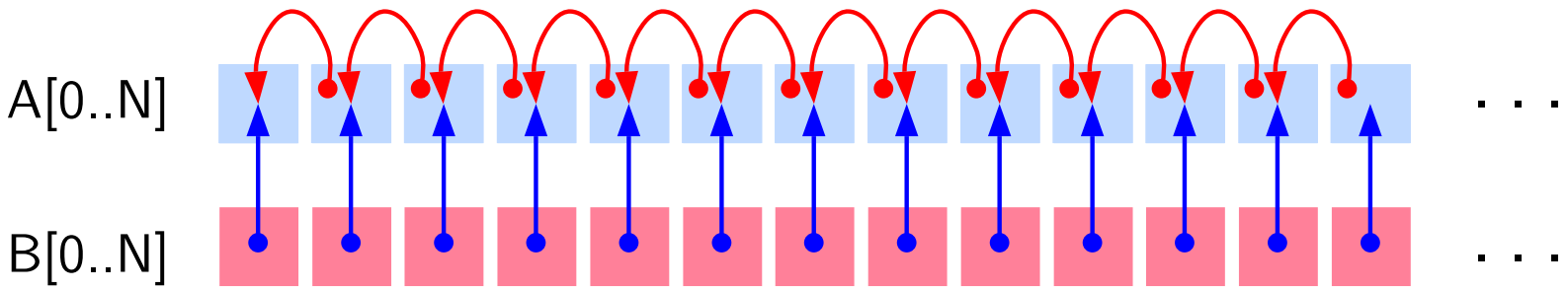
Example 2: The Moral of the Story

Vectorization	(SISD $\Rightarrow$ SIMD)	: Yes
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



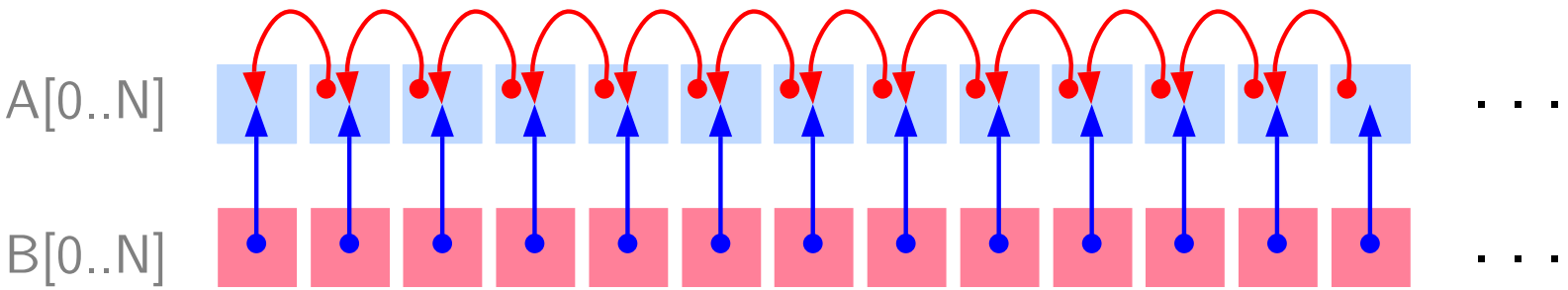
Example 2: The Moral of the Story

Vectorization	(SISD $\Rightarrow$ SIMD)	: Yes
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	
Write	Read	
Write	Write	
Read	Read	



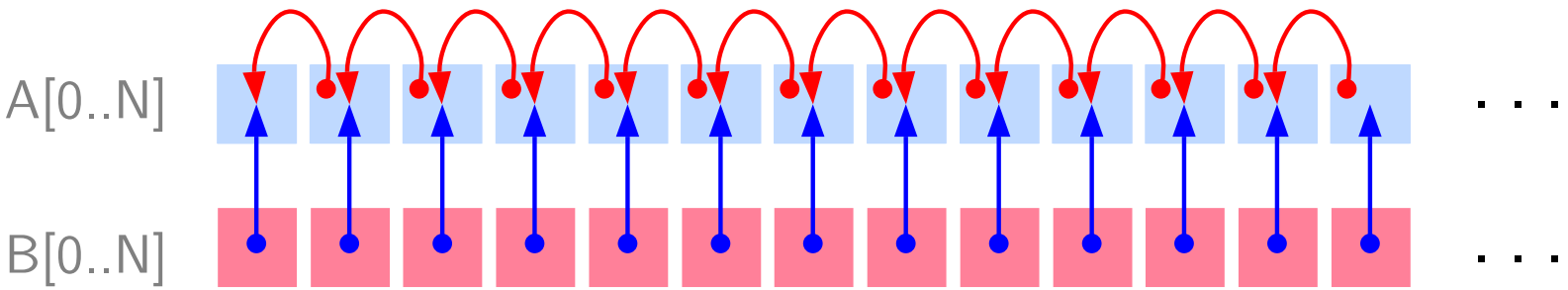
Example 2: The Moral of the Story

Vectorization	(SISD $\Rightarrow$ SIMD)	: Yes
Parallelization	(SISD $\Rightarrow$ MIMD)	: **No**

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	Yes
Write	Read	
Write	Write	
Read	Read	



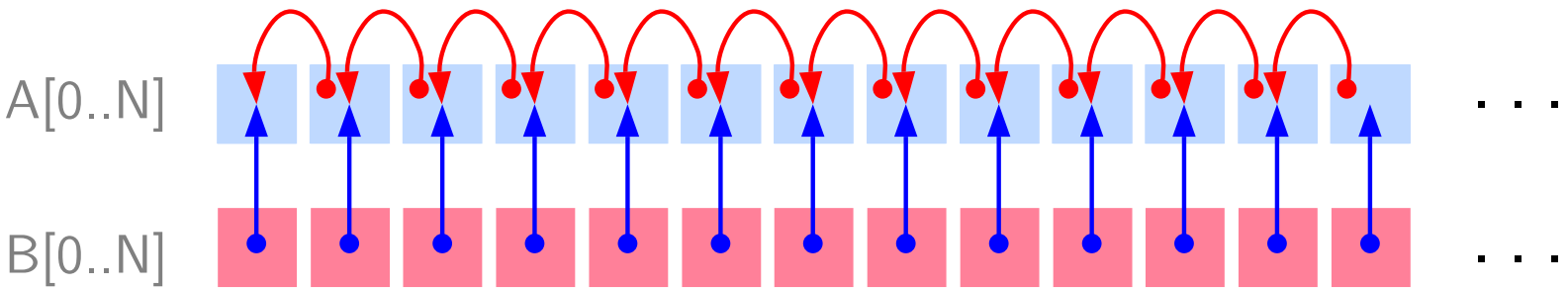
Example 2: The Moral of the Story

Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: **No**

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	Yes
Write	Read	No
Write	Write	
Read	Read	



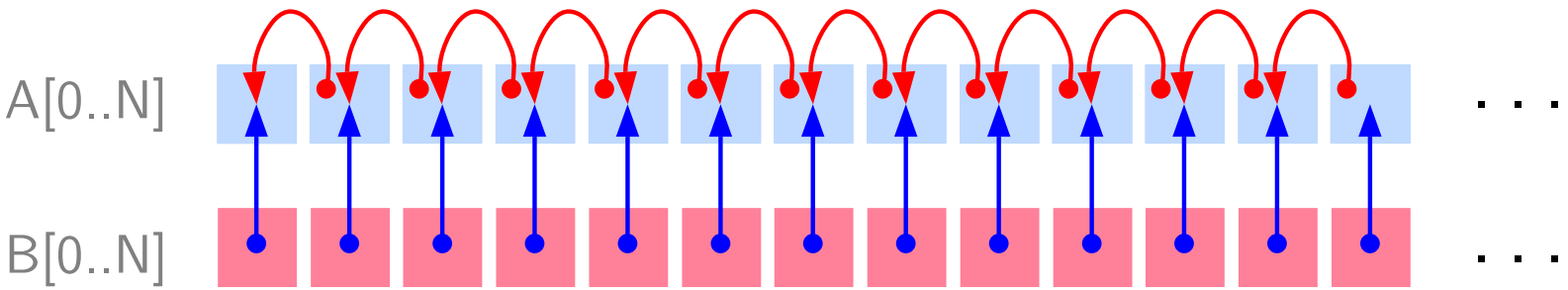
Example 2: The Moral of the Story

Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: **No**

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	Yes
Write	Read	No
Write	Write	No
Read	Read	



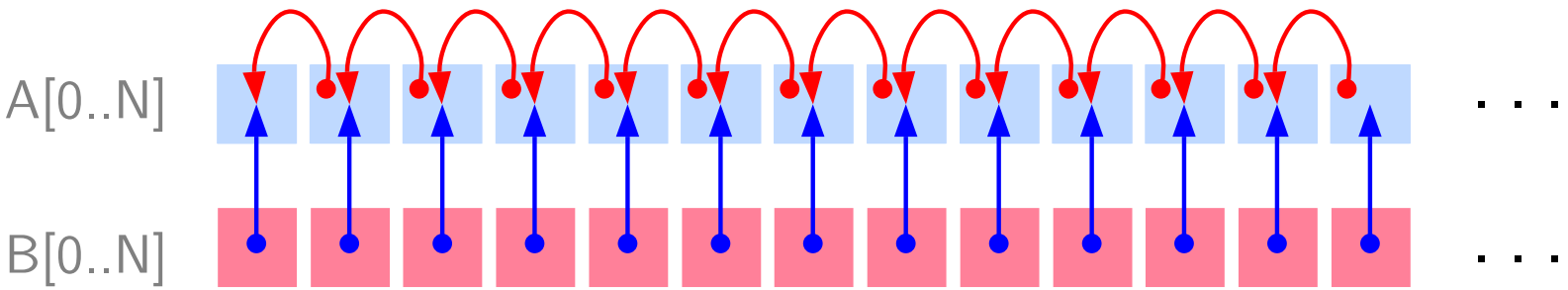
Example 2: The Moral of the Story

Vectorization	(SISD ⇒ SIMD)	: Yes
Parallelization	(SISD ⇒ MIMD)	: **No**

```
int A[
for (i
  A[i]
```

When the same location is accessed across different iterations, the order of reads and writes must be preserved

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	Yes
Write	Read	No
Write	Write	No
Read	Read	Does not matter



Example 3

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

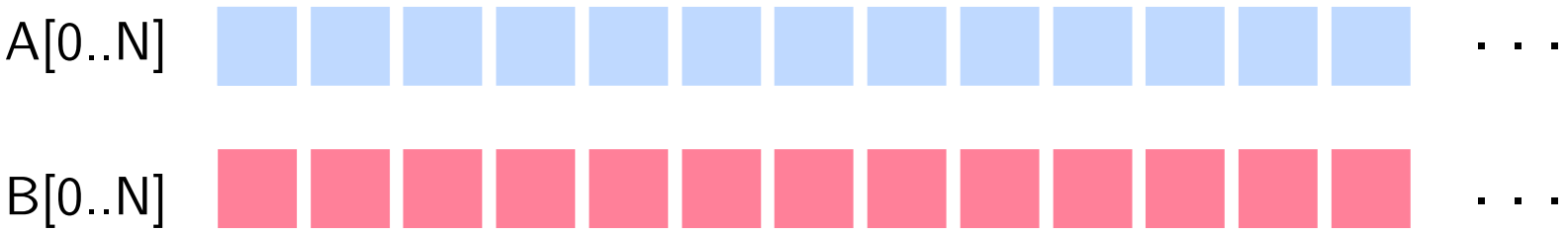
Observe reads and writes into a given location

Example 3

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

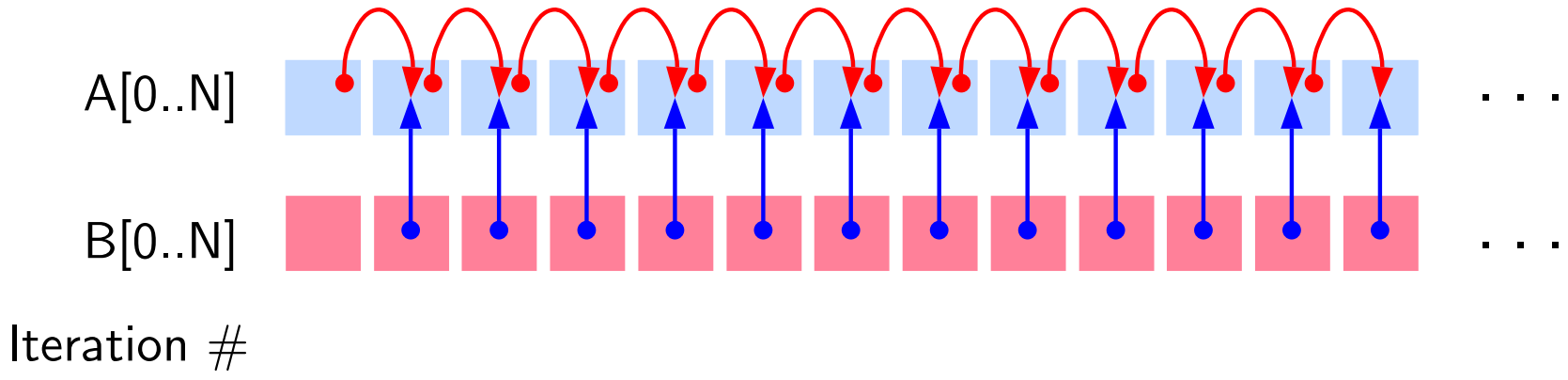


Example 3

Vectorization	(SISD ⇒ SIMD)	: No
Parallelization	(SISD ⇒ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

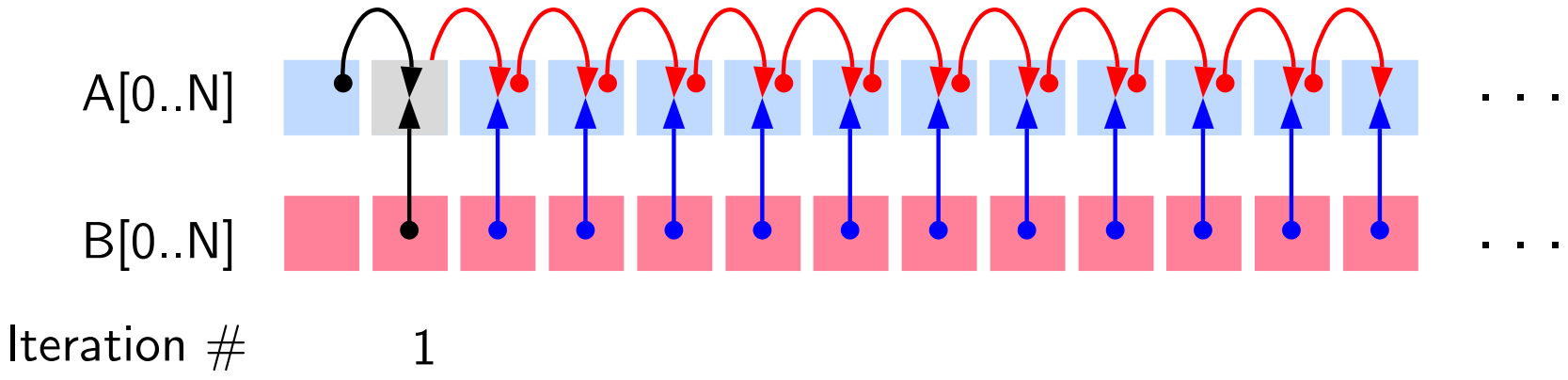


Example 3

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

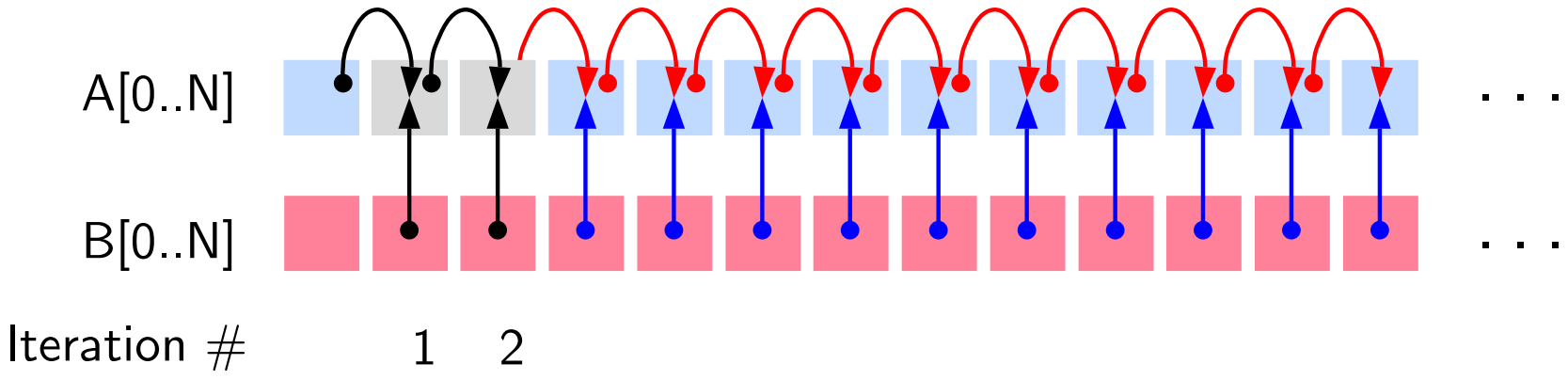


Example 3

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

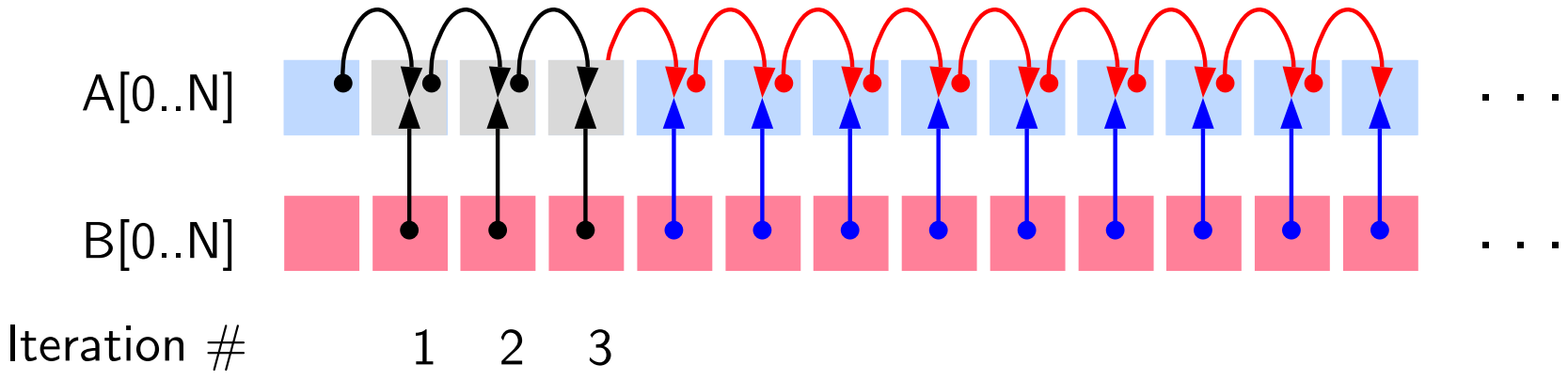


Example 3

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

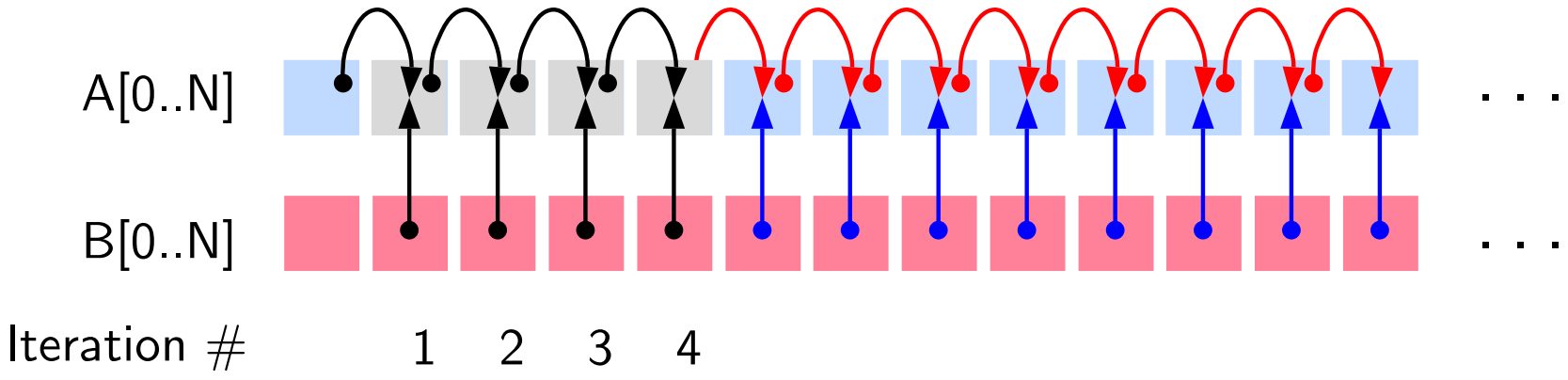


Example 3

Vectorization	(SISD ⇒ SIMD)	: No
Parallelization	(SISD ⇒ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

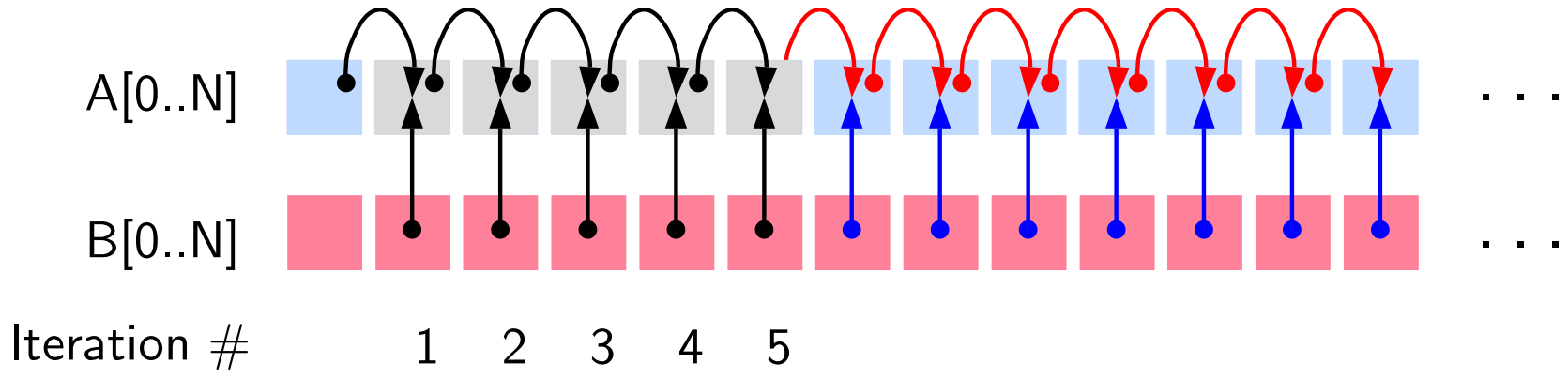


Example 3

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

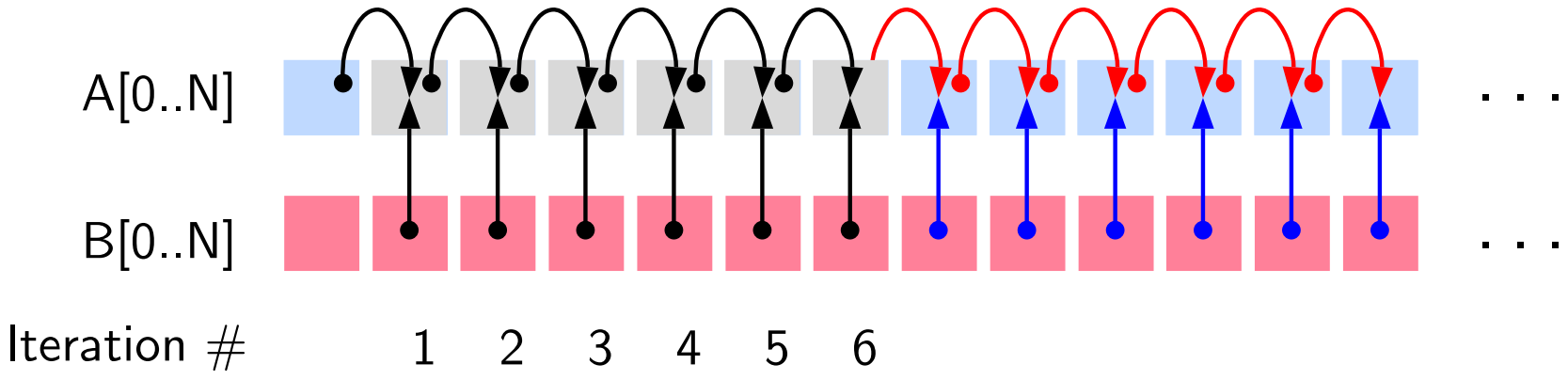


Example 3

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

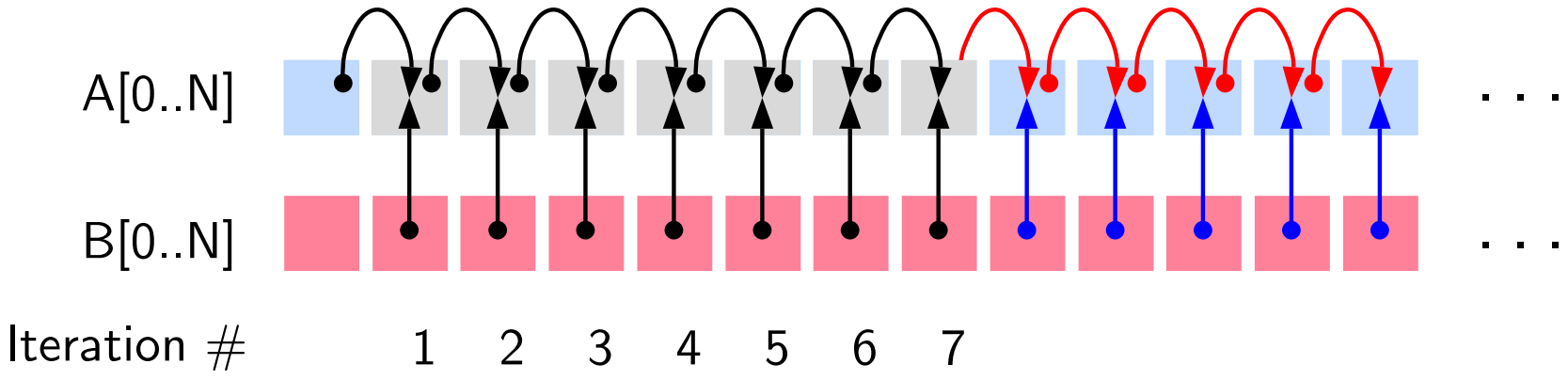


Example 3

Vectorization	(SISD ⇒ SIMD)	: No
Parallelization	(SISD ⇒ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

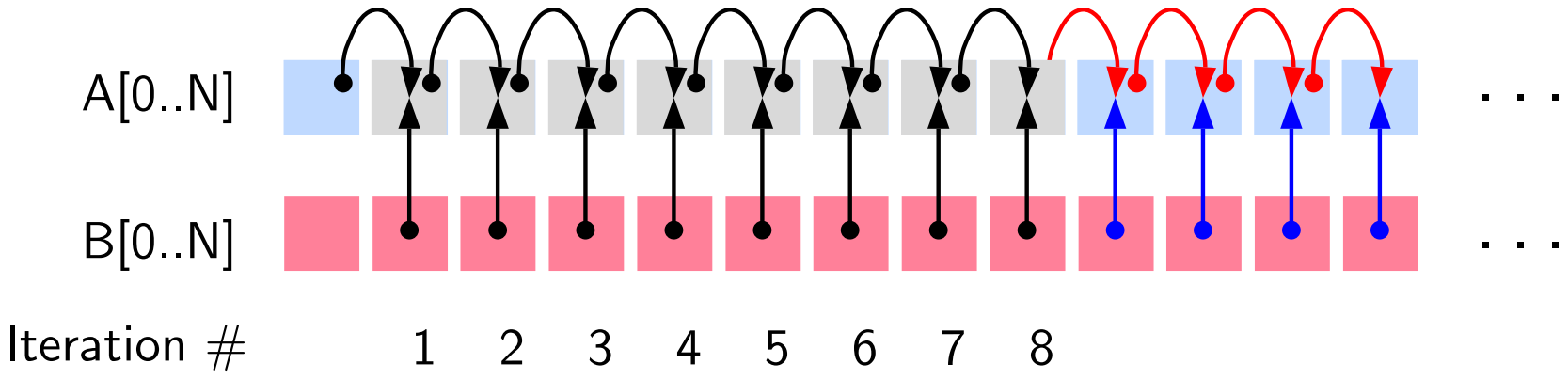


Example 3

Vectorization	(SISD ⇒ SIMD)	: No
Parallelization	(SISD ⇒ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

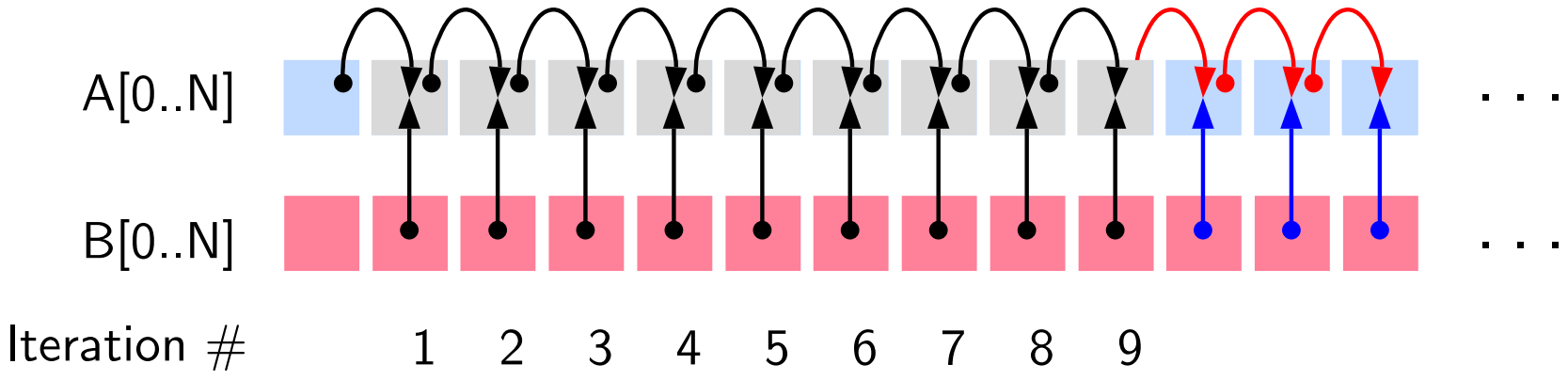


Example 3

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

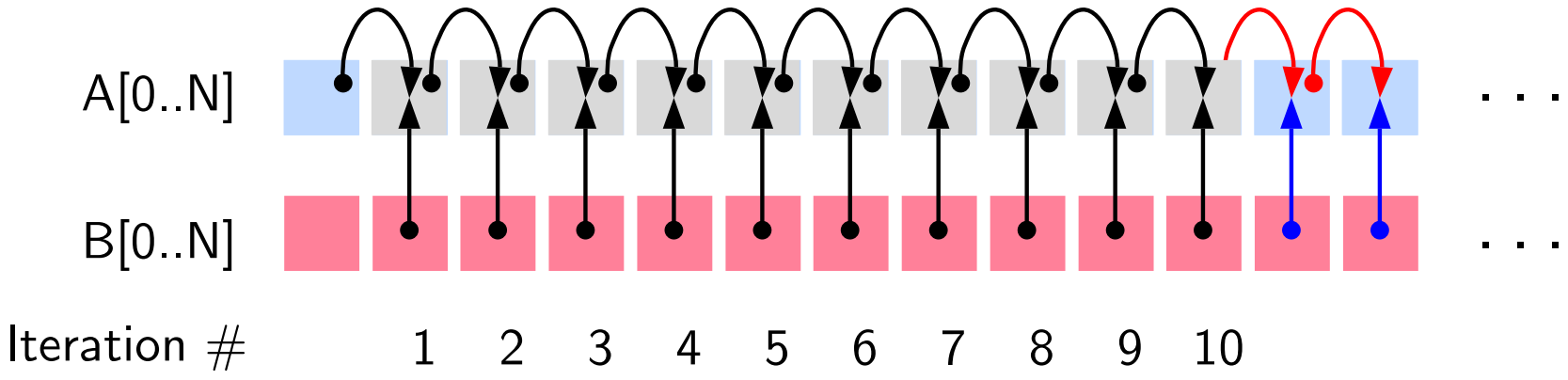


Example 3

Vectorization	(SISD ⇒ SIMD)	: No
Parallelization	(SISD ⇒ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

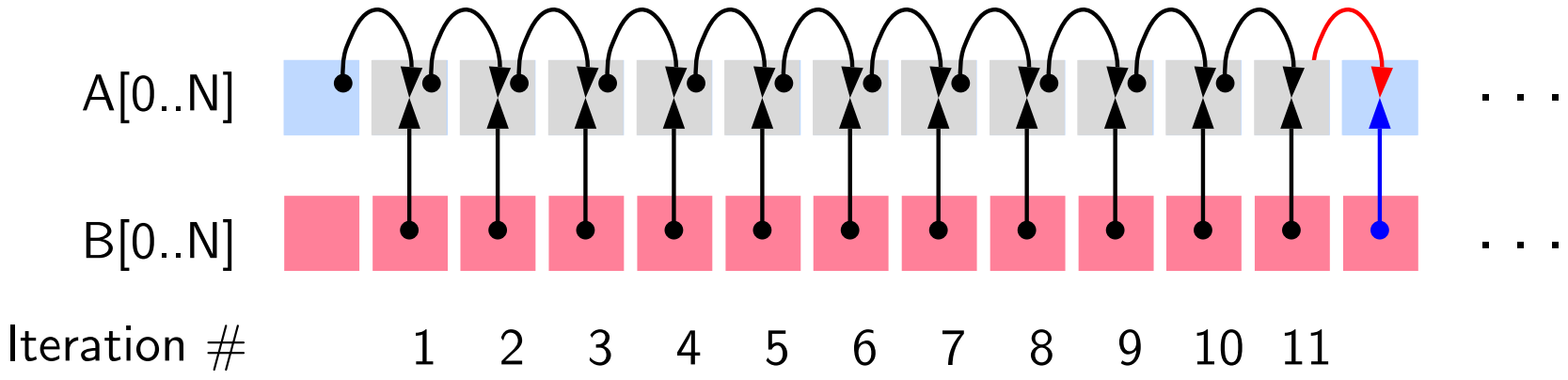


Example 3

Vectorization	(SISD ⇒ SIMD)	: No
Parallelization	(SISD ⇒ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

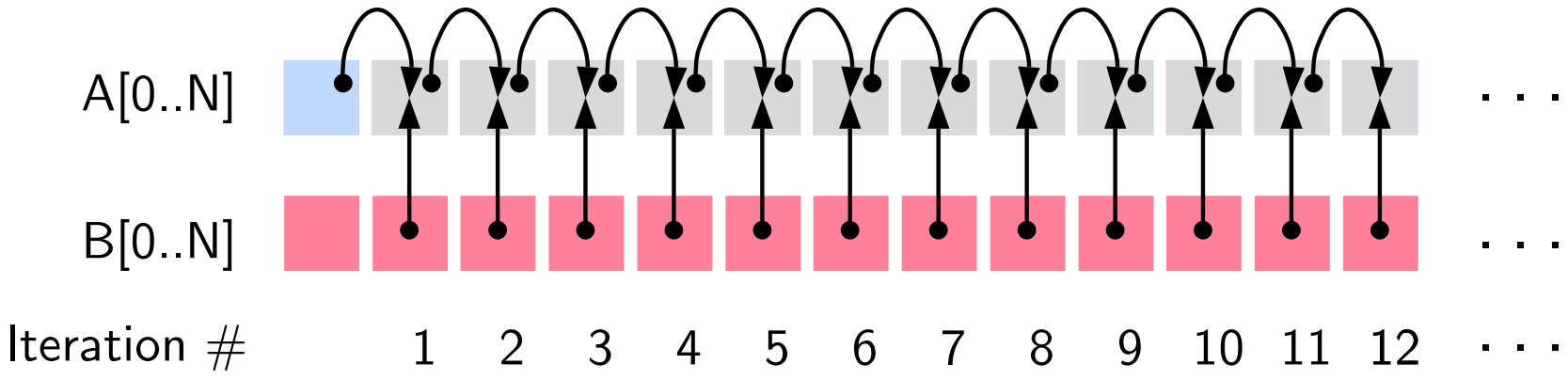


Example 3

Vectorization	(SISD ⇒ SIMD)	: No
Parallelization	(SISD ⇒ MIMD)	: No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

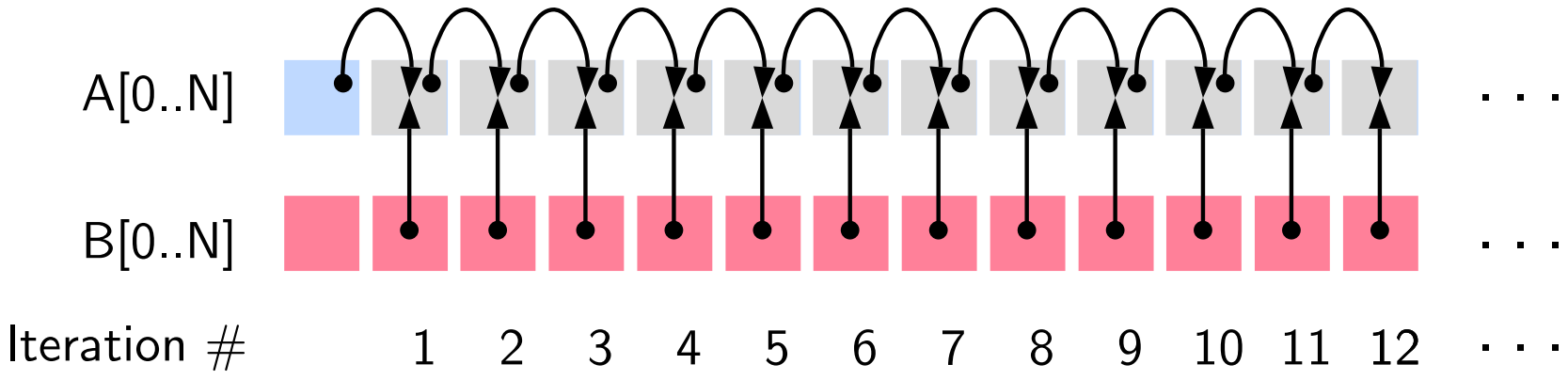


Example 3

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: No

```
int A[N], B
for (i=0; i
  A[i+1] =
```

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	No
Write	Read	
Write	Write	
Read	Read	

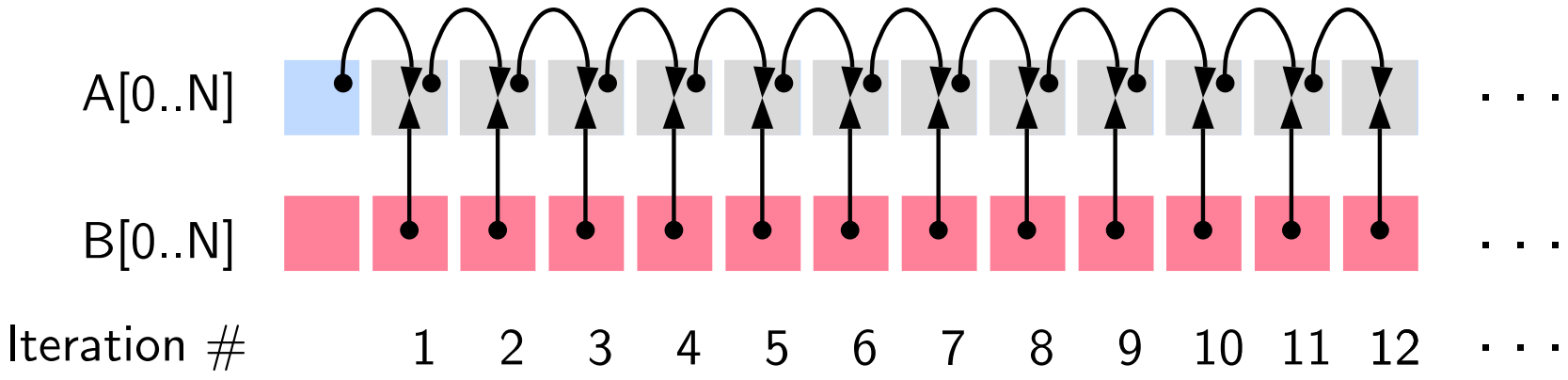


Example 3

Vectorization (SISD ⇒ SIMD) : **No**
Parallelization (SISD ⇒ MIMD) : **No**

```
int A[N], B
for (i=0; i
  A[i+1] =
```

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	No
Write	Read	Yes
Write	Write	
Read	Read	

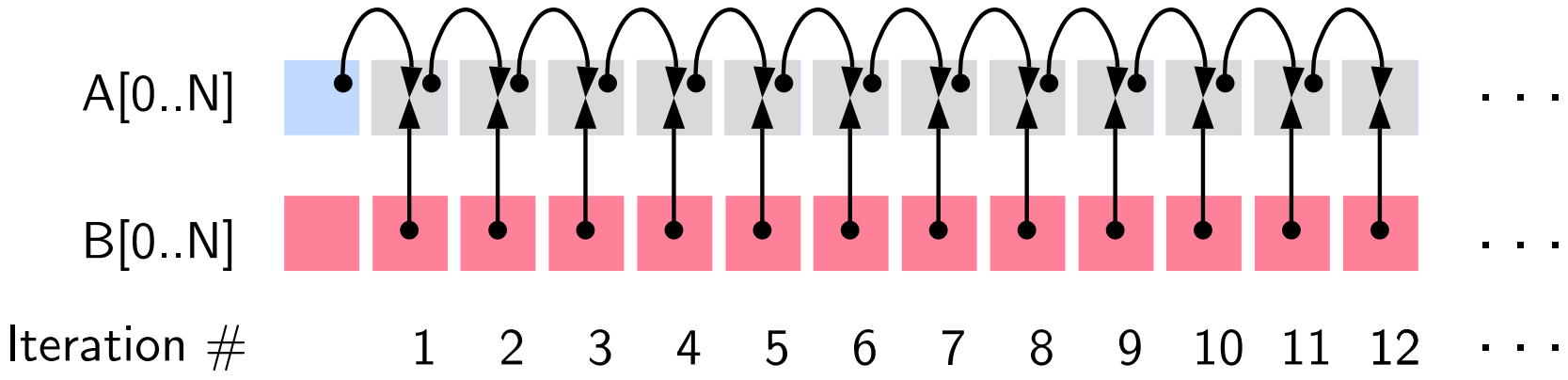


Example 3

Vectorization (SISD ⇒ SIMD) : **No**
Parallelization (SISD ⇒ MIMD) : **No**

```
int A[N], B
for (i=0; i
  A[i+1] =
```

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	No
Write	Read	Yes
Write	Write	No
Read	Read	

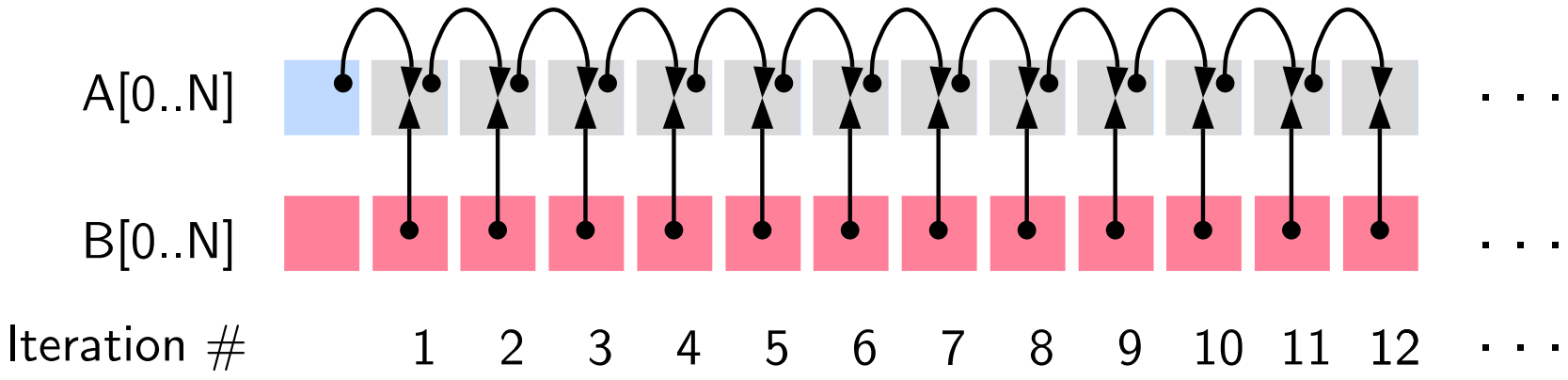


Example 3

Vectorization (SISD ⇒ SIMD) : **No**
Parallelization (SISD ⇒ MIMD) : **No**

```
int A[N], B
for (i=0; i
  A[i+1] =
```

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	No
Write	Read	Yes
Write	Write	No
Read	Read	Does not matter



Example 4

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: Yes

Example 4

Vectorization	(SISD $\Rightarrow$ SIMD)	: No
Parallelization	(SISD $\Rightarrow$ MIMD)	: Yes

- This case is not possible
- If parallelization is possible then vectorization is trivially possible
- Vectorization does not imply parallelization

Data Dependence

Let statements S_i and S_j access memory location m where execution of S_i at time t precedes execution of S_j at time $t + k$

Access in S_i	Access in S_j	Dependence	Notation
Read m	Write m	Anti (or Pseudo)	$S_i \ \bar{\delta} \ S_j$
Write m	Read m	Flow (or True)	$S_i \ \delta \ S_j$
Write m	Write m	Output (or Pseudo)	$S_i \ \delta^o \ S_j$
Read m	Read m	Does not matter	

- Pseudo dependences may be eliminated by some transformations
- True dependence cannot be eliminated

Data Dependence

Consider dependence between statements S_i and S_j in a loop

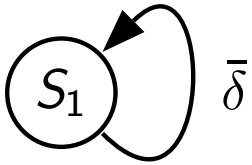
- Loop independent dependence. t and $t + k$ occur in the same iteration of a loop
  - S_i and S_j must be executed sequentially
  - Different iterations of the loop can be parallelized
- Loop carried dependence. t and $t + k$ occur in the different iterations of a loop
  - Within an iteration, S_i and S_j can be executed in parallel
  - Different iterations of the loop must be executed sequentially
- S_i and S_j may have both loop carried and loop independent dependences

Dependence in Example 1

- Program

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];   /* S1 */
```

- Dependence graph



- No loop carried dependence

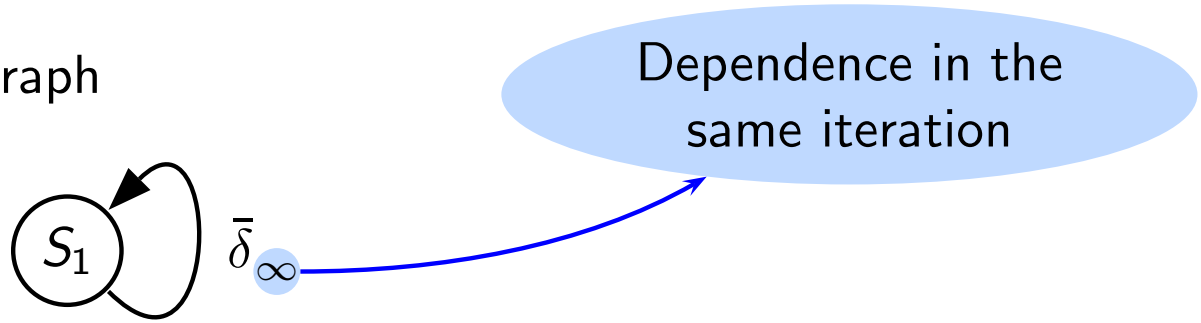
  Both vectorization and parallelization are possible

Dependence in Example 1

- Program

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];   /* S1 */
```

- Dependence graph



- No loop carried dependence

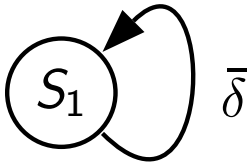
  Both vectorization and parallelization are possible

Dependence in Example 2

- Program

```
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i] = A[i+1] + B[i];  /* S1 */
```

- Dependence graph



- Loop carried anti-dependence

  Parallelization is not possible

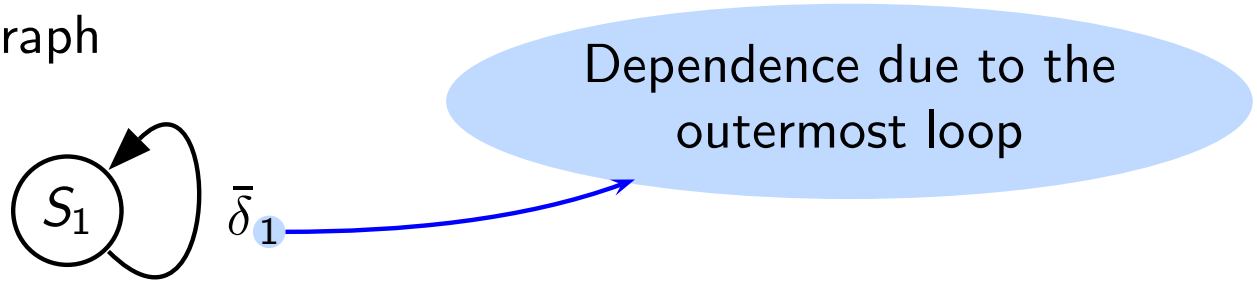
  Vectorization is possible since all reads are done before all writes

Dependence in Example 2

- Program

```
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i] = A[i+1] + B[i];  /* S1 */
```

- Dependence graph



- Loop carried anti-dependence

  Parallelization is not possible

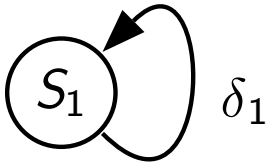
  Vectorization is possible since all reads are done before all writes

Dependence in Example 3

- Program

```
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];  /* S1 */
```

- Dependence graph



- Loop carried flow-dependence

  Neither parallelization not vectorization is possible

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: Loop carried dependence exists

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: Loop carried dependence exists

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: Loop carried dependence exists

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 2

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i][j] = a[i][j] + 2;
  }
```

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	0, 0	0, 0
0, 1	0, 1	0, 1
0, 2	0, 2	0, 2
0, 3	0, 3	0, 3
1, 0	1, 0	1, 0
1, 1	1, 1	1, 1
1, 2	1, 2	1, 2
1, 3	1, 3	1, 3
2, 0	2, 0	2, 0
2, 1	2, 1	2, 1
2, 2	2, 2	2, 2
2, 3	2, 3	2, 3
3, 0	3, 0	3, 0
3, 1	3, 1	3, 1
3, 2	3, 2	3, 2
3, 3	3, 3	3, 3

Iteration Vectors and Index Vectors: Example 2

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	0, 0	0, 0
0, 1	0, 1	0, 1
0, 2	0, 2	0, 2
0, 3	0, 3	0, 3
1, 0	1, 0	1, 0
1, 1	1, 1	1, 1
1, 2	1, 2	1, 2
1, 3	1, 3	1, 3
2, 0	2, 0	2, 0
2, 1	2, 1	2, 1
2, 2	2, 2	2, 2
2, 3	2, 3	2, 3
3, 0	3, 0	3, 0
3, 1	3, 1	3, 1
3, 2	3, 2	3, 2
3, 3	3, 3	3, 3

Iteration Vectors and Index Vectors: Example 2

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

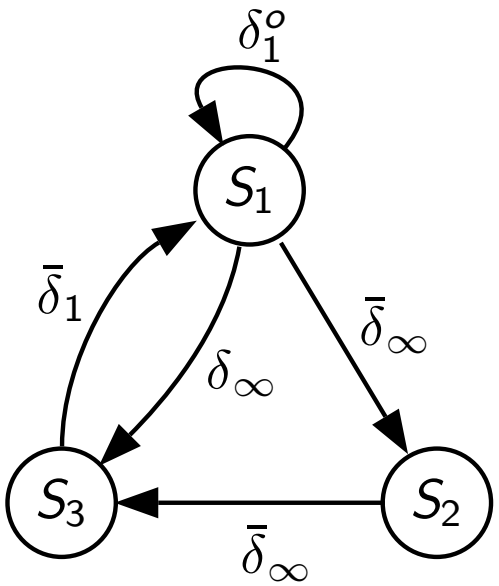
Conclusion: No loop carried dependence

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	0, 0	0, 0
0, 1	0, 1	0, 1
0, 2	0, 2	0, 2
0, 3	0, 3	0, 3
1, 0	1, 0	1, 0
1, 1	1, 1	1, 1
1, 2	1, 2	1, 2
1, 3	1, 3	1, 3
2, 0	2, 0	2, 0
2, 1	2, 1	2, 1
2, 2	2, 2	2, 2
2, 3	2, 3	2, 3
3, 0	3, 0	3, 0
3, 1	3, 1	3, 1
3, 2	3, 2	3, 2
3, 3	3, 3	3, 3

Dependence in Example 4

Program to swap arrays	Dependence Graph

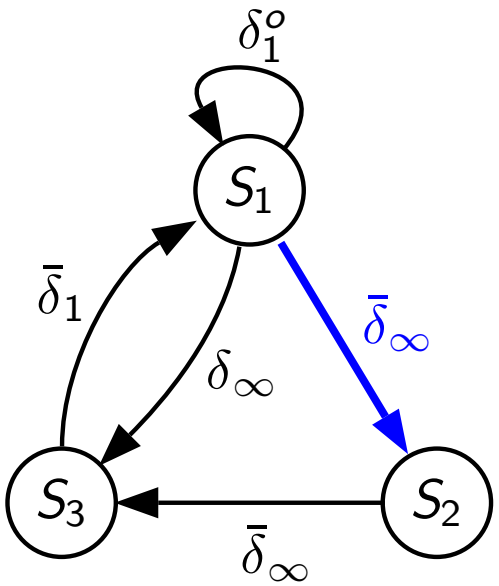
```
for (i=0; i<N; i++)
{
    T = A[i];          /* S1 */
    A[i] = B[i];       /* S2 */
    B[i] = T;          /* S3 */
}
```



Dependence in Example 4

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];          /* S1 */
    A[i] = B[i];       /* S2 */
    B[i] = T;          /* S3 */
}
```

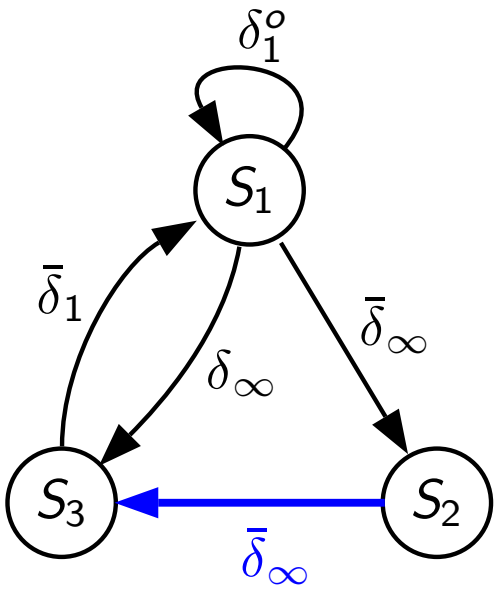


Loop independent anti dependence due to `A[i]`

Dependence in Example 4

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];          /* S1 */
    A[i] = B[i];       /* S2 */
    B[i] = T;          /* S3 */
}
```

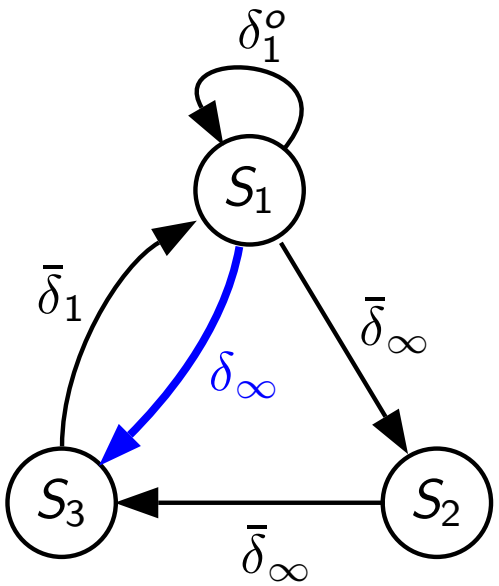


Loop independent anti dependence due to `B[i]`

Dependence in Example 4

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];          /* S1 */
    A[i] = B[i];       /* S2 */
    B[i] = T;          /* S3 */
}
```

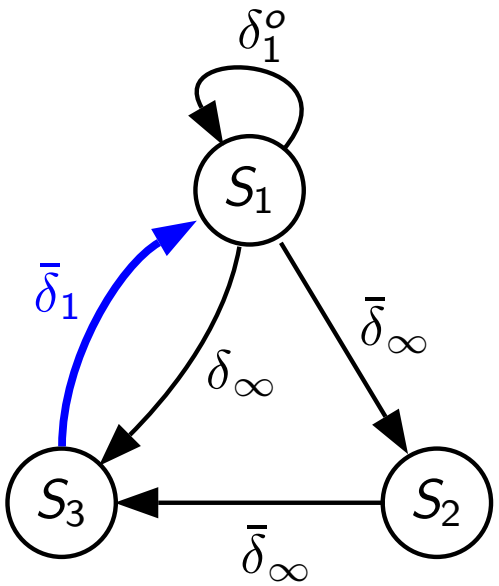


Loop independent flow dependence due to T

Dependence in Example 4

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];          /* S1 */
    A[i] = B[i];       /* S2 */
    B[i] = T;          /* S3 */
}
```

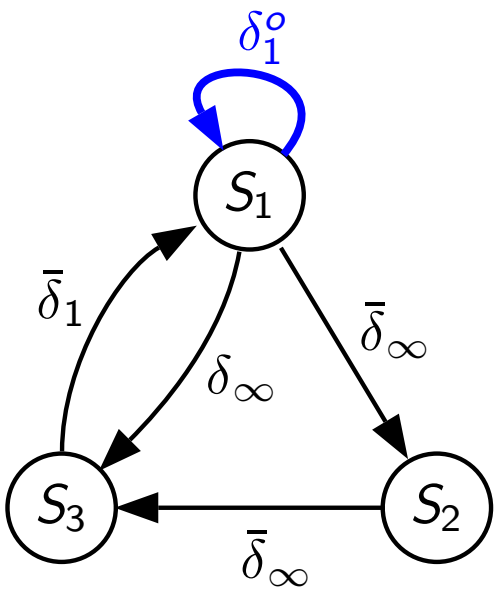


Loop carried anti dependence due to T

Dependence in Example 4

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];          /* S1 */
    A[i] = B[i];       /* S2 */
    B[i] = T;          /* S3 */
}
```

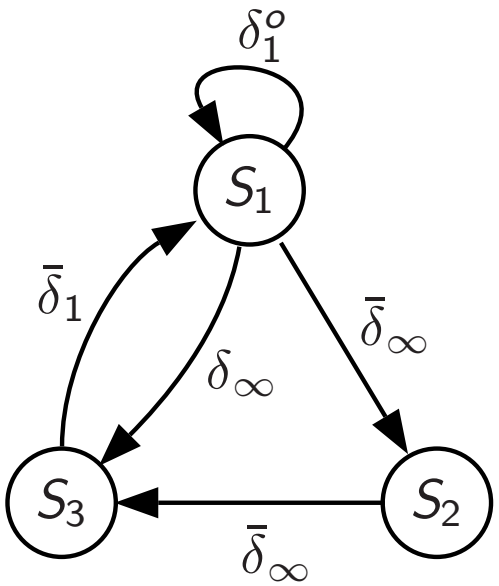


Loop carried output dependence due to T

Dependence in Example 4

Program to swap arrays	Dependence Graph

```
for (i=0; i<N; i++)
{
    T = A[i];          /* S1 */
    A[i] = B[i];       /* S2 */
    B[i] = T;          /* S3 */
}
```



Data Dependence Theorem

There exists a dependence from statement S_1 to statement S_2 in common nest of loops if and only if there exist two iteration vectors **i** and **j** for the nest, such that

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

Anti Dependence and Vectorization

Read precedes Write lexicographically

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: C[i] = A[i+2];
    S2: A[i] = B[i];
}
```

Anti Dependence and Vectorization

Read precedes Write lexicographically

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: C[i] = A[i+2];
    S2: A[i] = B[i];
}
```

→

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    S1: C[i:i+3] = A[i+2:i+5];
    S2: A[i:i+3] = B[i:i+3];
}
```

Anti Dependence and Vectorization

Write precedes Read lexicographically

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: C[i] = A[i+2];
}
```

Anti Dependence and Vectorization

Write precedes Read lexicographically

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: C[i] = A[i+2];
}
```

→

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S2: C[i] = A[i+2];
    S1: A[i] = B[i];
}
```

Anti Dependence and Vectorization

Write precedes Read lexicographically

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: C[i] = A[i+2];
}
```

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S2: C[i] = A[i+2];
    S1: A[i] = B[i];
}
```

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    S2: C[i:i+3] = A[i+2:i+5];
    S1: A[i:i+3] = B[i:i+3];
}
```

True Dependence and Vectorization

Write precedes Read lexicographically

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i+2] = C[i];
    S2: B[i] = A[i];
}
```

True Dependence and Vectorization

Write precedes Read lexicographically

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    S1: A[i+2] = C[i];
    S2: B[i] = A[i];
}
```

→

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    S1: A[i+2:i+5] = C[i:i+3];
    S2: B[i:i+3] = A[i:i+3];
}
```

Multiple Dependences and Vectorization

True Dependence and Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: B[i+2] = A[i+1];
}
```

Multiple Dependences and Vectorization

True Dependence and Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: B[i+2] = A[i+1];
}
```

→

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S2: B[i+2] = A[i+1];
    S1: A[i] = B[i];
}
```

Multiple Dependences and Vectorization

True Dependence and Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: B[i+2] = A[i+1];
}
```

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S2: B[i+2] = A[i+1];
    S1: A[i] = B[i];
}
```

```
int A[N], B[N], i;
for (i=0; i<N; i=i+4) {
    S2: B[i+2:i+5] = A[i+1:i+4];
    S1: A[i:i+3] = B[i:i+3];
}
```

Observation: Feasibility of Vectorization

- For the given loop, consider the corresponding Data Dependence Graph (DDG)
- If the DDG does not have a cycle, vectorization is possible
- If the DDG has a cycle, vectorization may or may not be possible depending on the nature of cycle

Cyclic Dependences and Vectorization

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+1] = B[i];
}
```

Cyclic Dependences and Vectorization

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+1] = B[i];
}
```

Cyclic Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+1];
    S2: A[i] = B[i+2];
}
```

Cyclic Dependences and Vectorization

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+1] = B[i];
}
```

Cyclic Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+1];
    S2: A[i] = B[i+2];
}
```

- Rescheduling of statements will not break the cyclic dependence
- Both the dependence distances are $<$ VF

Cyclic Dependences and Vectorization

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+1] = B[i];
}
```

Cyclic Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+1];
    S2: A[i] = B[i+2];
}
```

- Rescheduling of statements will not break the cyclic dependence
- Both the dependence distances are $<$ VF

Cannot Vectorize

Cyclic Dependences and Vectorization

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+5] = B[i];
}
```

Cyclic Dependences and Vectorization

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+5] = B[i];
}
```

Cyclic Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+2];
    S2: A[i] = B[i+5];
}
```

Cyclic Dependences and Vectorization

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+5] = B[i];
}
```

Cyclic Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+2];
    S2: A[i] = B[i+5];
}
```

- Rescheduling of statements will not break the cyclic dependence
- One of the dependence distances is $\geq$ VF

Cyclic Dependences and Vectorization

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+5] = B[i];
}
```

Cyclic Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+2];
    S2: A[i] = B[i+5];
}
```

- Rescheduling of statements will not break the cyclic dependence
- One of the dependence distances is $\geq$ VF

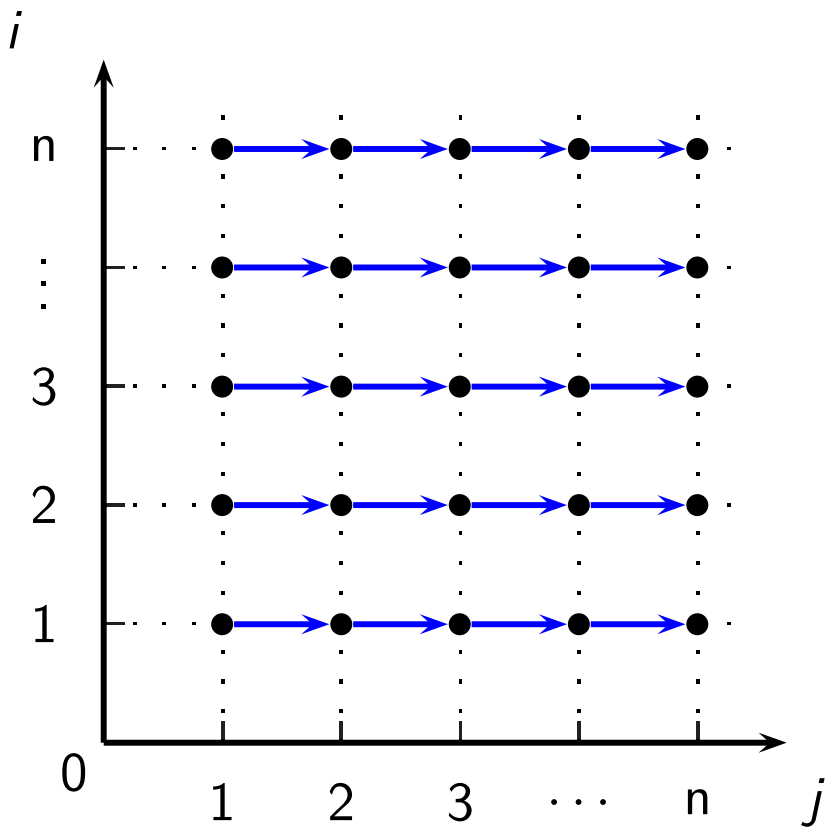
Can Vectorize

- Is vectorization possible if 2 and 5 are swapped in the examples above?

Feasibility of Parallelization

Outer Parallel

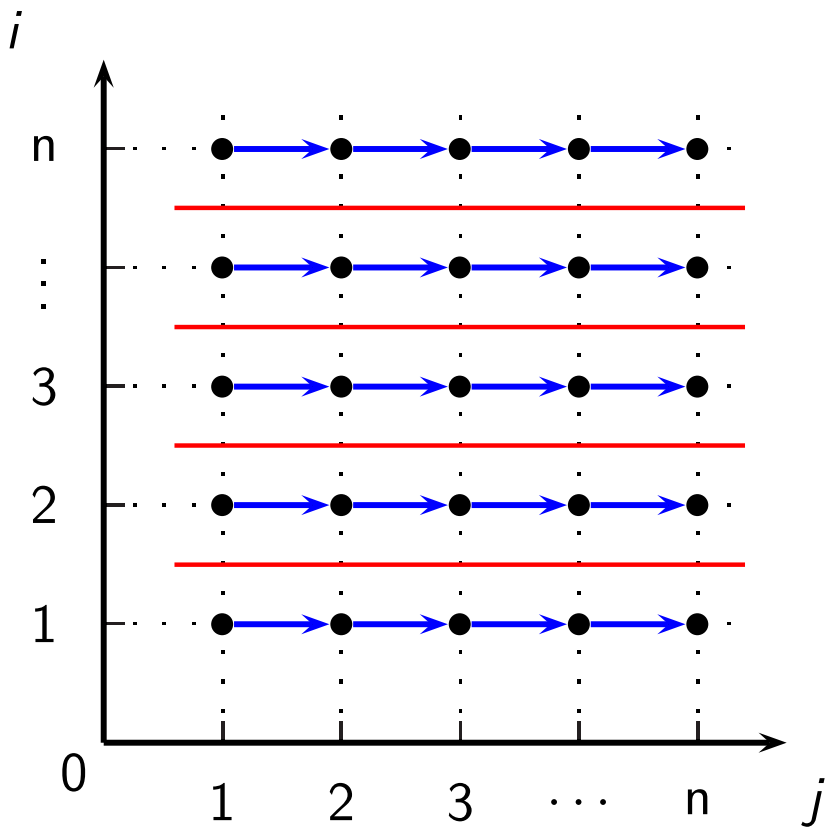
```
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      A[i][j] = A[i][j+1];
```



Feasibility of Parallelization

Outer Parallel

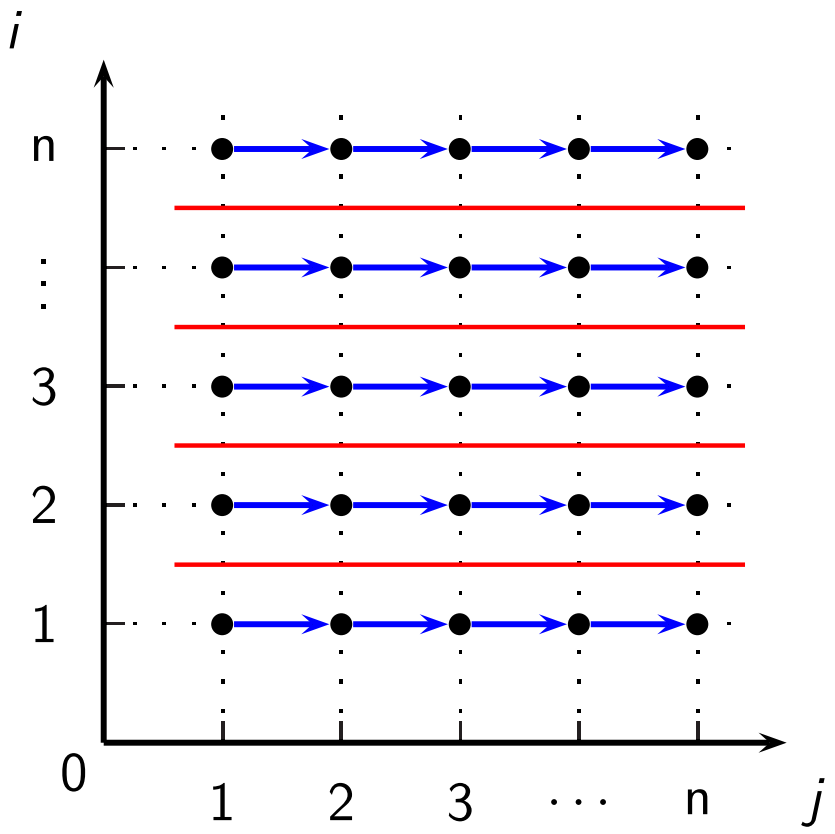
```
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      A[i][j] = A[i][j+1];
```



Feasibility of Parallelization

Outer Parallel

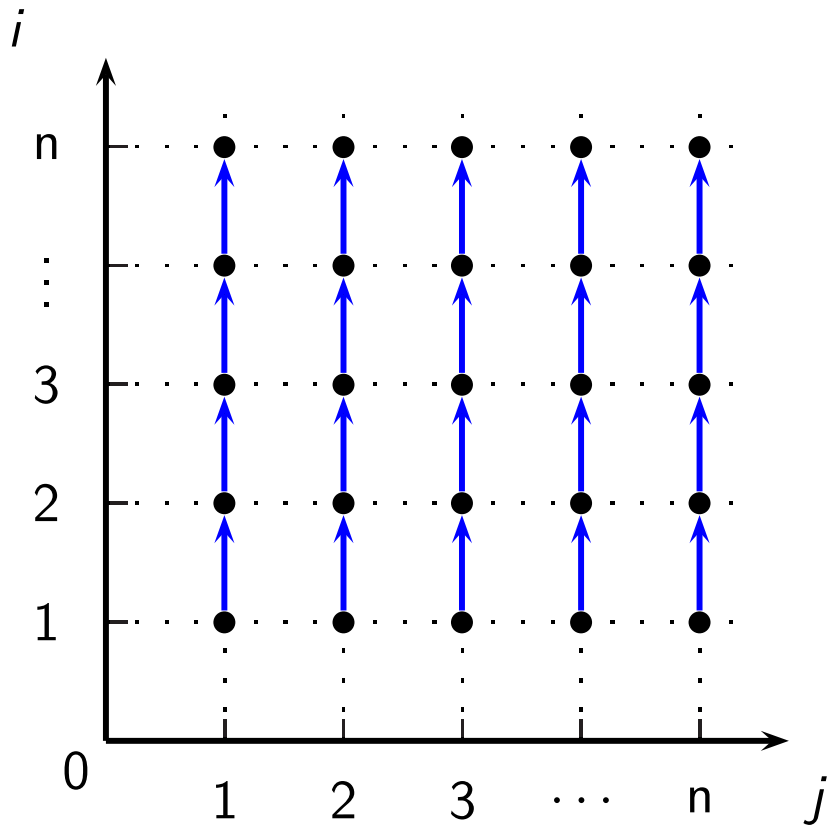
```
for-all (i=1 to n)
   for (j=1; j<n; j++)
      A[i][j] = A[i][j+1];
Synchronize
```



Feasibility of Parallelization

Inner Parallel

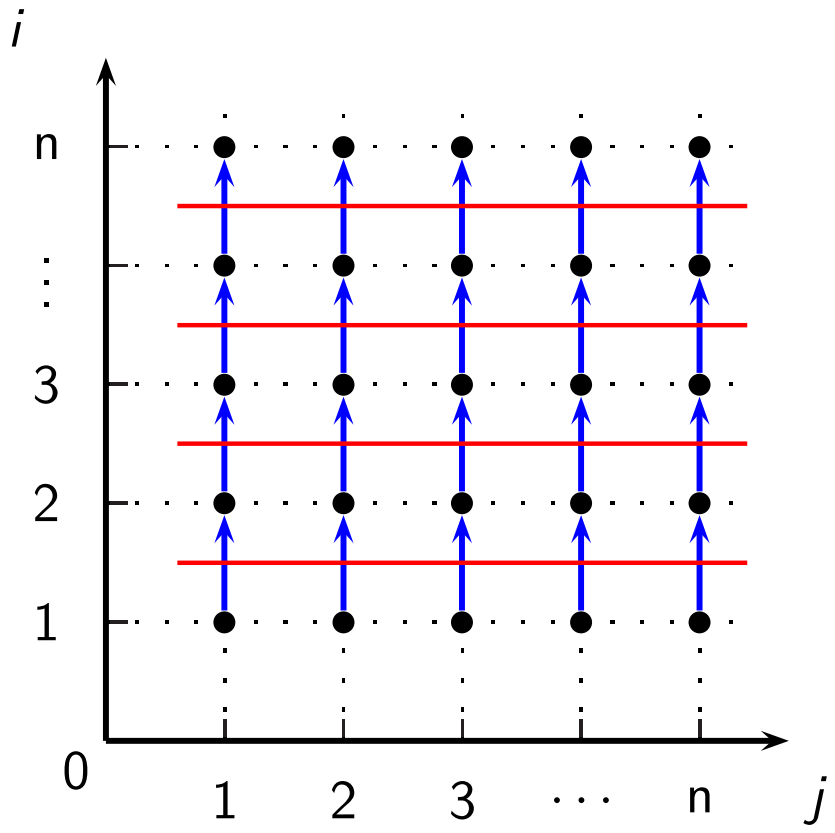
```
for (i=2; i<n; i++)
   for (j=1; j<n; j++)
      A[i][j] = A[i-1][j];
```



Feasibility of Parallelization

Inner Parallel

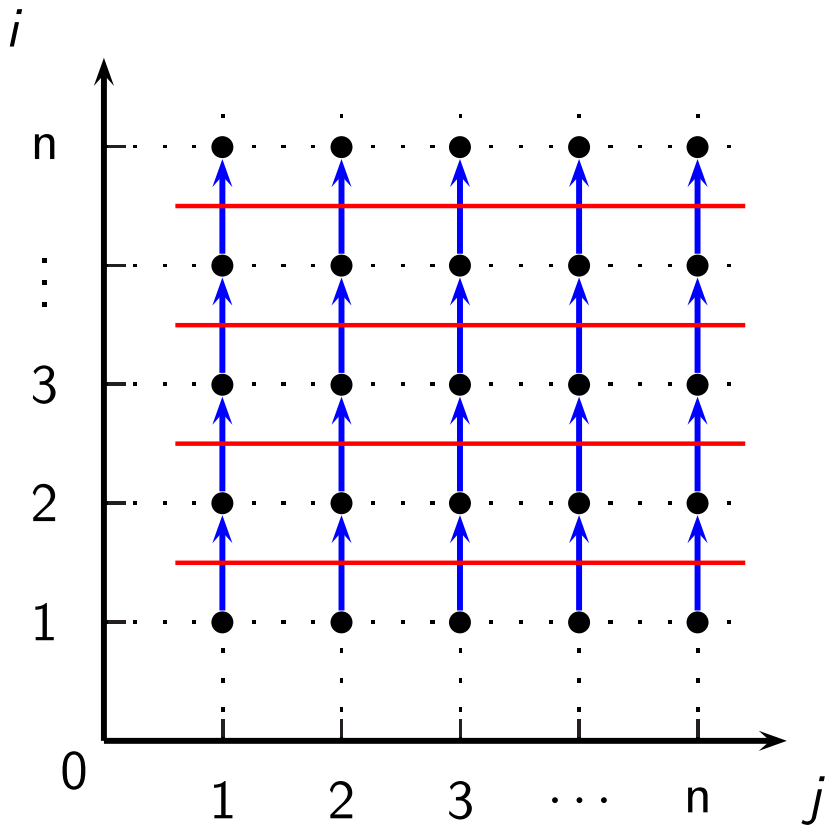
```
for (i=2; i<n; i++)
   for (j=1; j<n; j++)
      A[i][j] = A[i-1][j];
```



Feasibility of Parallelization

Inner Parallel

```
for (i=2; i<n; i++)
   for-all (j=1 to n)
      A[i][j] = A[i-1][j];
   Synchronize
```



Part 2

The GCC-4.7.2 Framework

GCC-4.7.2 Framework for Loop Transforms

- Getting loop information (Loop discovery)
- Finding value spaces of induction variables, array subscript functions, and pointer accesses
- Analyzing data dependence
- Performing loop transformations

Loop Transformation Passes in GCC-4.7.2

```
NEXT_PASS (pass_tree_loop);
  {
     struct opt_pass **p = &pass_tree_loop.pass.sub;
     NEXT_PASS (pass_tree_loop_init);
     NEXT_PASS (pass_lim);
     ...
     NEXT_PASS (pass_check_data_deps);
     NEXT_PASS (pass_loop_distribution);
     NEXT_PASS (pass_copy_prop);
     NEXT_PASS (pass_graphite);
       {
          struct opt_pass **p = &pass_graphite.pass.sub;
          NEXT_PASS (pass_graphite_transforms);
          ...
       }
     NEXT_PASS (pass_iv_canon);
     NEXT_PASS (pass_if_conversion);
     NEXT_PASS (pass_vectorize);
       {
          struct opt_pass **p = &pass_vectorize.pass.sub;
          NEXT_PASS (pass_lower_vector_ssa);
          NEXT_PASS (pass_dce_loop);
       }
     NEXT_PASS (pass_predcom);
     NEXT_PASS (pass_complete_unroll);
     NEXT_PASS (pass_slp_vectorize);
     NEXT_PASS (pass_parallelize_loops);
     NEXT_PASS (pass_loop_prefetch);
     NEXT_PASS (pass_iv_optimize);
     NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form

  A variant of Gimple IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features

  (Vectorization factor, alignment etc.)

Loop Transformation Passes in GCC-4.7.2

```
NEXT_PASS (pass_tree_loop);
  {
     struct opt_pass **p = &pass_tree_loop.pass.sub;
     NEXT_PASS (pass_tree_loop_init);
     NEXT_PASS (pass_lim);
     ...
     NEXT_PASS (pass_check_data_deps);
     NEXT_PASS (pass_loop_distribution);
     NEXT_PASS (pass_copy_prop);
     NEXT_PASS (pass_graphite);
       {
          struct opt_pass **p = &pass_graphite.pass.sub;
          NEXT_PASS (pass_graphite_transforms);
          ...
       }
     NEXT_PASS (pass_iv_canon);
     NEXT_PASS (pass_if_conversion);
     NEXT_PASS (pass_vectorize);
       {
          struct opt_pass **p = &pass_vectorize.pass.sub;
          NEXT_PASS (pass_lower_vector_ssa);
          NEXT_PASS (pass_dce_loop);
       }
     NEXT_PASS (pass_predcom);
     NEXT_PASS (pass_complete_unroll);
     NEXT_PASS (pass_slp_vectorize);
     NEXT_PASS (pass_parallelize_loops);
     NEXT_PASS (pass_loop_prefetch);
     NEXT_PASS (pass_iv_optimize);
     NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form

  A variant of Gimple IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features

  (Vectorization factor, alignment etc.)

Loop Transformation Passes in GCC-4.7.2

```
NEXT_PASS (pass_tree_loop);
  {
     struct opt_pass **p = &pass_tree_loop.pass.sub;
     NEXT_PASS (pass_tree_loop_init);
     NEXT_PASS (pass_lim);
     ...
     NEXT_PASS (pass_check_data_deps);
     NEXT_PASS (pass_loop_distribution);
     NEXT_PASS (pass_copy_prop);
     NEXT_PASS (pass_graphite);
       {
          struct opt_pass **p = &pass_graphite.pass.sub;
          NEXT_PASS (pass_graphite_transforms);
          ...
       }
     NEXT_PASS (pass_iv_canon);
     NEXT_PASS (pass_if_conversion);
     NEXT_PASS (pass_vectorize);
       {
          struct opt_pass **p = &pass_vectorize.pass.sub;
          NEXT_PASS (pass_lower_vector_ssa);
          NEXT_PASS (pass_dce_loop);
       }
     NEXT_PASS (pass_predcom);
     NEXT_PASS (pass_complete_unroll);
     NEXT_PASS (pass_slp_vectorize);
     NEXT_PASS (pass_parallelize_loops);
     NEXT_PASS (pass_loop_prefetch);
     NEXT_PASS (pass_iv_optimize);
     NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form

  A variant of Gimple IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features

  (Vectorization factor, alignment etc.)

Loop Transformation Passes in GCC: Our Focus

Data Dependence	Pass variable name	`pass_check_data_deps`
	Enabling switch	`-fcheck-data-deps`
	Dump switch	`-fdump-tree-ckdd`
	Dump file extension	`.ckdd`
Loop Distribution	Pass variable name	`pass_loop_distribution`
	Enabling switch	`-ftree-loop-distribution`
	Dump switch	`-fdump-tree-ldist`
	Dump file extension	`.ldist`
Vectorization	Pass variable name	`pass_vectorize`
	Enabling switch	`-ftree-vectorize`
	Dump switch	`-fdump-tree-vect`
	Dump file extension	`.vect`
Parallelization	Pass variable name	`pass_parallelize_loops`
	Enabling switch	`-ftree-parallelize-loops=n`
	Dump switch	`-fdump-tree-parloops`
	Dump file extension	`.parloops`

Compiling for Emitting Dumps

- Other necessary command line switches
  - `-O2 -fdump-tree-all`
    `-O3` enables `-ftree-vectorize`. Other flags must be enabled explicitly
- Processor related switches to enable transformations apart from analysis
  - `-mtune=pentium -msse4`
- Other useful options
  - Suffixing `-all` to all dump switches
  - `-S` to stop the compilation with assembly generation
  - `--verbose-asm` to see more detailed assembly dump

Representing Value Spaces of Variables and Expressions

Chain of Recurrences: 3-tuple ⟨Starting Value, modification, stride⟩

```
for (i=3; i<=15; i=i+3)
{
    for (j=11; j>=1; j=j-2)
    {
        A[i+1][2*j-1] = ...
    }
}
```

Entity	CR
Induction variable i	$\{3, +, 3\}$
Induction variable j	$\{11, +, -2\}$
Index expression i+1	$\{4, +, 3\}$
Index expression 2*j-1	$\{21, +, -4\}$

Example 1: Observing Data Dependence

Step 0: Compiling

```
int a[200];
int main()
{
   int i;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}
```

```
gcc -fcheck-data-deps -fdump-tree-ckdd-all -O2 -S datadep.c
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

`int a[200];`
`int main()`
`{`
`  int i;`
`  for(i=0;i<150;i++)`
`  {`
`  a[i]=a[i+1]+2;`
`  }`
`  return 0;`
`}`
	`# BLOCK 3 freq:9900`
	`# iD.1358_13 = PHI <iD.1358_3(4), 0(2)>`
	`# .MEMD.1368_14 = PHI`
	`  <.MEMD.1368_10(4), .MEMD.1368_9(D)(2)>`
	`iD.1358_3 = iD.1358_13 + 1;`
	`D.1364_4 = aD.1355[iD.1358_3];`
	`D.1365_5 = D.1364_4 + 2;`
	`aD.1355[iD.1358_13] = D.1365_5;`
	`if (iD.1358_3 != 150)`
	`  goto <bb 4>;`
	`else`
	`  goto <bb 5>;`

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

`int a[200];`
`int main()`
`{`
`int i;`
`for(i=0;i<150;i++)`
`{`
`a[i]=a[i+1]+2;`
`}`
`return 0;`
`}`
	`# BLOCK 3 freq:9900`
	`# iD.1358_13 = PHI <iD.1358_3(4), 0(2)>`
	`# .MEMD.1368_14 = PHI`
	`<.MEMD.1368_10(4), .MEMD.1368_9(D)(2)>`
	`iD.1358_3 = iD.1358_13 + 1;`
	`D.1364_4 = aD.1355[iD.1358_3];`
	`D.1365_5 = D.1364_4 + 2;`
	`aD.1355[iD.1358_13] = D.1365_5;`
	`if (iD.1358_3 != 150)`
	`goto <bb 4>;`
	`else`
	`goto <bb 5>;`

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

`int a[200];`
`int main()`
`{`
`int i;`
`for(i=0;i<150;i++)`
`{`
`a[i]=a[i+1]+2;`
`}`
`return 0;`
`}`
	`# BLOCK 3 freq:9900`
	`# iD.1358_13 = PHI <iD.1358_3(4), 0(2)>`
	`# .MEMD.1368_14 = PHI`
	`<.MEMD.1368_10(4), .MEMD.1368_9(D)(2)>`
	`iD.1358_3 = iD.1358_13 + 1;`
	`D.1364_4 = aD.1355[iD.1358_3];`
	`D.1365_5 = D.1364_4 + 2;`
	`aD.1355[iD.1358_13] = D.1365_5;`
	`if (iD.1358_3 != 150)`
	`goto <bb 4>;`
	`else`
	`goto <bb 5>;`

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

`int a[200];`
`int main()`
`{`
`int i;`
`for(i=0;i<150;i++)`
`{`
`a[i]=a[i+1]+2;`
`}`
`return 0;`
`}`
	`# BLOCK 3 freq:9900`
	`# iD.1358_13 = PHI <iD.1358_3(4), 0(2)>`
	`# .MEMD.1368_14 = PHI`
	`<.MEMD.1368_10(4), .MEMD.1368_9(D)(2)>`
	`iD.1358_3 = iD.1358_13 + 1;`
	`D.1364_4 = aD.1355[iD.1358_3];`
	`D.1365_5 = D.1364_4 + 2;`
	`aD.1355[iD.1358_13] = D.1365_5;`
	`if (iD.1358_3 != 150)`
	`goto <bb 4>;`
	`else`
	`goto <bb 5>;`

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 2>:
  i = 0;
  goto <bb 4>;

<bb 3>:
  D.1856 = i + 1;
  D.1857 = a[D.1856];
  D.1858 = D.1857 + 2;
  a[i] = D.1858;
  i = i + 1;

<bb 4>:
  if (i <= 149)
    goto <bb 3>;
  else
    goto <bb 5>;
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 2>:
  i = 0;
  goto <bb 4>;

<bb 3>:
  D.1856 = i + 1;
  D.1857 = a[D.1856];
  D.1858 = D.1857 + 2;
  a[i] = D.1858;
  i = i + 1;

<bb 4>:
  if (i <= 149)
    goto <bb 3>;
  else
    goto <bb 5>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                address: 4
aligned to: 128
(chrec = {1, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 2>:
  i = 0;
  goto <bb 4>;

<bb 3>:
  D.1856 = i + 1;
  D.1857 = a[D.1856];
  D.1858 = D.1857 + 2;
  a[i] = D.1858;
  i = i + 1;

<bb 4>:
  if (i <= 149)
    goto <bb 3>;
  else
    goto <bb 5>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                address:  0
aligned to: 128
base_object: a[0]
(chrec = {0, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 2>:
  i = 0;
  goto <bb 4>;

<bb 3>:
  D.1856 = i + 1;
  D.1857 = a[D.1856];
  D.1858 = D.1857 + 2;
  a[i] = D.1858;
  i = i + 1;

<bb 4>:
  if (i <= 149)
    goto <bb 3>;
  else
    goto <bb 5>;
```

(scalar_evolution = {0, +, 1}_1)

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 2>:
  i = 0;
  goto <bb 4>;

<bb 3>:
  D.1856 = i + 1;
  D.1857 = a[D.1856];
  D.1858 = D.1857 + 2;
  a[i] = D.1858;
  i = i + 1;

<bb 4>:
  if (i <= 149)
    goto <bb 3>;
  else
    goto <bb 5>;
```

(scalar_evolution = {1, +, 1}_1)

Example 1: Observing Data Dependence

Step 3: Observing the data dependence information

for(i=0;i ¡ 150;i++)
a[i]=a[i+1] + 2;

```
iterations_that_access_an_element_twice_in_A: [1 + 1*x_1]
last_conflict: 149
iterations_that_access_an_element_twice_in_B: [0 + 1*x_1]
last_conflict: 149
Subscript distance: 1
```

```
inner loop index: 0
loop nest: (1)
distance_vector: 1
direction_vector: +
```

Example 2: Observing Vectorization and Parallelization

Step 0: Compiling the code with -O2

```
int a[4096], b[4096];
int main()
{
    int i;
    for (i=0; i<4096; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

- Additional options for parallelization
  -ftree-parallelize-loops=8 -fdump-tree-parloops-all
- Additional options for vectorization
  -fdump-tree-vect-all -msse4 -ftree-vectorize

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program	Control Flow Graph

`int a[4096], b[4096];`
`int main()`
`{`
`  int i;`
`  for(i=0;i<4096;i++)`
`  {`
`  a[i] = b[i];`
`  }`
`  return 0;`
`}`
	`# BLOCK 3 freq:9900`
	`# iD.1359_11=PHI <iD.1359_4(4), 0(2)>`
	`# .MEMD.1367_12 = PHI`
	`  <.MEMD.1367_8(4),.MEMD.1367_7(D)(2)>`
	`  D.1364_3 = bD.1356[iD.1359_11];`
	`  aD.1355[iD.1359_11] = D.1364_3;`
	`  iD.1359_4 = iD.1359_11 + 1;`
	`  if (iD.1359_4 != 4096)`
	`  goto <bb 4>;`
	`  else`
	`  goto <bb 5>;`

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program	Control Flow Graph

`int a[4096], b[4096];`
`int main()`
`{`
`  int i;`
`  for(i=0;i<4096;i++)`
`  {`
`  a[i] = b[i];`
`  }`
`  return 0;`
`}`
	`# BLOCK 3 freq:9900`
	`# iD.1359_11=PHI <iD.1359_4(4), 0(2)>`
	`# .MEMD.1367_12 = PHI`
	`  <.MEMD.1367_8(4),.MEMD.1367_7(D)(2)>`
	`  D.1364_3 = bD.1356[iD.1359_11];`
	`  aD.1355[iD.1359_11] = D.1364_3;`
	`  iD.1359_4 = iD.1359_11 + 1;`
	`  if (iD.1359_4 != 4096)`
	`  goto <bb 4>;`
	`  else`
	`  goto <bb 5>;`

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program	Control Flow Graph

`int a[4096], b[4096];`
`int main()`
`{`
`  int i;`
`  for(i=0;i<4096;i++)`
`  {`
`  a[i] = b[i];`
`  }`
`  return 0;`
`}`
	`# BLOCK 3 freq:9900`
	`# iD.1359_11=PHI <iD.1359_4(4), 0(2)>`
	`# .MEMD.1367_12 = PHI`
	`  <.MEMD.1367_8(4),.MEMD.1367_7(D)(2)>`
	`  D.1364_3 = bD.1356[iD.1359_11];`
	`  aD.1355[iD.1359_11] = D.1364_3;`
	`  iD.1359_4 = iD.1359_11 + 1;`
	`  if (iD.1359_4 != 4096)`
	`  goto <bb 4>;`
	`  else`
	`  goto <bb 5>;`

Example 2: Observing Vectorization and Parallelization

Step 2: Observing the final decision about vectorization

```
      5: LOOP VECTORIZED.
Example2.c:3: note: vectorized 1 loops in function.
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
# BLOCK 3 freq:9900
# iD.1359_11 =
PHI <iD.1359_4(4), 0(2)>
# .MEMD.1367_12 = PHI
  <.MEMD.1367_8(4),
  .MEMD.1367_7(D)(2)>
 D.1364_3 = bD.1356[iD.1359_11];
 aD.1355[iD.1359_11] = D.1364_3;
 iD.1359_4 = iD.1359_11 + 1;
 if (iD.1359_4 != 4096)
   goto <bb 4>;
 else
   goto <bb 5>;
```

Transformed control flow graph

```
<bb 2>:
  vect_pb.8_10 = &b;
  vect_pa.13_15 = &a;
<bb 3>:
 # vect_pb.5_6 = PHI
    <vect_pb.5_13(4),vect_pb.8_10(2)>
 # vect_pa.10_16 = PHI <vect_pa.10_17(4),
    vect_pa.13_15(2)>
 vect_var_.9_14=MEM[(int[4096] *)vect_pb.5_6];
 MEM[(int[4096] *)vect_pa.10_16]=vect_var_.9_14;
 ivtmp.4_9 = ivtmp.4_1 - 1;
 vect_pb.5_13 = vect_pb.5_6 + 16;
 vect_pa.10_17 = vect_pa.10_16 + 16;
 ivtmp.14_19 = ivtmp.14_18 + 1;
 if (ivtmp.14_19 < 1024)
   goto <bb 4>;
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
# BLOCK 3 freq:9900
# iD.1359_11 =
PHI <iD.1359_4(4), 0(2)>
# .MEMD.1367_12 = PHI
  <.MEMD.1367_8(4),
  .MEMD.1367_7(D)(2)>
 D.1364_3 = bD.1356[iD.1359_11];
 aD.1355[iD.1359_11] = D.1364_3;
 iD.1359_4 = iD.1359_11 + 1;
 if (iD.1359_4 != 4096)
   goto <bb 4>;
 else
   goto <bb 5>;
```

Transformed control flow graph

```
<bb 2>:
  vect_pb.8_10 = &b;
  vect_pa.13_15 = &a;
<bb 3>:
 # vect_pb.5_6 = PHI
    <vect_pb.5_13(4),vect_pb.8_10(2)>
 # vect_pa.10_16 = PHI <vect_pa.10_17(4),
    vect_pa.13_15(2)>
 vect_var_.9_14=MEM[(int[4096] *)vect_pb.5_6];
 MEM[(int[4096] *)vect_pa.10_16]=vect_var_.9_14;
 ivtmp.4_9 = ivtmp.4_1 - 1;
 vect_pb.5_13 = vect_pb.5_6 + 16;
 vect_pa.10_17 = vect_pa.10_16 + 16;
 ivtmp.14_19 = ivtmp.14_18 + 1;
 if (ivtmp.14_19 < 1024)
   goto <bb 4>;
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
# BLOCK 3 freq:9900
# iD.1359_11 =
PHI <iD.1359_4(4), 0(2)>
# .MEMD.1367_12 = PHI
  <.MEMD.1367_8(4),
  .MEMD.1367_7(D)(2)>
 D.1364_3 = bD.1356[iD.1359_11];
 aD.1355[iD.1359_11] = D.1364_3;
 iD.1359_4 = iD.1359_11 + 1;
 if (iD.1359_4 != 4096)
   goto <bb 4>;
 else
   goto <bb 5>;
```

Transformed control flow graph

```
<bb 2>:
  vect_pb.8_10 = &b;
  vect_pa.13_15 = &a;
<bb 3>:
 # vect_pb.5_6 = PHI
    <vect_pb.5_13(4),vect_pb.8_10(2)>
 # vect_pa.10_16 = PHI <vect_pa.10_17(4),
    vect_pa.13_15(2)>
 vect_var_.9_14=MEM[(int[4096] *)vect_pb.5_6];
 MEM[(int[4096] *)vect_pa.10_16]=vect_var_.9_14;
 ivtmp.4_9 = ivtmp.4_1 - 1;
 vect_pb.5_13 = vect_pb.5_6 + 16;
 vect_pa.10_17 = vect_pa.10_16 + 16;
 ivtmp.14_19 = ivtmp.14_18 + 1;
 if (ivtmp.14_19 < 1024)
   goto <bb 4>;
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
# BLOCK 3 freq:9900
# iD.1359_11 =
PHI <iD.1359_4(4), 0(2)>
# .MEMD.1367_12 = PHI
  <.MEMD.1367_8(4),
  .MEMD.1367_7(D)(2)>
 D.1364_3 = bD.1356[iD.1359_11];
 aD.1355[iD.1359_11] = D.1364_3;
 iD.1359_4 = iD.1359_11 + 1;
 if (iD.1359_4 != 4096)
   goto <bb 4>;
 else
   goto <bb 5>;
```

Transformed control flow graph

```
<bb 2>:
  vect_pb.8_10 = &b;
  vect_pa.13_15 = &a;
<bb 3>:
 # vect_pb.5_6 = PHI
    <vect_pb.5_13(4),vect_pb.8_10(2)>
 # vect_pa.10_16 = PHI <vect_pa.10_17(4),
    vect_pa.13_15(2)>
 vect_var_.9_14=MEM[(int[4096] *)vect_pb.5_6];
 MEM[(int[4096] *)vect_pa.10_16]=vect_var_.9_14;
 ivtmp.4_9 = ivtmp.4_1 - 1;
 vect_pb.5_13 = vect_pb.5_6 + 16;
 vect_pa.10_17 = vect_pa.10_16 + 16;
 ivtmp.14_19 = ivtmp.14_18 + 1;
 if (ivtmp.14_19 < 1024)
   goto <bb 4>;
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
# BLOCK 3 freq:9900
# iD.1359_11 =
PHI <iD.1359_4(4), 0(2)>
# .MEMD.1367_12 = PHI
  <.MEMD.1367_8(4),
  .MEMD.1367_7(D)(2)>
 D.1364_3 = bD.1356[iD.1359_11];
 aD.1355[iD.1359_11] = D.1364_3;
 iD.1359_4 = iD.1359_11 + 1;
 if (iD.1359_4 != 4096)
   goto <bb 4>;
 else
   goto <bb 5>;
```

Transformed control flow graph

```
<bb 2>:
  vect_pb.8_10 = &b;
  vect_pa.13_15 = &a;
<bb 3>:
 # vect_pb.5_6 = PHI
    <vect_pb.5_13(4),vect_pb.8_10(2)>
 # vect_pa.10_16 = PHI <vect_pa.10_17(4),
    vect_pa.13_15(2)>
 vect_var_.9_14=MEM[(int[4096] *)vect_pb.5_6];
 MEM[(int[4096] *)vect_pa.10_16]=vect_var_.9_14;
 ivtmp.4_9 = ivtmp.4_1 - 1;
 vect_pb.5_13 = vect_pb.5_6 + 16;
 vect_pa.10_17 = vect_pa.10_16 + 16;
 ivtmp.14_19 = ivtmp.14_18 + 1;
 if (ivtmp.14_19 < 1024)
   goto <bb 4>;
```

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread

  $\Rightarrow$ Distribute iteration space across all threads

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread

  $\Rightarrow$ Distribute iteration space across all threads
- Create the following code body for each thread t_i

```
for (j=start_for_thread_i; j<=end_for_thread_i; j++)
{
     /* execute the loop body to be parallelized */
}
```

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  $\Rightarrow$ Distribute iteration space across all threads
- Create the following code body for each thread t_i

```
for (j=start_for_thread_i; j<=end_for_thread_i; j++)
{
     /* execute the loop body to be parallelized */
}
```

- All threads are executed in parallel

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1896_6 = omp_get_num_threadsD.1240 ();
D.1897_7 = (<unnamed-unsigned:32>) D.1896_6;
D.1898_8 = omp_get_thread_numD.1239 ();
D.1899_9 = (<unnamed-unsigned:32>) D.1898_8;
q.16D.1900_10 = 4095 / D.1897_7;
tt.17D.1901_11 = 4095 % D.1897_7;
if (D.1899_9 < tt.17D.1901_11)
  goto <bb 9>;
 else
  goto <bb 3>;

<bb 9>:
q.16D.1900_13 = q.16D.1900_10 + 1;
  goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1896_6 = omp_get_num_threadsD.1240 ();
D.1897_7 = (<unnamed-unsigned:32>) D.1896_6;
D.1898_8 = omp_get_thread_numD.1239 ();
D.1899_9 = (<unnamed-unsigned:32>) D.1898_8;
q.16D.1900_10 = 4095 / D.1897_7;
tt.17D.1901_11 = 4095 % D.1897_7;
if (D.1899_9 < tt.17D.1901_11)
  goto <bb 9>;
 else
  goto <bb 3>;

<bb 9>:
q.16D.1900_13 = q.16D.1900_10 + 1;
  goto <bb 3>;
```

Get the number of threads

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1896_6 = omp_get_num_threadsD.1240 ();
D.1897_7 = (<unnamed-unsigned:32>) D.1896_6;
D.1898_8 = omp_get_thread_numD.1239 ();
D.1899_9 = (<unnamed-unsigned:32>) D.1898_8;
q.16D.1900_10 = 4095 / D.1897_7;
tt.17D.1901_11 = 4095 % D.1897_7;
if (D.1899_9 < tt.17D.1901_11)
  goto <bb 9>;
 else
  goto <bb 3>;

<bb 9>:
q.16D.1900_13 = q.16D.1900_10 + 1;
  goto <bb 3>;
```

Get thread identity

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1896_6 = omp_get_num_threadsD.1240 ();
D.1897_7 = (<unnamed-unsigned:32>) D.1896_6;
D.1898_8 = omp_get_thread_numD.1239 ();
D.1899_9 = (<unnamed-unsigned:32>) D.1898_8;
q.16D.1900_10 = 4095 / D.1897_7;
tt.17D.1901_11 = 4095 % D.1897_7;
if (D.1899_9 < tt.17D.1901_11)
  goto <bb 9>;
 else
  goto <bb 3>;

<bb 9>:
q.16D.1900_13 = q.16D.1900_10 + 1;
  goto <bb 3>;
```

Perform load calculations

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1896_6 = omp_get_num_threadsD.1240 ();
D.1897_7 = (<unnamed-unsigned:32>) D.1896_6;
D.1898_8 = omp_get_thread_numD.1239 ();
D.1899_9 = (<unnamed-unsigned:32>) D.1898_8;
q.16D.1900_10 = 4095 / D.1897_7;
tt.17D.1901_11 = 4095 % D.1897_7;
if (D.1899_9 < tt.17D.1901_11)
  goto <bb 9>;
 else
  goto <bb 3>;

<bb 9>:
q.16D.1900_13 = q.16D.1900_10 + 1;
  goto <bb 3>;
```

Assign end iteration to the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1896_6 = omp_get_num_threadsD.1240 ();
D.1897_7 = (<unnamed-unsigned:32>) D.1896_6;
D.1898_8 = omp_get_thread_numD.1239 ();
D.1899_9 = (<unnamed-unsigned:32>) D.1898_8;
q.16D.1900_10 = 4095 / D.1897_7;
tt.17D.1901_11 = 4095 % D.1897_7;
if (D.1899_9 < tt.17D.1901_11)
  goto <bb 9>;
 else
  goto <bb 3>;

<bb 9>:
q.16D.1900_13 = q.16D.1900_10 + 1;
  goto <bb 3>;
```

Start execution of chosen thread

Example 2: Observing Vectorization and Parallelization

Total no. of iterations in the program: $N = 4096$
Iterations get executed in parallel in gcc-4.7.2: $N - 1 = 4095$
Total no. of threads we created $= 8$

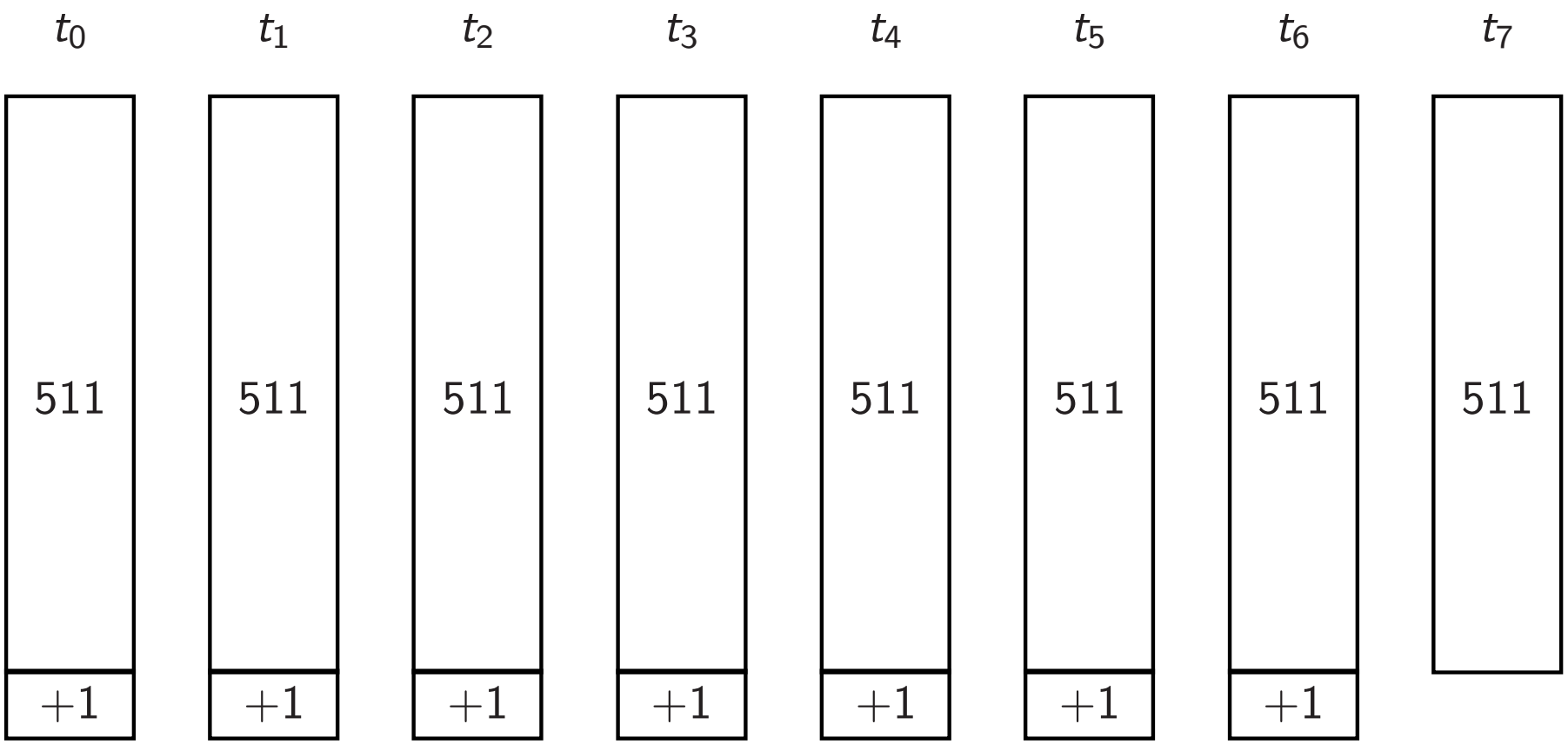
Example 2: Observing Vectorization and Parallelization

Highest count of iterations which each thread can have $= \left\lfloor \frac{4095}{8} \right\rfloor = 511$

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
511	511	511	511	511	511	511	511

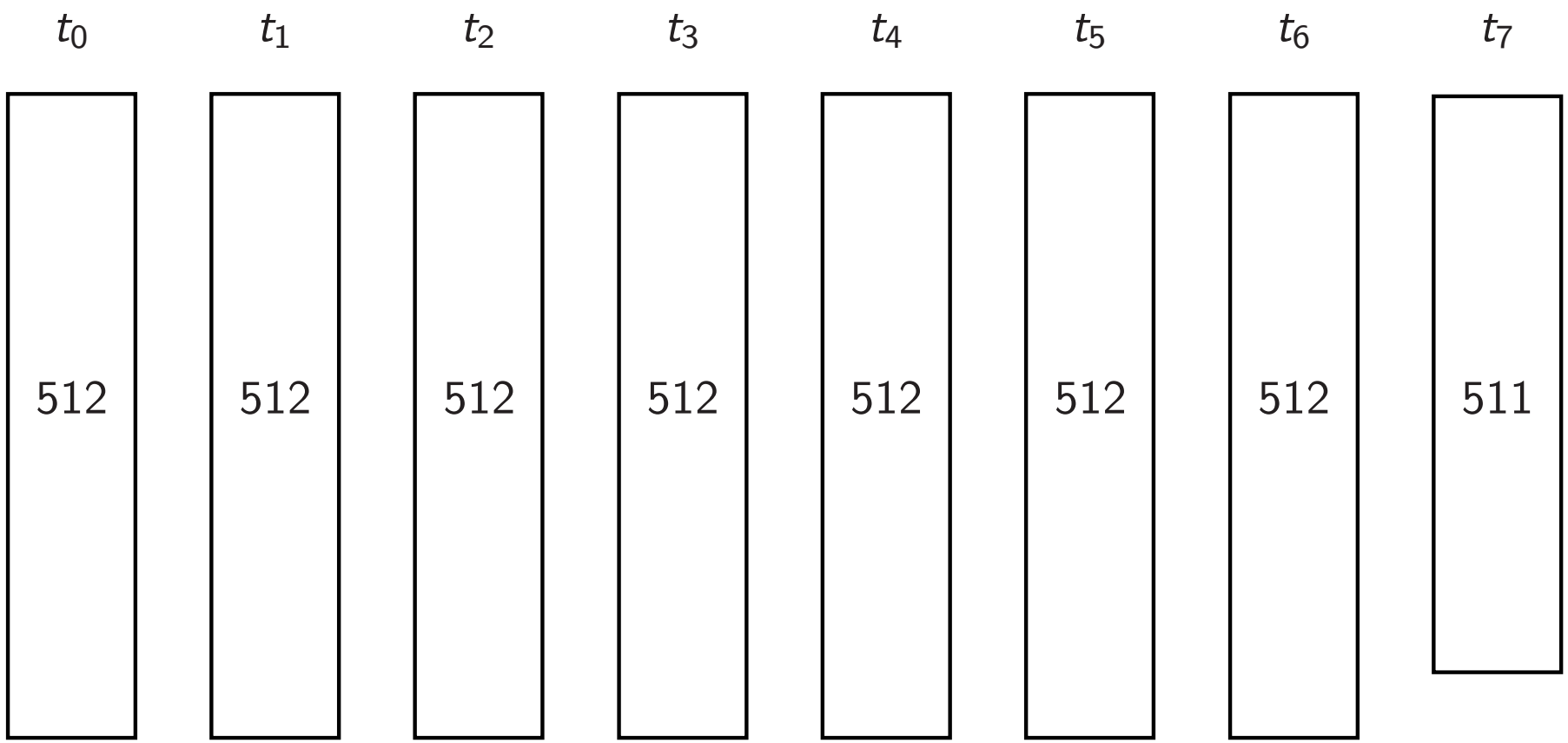
Example 2: Observing Vectorization and Parallelization

Remaining no. of iterations $= 4095\%8 = 7$
Distribution of remaining iterations



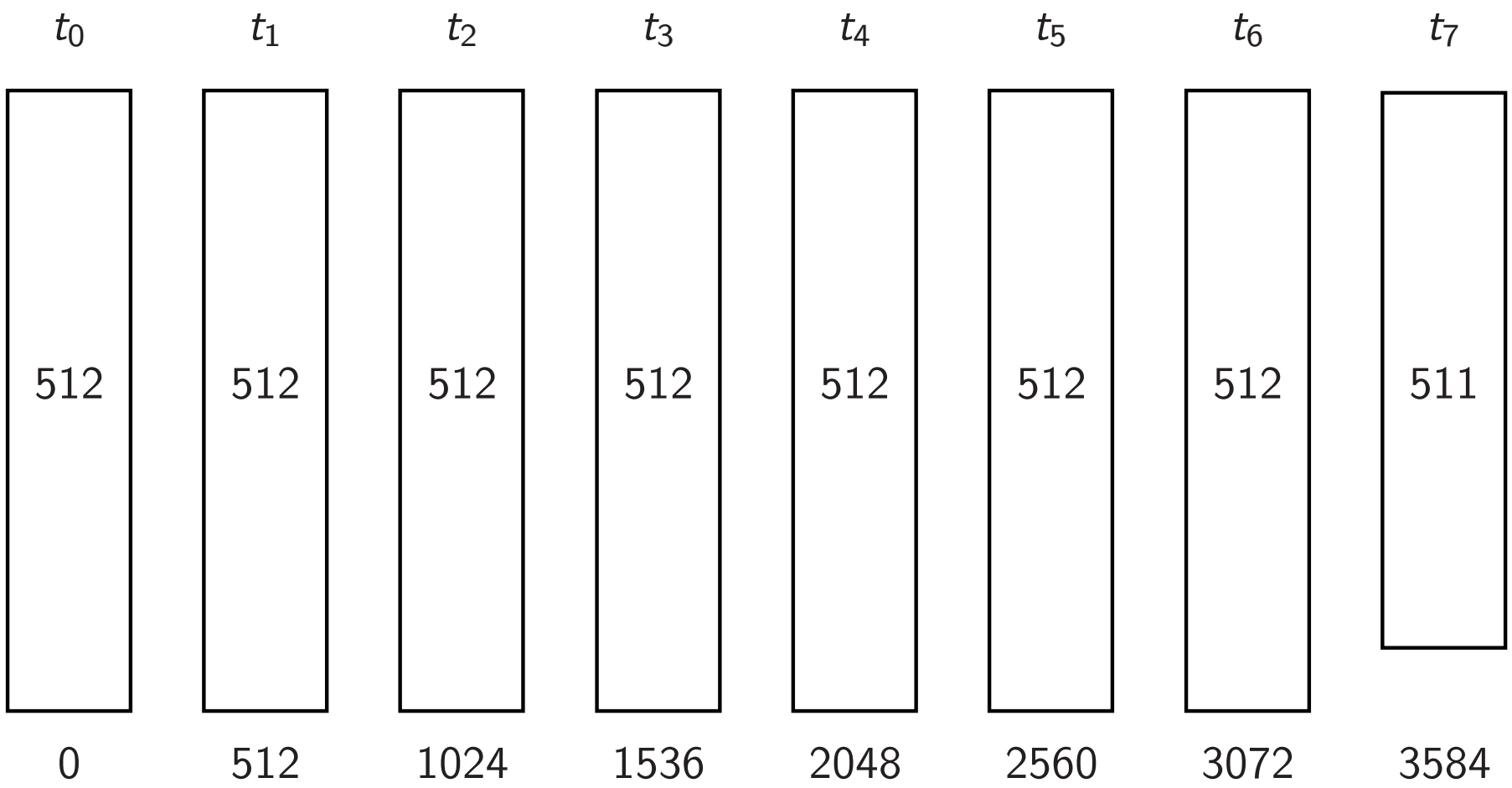
Example 2: Observing Vectorization and Parallelization

Remaining no. of iterations $= 4095\%8 = 7$
Distribution of remaining iterations



Example 2: Observing Vectorization and Parallelization

Distribution of iteration range over threads



Example 2: Observing Vectorization and Parallelization

Step 6: Execution of last iteration

```
.paral_data_store.14.b.6 = b.6_32;
.paral_data_store.14.a.7 = a.7_33;
__builtin_GOMP_parallel_start (main._loopfn.0,
        &.paral_data_store.14, 2);
main__loopfn.0 (&.paral_data_store.14);
__builtin_GOMP_parallel_end ();
ivtmp.5_24 = 4095;
i_25 = (int) ivtmp.5_24;
ivtmp.4_26 = 4096 - ivtmp.5_24;
D.1857_27 = b[i_25];
a[i_25] = D.1857_27;
i_29 = i_25 + 1;
ivtmp.4_30 = ivtmp.4_26 - 1;
ivtmp.5_31 = ivtmp.5_24 + 1;
```

Example 2: Observing Vectorization and Parallelization

Step 6: Execution of last iteration

```
.paral_data_store.14.b.6 = b.6_32;
.paral_data_store.14.a.7 = a.7_33;
__builtin_GOMP_parallel_start (main._loopfn.0,
        &.paral_data_store.14, 2);
main__loopfn.0 (&.paral_data_store.14);
__builtin_GOMP_parallel_end ();
ivtmp.5_24 = 4095;
i_25 = (int) ivtmp.5_24;
ivtmp.4_26 = 4096 - ivtmp.5_24;
D.1857_27 = b[i_25];
a[i_25] = D.1857_27;
i_29 = i_25 + 1;
ivtmp.4_30 = ivtmp.4_26 - 1;
ivtmp.5_31 = ivtmp.5_24 + 1;
```

Call for parallel execution

Example 2: Observing Vectorization and Parallelization

Step 6: Execution of last iteration

```
.paral_data_store.14.b.6 = b.6_32;
.paral_data_store.14.a.7 = a.7_33;
__builtin_GOMP_parallel_start (main._loopfn.0,
        &.paral_data_store.14, 2);
main__loopfn.0 (&.paral_data_store.14);
__builtin_GOMP_parallel_end ();
ivtmp.5_24 = 4095;
i_25 = (int) ivtmp.5_24;
ivtmp.4_26 = 4096 - ivtmp.5_24;
D.1857_27 = b[i_25];
a[i_25] = D.1857_27;
i_29 = i_25 + 1;
ivtmp.4_30 = ivtmp.4_26 - 1;
ivtmp.5_31 = ivtmp.5_24 + 1;
```

Execution of last iteration after parallel execution of all threads

Example 2: Observing Vectorization and Parallelization

Step 7: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

`<bb 3>:`
`  D.1857 = b[i];`
`  a[i] = D.1857;`
`  i = i + 1;`
 	
`<bb 4>:`
`  if (i <= 4095)`
`  goto <bb 3>;`
`  else`
`  goto <bb 5>;`
	`<bb 6>:`
	`  # ivtmp.8_20 = PHI <ivtmp.8_17(5), ivtmp.8_2`
	`  i.9_23 = (int) ivtmp.8_20;`
	`  D.1908_25 = *b.11_4[i.9_23];`
	`  *a.12_5[i.9_23] = D.1908_25;`
	`  ivtmp.8_21 = ivtmp.8_20 + 1;`
	`  if (D.1904_18 > ivtmp.8_21)`
	`  goto <bb 8>;`
	`  else`
	`  goto <bb 7>;`

Example 2: Observing Vectorization and Parallelization

Step 7: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

<bb 3>:
D.1857 = b[i];
a[i] = D.1857;
i = i + 1;
 	
<bb 4>:
if (i <= 4095)
goto <bb 3>;
else
goto <bb 5>;
	<bb 6>:
	# ivtmp.8_20 = PHI <ivtmp.8_17(5), ivtmp.8_2
	i.9_23 = (int) ivtmp.8_20;
	D.1908_25 = *b.11_4[i.9_23];
	*a.12_5[i.9_23] = D.1908_25;
	ivtmp.8_21 = ivtmp.8_20 + 1;
	if (D.1904_18 > ivtmp.8_21)
	goto <bb 8>;
	else
	goto <bb 7>;

Example 2: Observing Vectorization and Parallelization

Step 7: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

`<bb 3>:`
`  D.1857 = b[i];`
`  a[i] = D.1857;`
`  i = i + 1;`
 	
`<bb 4>:`
`  if (i <= 4095)`
`  goto <bb 3>;`
`  else`
`  goto <bb 5>;`
	`<bb 6>:`
	`  # ivtmp.8_20 = PHI <ivtmp.8_17(5), ivtmp.8_2`
	`  i.9_23 = (int) ivtmp.8_20;`
	`  D.1908_25 = *b.11_4[i.9_23];`
	`  *a.12_5[i.9_23] = D.1908_25;`
	`  ivtmp.8_21 = ivtmp.8_20 + 1;`
	`  if (D.1904_18 > ivtmp.8_21)`
	`  goto <bb 8>;`
	`  else`
	`  goto <bb 7>;`

Example 2: Observing Vectorization and Parallelization

Step 7: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

`<bb 3>:`
`  D.1857 = b[i];`
`  a[i] = D.1857;`
`  i = i + 1;`
 	
`<bb 4>:`
`  if (i <= 4095)`
`  goto <bb 3>;`
`  else`
`  goto <bb 5>;`
	`<bb 6>:`
	`  # ivtmp.8_20 = PHI <ivtmp.8_17(5), ivtmp.8_2`
	`  i.9_23 = (int) ivtmp.8_20;`
	`  D.1908_25 = *b.11_4[i.9_23];`
	`  *a.12_5[i.9_23] = D.1908_25;`
	`  ivtmp.8_21 = ivtmp.8_20 + 1;`
	`  if (D.1904_18 > ivtmp.8_21)`
	`  goto <bb 8>;`
	`  else`
	`  goto <bb 7>;`

Example 2: Observing Vectorization and Parallelization

Step 7: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

`<bb 3>:`
`  D.1857 = b[i];`
`  a[i] = D.1857;`
`  i = i + 1;`
 	
`<bb 4>:`
`  if (i <= 4095)`
`  goto <bb 3>;`
`  else`
`  goto <bb 5>;`
	`<bb 6>:`
	`  # ivtmp.8_20 = PHI <ivtmp.8_17(5), ivtmp.8_2`
	`  i.9_23 = (int) ivtmp.8_20;`
	`  D.1908_25 = *b.11_4[i.9_23];`
	`  *a.12_5[i.9_23] = D.1908_25;`
	`  ivtmp.8_21 = ivtmp.8_20 + 1;`
	`  if (D.1904_18 > ivtmp.8_21)`
	`  goto <bb 8>;`
	`  else`
	`  goto <bb 7>;`

Example 3: Vectorization but No Parallelization

Step 0: Compiling with
`-O2 -fdump-tree-vect-all -msse4 -ftree-vectorize`

```
int a[624];
int main()
{
    int i;
    for (i=0; i<619; i++)
    {
        a[i] = a[i+4];
    }
    return 0;
}
```

Example 3: Vectorization but No Parallelization

Step 1: Observing the final decision about vectorization

```
  6: LOOP VECTORIZED.
example3.c:3: note: vectorized 1 loops in function.
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  D.1820 = i + 4;
  D.1821 = a[D.1820];
  a[i] = D.1821;
  i = i + 1;

<bb 4>:
  if (i <= 618)]
    goto <bb 3>;
  else
    goto <bb 5>;
```

Vectorized Control Flow Graph

```
<bb 2>:
 vect_pa.11_31 = &MEM[(void *)&a + 16B];
 vect_pa.16_35 = &a;
<bb 3>:
 # vect_pa.8_32 = PHI <vect_pa.8_33(7),
    vect_pa.11_31(2)>
 # vect_pa.13_36 = PHI <vect_pa.13_37(7),
    vect_pa.16_35(2)>
 vect_var_.12_34=MEM[(int[624] *)vect_pa.8_32]
 MEM[(int[624] *)vect_pa.13_36] =
    vect_var_.12_34;
 vect_pa.8_33 = vect_pa.8_32 + 16;
 vect_pa.13_37 = vect_pa.13_36 + 16;
 ivtmp.17_39 = ivtmp.17_38 + 1;
 if (ivtmp.17_39 < 154)
   goto <bb 7>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  D.1820 = i + 4;
  D.1821 = a[D.1820];
  a[i] = D.1821;
  i = i + 1;

<bb 4>:
  if (i <= 618)]
    goto <bb 3>;
  else
    goto <bb 5>;
```

Vectorized Control Flow Graph

```
<bb 2>:
 vect_pa.11_31 = &MEM[(void *)&a + 16B];
 vect_pa.16_35 = &a;
<bb 3>:
 # vect_pa.8_32 = PHI <vect_pa.8_33(7),
    vect_pa.11_31(2)>
 # vect_pa.13_36 = PHI <vect_pa.13_37(7),
    vect_pa.16_35(2)>
 vect_var_.12_34=MEM[(int[624] *)vect_pa.8_32]
 MEM[(int[624] *)vect_pa.13_36] =
    vect_var_.12_34;
 vect_pa.8_33 = vect_pa.8_32 + 16;
 vect_pa.13_37 = vect_pa.13_36 + 16;
 ivtmp.17_39 = ivtmp.17_38 + 1;
 if (ivtmp.17_39 < 154)
   goto <bb 7>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  D.1820 = i + 4;
  D.1821 = a[D.1820];
  a[i] = D.1821;
  i = i + 1;

<bb 4>:
  if (i <= 618)]
    goto <bb 3>;
  else
    goto <bb 5>;
```

Vectorized Control Flow Graph

```
<bb 2>:
 vect_pa.11_31 = &MEM[(void *)&a + 16B];
 vect_pa.16_35 = &a;
<bb 3>:
 # vect_pa.8_32 = PHI <vect_pa.8_33(7),
    vect_pa.11_31(2)>
 # vect_pa.13_36 = PHI <vect_pa.13_37(7),
    vect_pa.16_35(2)>
 vect_var_.12_34=MEM[(int[624] *)vect_pa.8_32];
 MEM[(int[624] *)vect_pa.13_36] =
    vect_var_.12_34;
 vect_pa.8_33 = vect_pa.8_32 + 16;
 vect_pa.13_37 = vect_pa.13_36 + 16;
 ivtmp.17_39 = ivtmp.17_38 + 1;
 if (ivtmp.17_39 < 154)
   goto <bb 7>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  D.1820 = i + 4;
  D.1821 = a[D.1820];
  a[i] = D.1821;
  i = i + 1;

<bb 4>:
  if (i <= 618)]
    goto <bb 3>;
  else
    goto <bb 5>;
```

Vectorized Control Flow Graph

```
<bb 2>:
 vect_pa.11_31 = &MEM[(void *)&a + 16B];
 vect_pa.16_35 = &a;
<bb 3>:
 # vect_pa.8_32 = PHI <vect_pa.8_33(7),
    vect_pa.11_31(2)>
 # vect_pa.13_36 = PHI <vect_pa.13_37(7),
    vect_pa.16_35(2)>
 vect_var_.12_34=MEM[(int[624] *)vect_pa.8_32]
 MEM[(int[624] *)vect_pa.13_36] =
    vect_var_.12_34;
 vect_pa.8_33 = vect_pa.8_32 + 16;
 vect_pa.13_37 = vect_pa.13_36 + 16;
 ivtmp.17_39 = ivtmp.17_38 + 1;
 if (ivtmp.17_39 < 154)
   goto <bb 7>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  D.1820 = i + 4;
  D.1821 = a[D.1820];
  a[i] = D.1821;
  i = i + 1;

<bb 4>:
  if (i <= 618)]
    goto <bb 3>;
  else
    goto <bb 5>;
```

Vectorized Control Flow Graph

```
<bb 2>:
 vect_pa.11_31 = &MEM[(void *)&a + 16B];
 vect_pa.16_35 = &a;
<bb 3>:
 # vect_pa.8_32 = PHI <vect_pa.8_33(7),
    vect_pa.11_31(2)>
 # vect_pa.13_36 = PHI <vect_pa.13_37(7),
    vect_pa.16_35(2)>
 vect_var_.12_34=MEM[(int[624] *)vect_pa.8_32]
 MEM[(int[624] *)vect_pa.13_36] =
    vect_var_.12_34;
 vect_pa.8_33 = vect_pa.8_32 + 16;
 vect_pa.13_37 = vect_pa.13_36 + 16;
 ivtmp.17_39 = ivtmp.17_38 + 1;
 if (ivtmp.17_39 < 154)
   goto <bb 7>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  D.1820 = i + 4;
  D.1821 = a[D.1820];
  a[i] = D.1821;
  i = i + 1;

<bb 4>:
  if (i <= 618)]
    goto <bb 3>;
  else
    goto <bb 5>;
```

Vectorized Control Flow Graph

```
<bb 2>:
 vect_pa.11_31 = &MEM[(void *)&a + 16B];
 vect_pa.16_35 = &a;
<bb 3>:
 # vect_pa.8_32 = PHI <vect_pa.8_33(7),
    vect_pa.11_31(2)>
 # vect_pa.13_36 = PHI <vect_pa.13_37(7),
    vect_pa.16_35(2)>
 vect_var_.12_34=MEM[(int[624] *)vect_pa.8_32]
 MEM[(int[624] *)vect_pa.13_36] =
    vect_var_.12_34;
 vect_pa.8_33 = vect_pa.8_32 + 16;
 vect_pa.13_37 = vect_pa.13_36 + 16;
 ivtmp.17_39 = ivtmp.17_38 + 1;
 if (ivtmp.17_39 < 154)
   goto <bb 7>;
```

Example 3: Vectorization but No Parallelization

- Step 3: Observing the conclusion about dependence information

```
inner loop index: 0
loop nest: (1 )
distance_vector: 4
direction_vector: +
```

- Step 4: Observing the final decision about parallelization

```
FAILED: data dependency between iterations
```

Example 4: No Vectorization and No Parallelization

Step 0: Compiling the code with -O2

```
int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<216; i++)
    {
        a[i+2] = b[i] + 5;
        b[i+1] = a[i] + 10;
    }
    return 0;
}
```

- Additional options for parallelization

  -ftree-parallelize-loops=2 -fdump-tree-parloops-all

- Additional options for vectorization

  -fdump-tree-vect-all -msse4 -ftree-vectorize

Example 4: No Vectorization and No Parallelization

- Step 1: Observing the final decision about vectorization

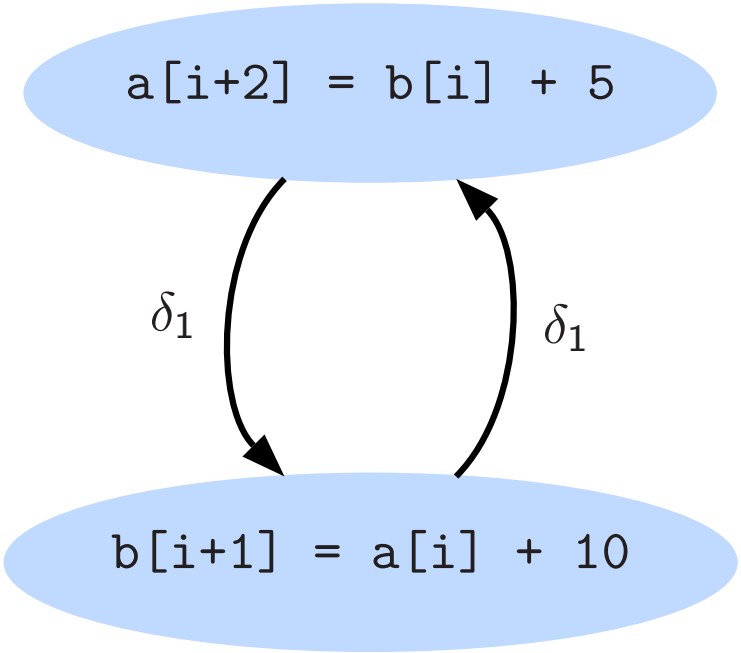
```
example4.c:5: note: vectorized 0 loops in function.
```

- Step 2: Observing the final decision about parallelization

```
FAILED: data dependencies exist across iterations
```

Example 4: No Vectorization and No Parallelization

Step 3: Understanding the dependences that prohibit vectorization and parallelization



Part 3

Transformations Enhancing Vectorization and Parallelization

Transformations Enhancing Vectorization and Parallelization

Some transformations increase the scope of parallelization and vectorization by either enabling them, or by improving their run time performance. Most important of such transformations are:

- Loop Interchange
- Loop Distribution
- Loop Fusion
- Loop Peeling
- Loop alignment

Loop Interchange

Loop Interchange for Vectorization

Original Code

```
for (i=0; i<120; i++) {
   for (j=0; j<200; j++)
      a[j][i] = a[j][i+1]
+ 5; }
```

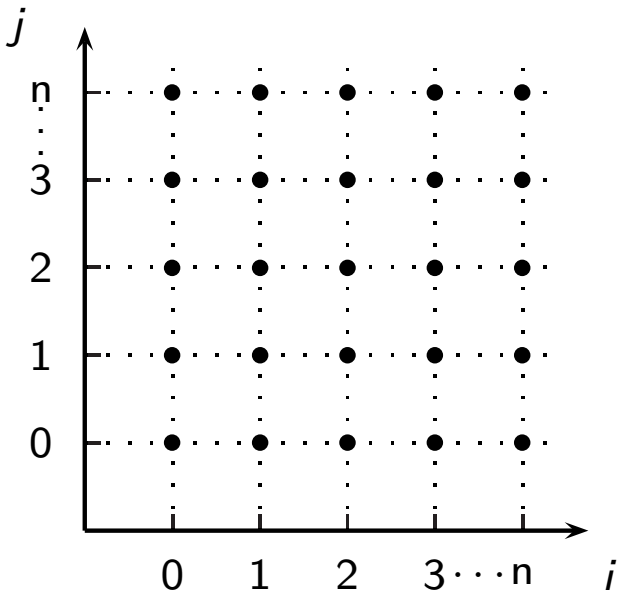
- Outer loop is vectorizable
- Mismatch between nesting order of loops and array access

Loop Interchange

Loop Interchange for Vectorization

Original Code

```
for (i=0; i<120; i++) {
   for (j=0; j<200; j++)
      a[j][i] = a[j][i+1]
+ 5; }
```



- Outer loop is vectorizable
- Mismatch between nesting order of loops and array access

Loop Interchange

Loop Interchange for Vectorization

Original Code

```
for (i=0; i<120; i++) {
   for (j=0; j<200; j++)
      a[j][i] = a[j][i+1]
+ 5; }
```

After Interchange

```
for (j=0; j<200; j++) {
   for (i=0; i<120; i++)
      a[j][i] = a[j][i+1]
+ 5; }
```

- Innermost loop is vectorizable
- Loop Interchange improves data locality

Loop Interchange

Original Code and Respective Assembly Code

Original Code

```
int a[256][256];
int main()
{
int i, j;
for (i=0; i<120; i++)
 {
  for (j = 0; j < 200; j++)
  a[j][i] = a[j][i+1] + 5;
 }
return 0;
}
```

Assembly Code

```
main:
movl $a+4, %edx
movdqa .LC0, %xmm1
.L2:
leal -4(%edx), %ecx
xorl %eax, %eax
.L3:
movdqu (%edx,%eax), %xmm0
paddd %xmm1, %xmm0
movdqa %xmm0, (%ecx,%eax)
addl $1024, %eax
cmpl $204800, %eax
jne .L3
addl $16, %edx
cmpl $a+484, %edx
jne .L2
```

```
.LC0:
.long 5
&.long 5
.long 5
.long 5
```

Loop Interchange

Transformed Code and Respective Assembly Code

Transformed Code

```
int a[256][256];
int main()
{
int i, j;
for (j = 0; j < 200; j++)
 {
  for (i=0; i<120; i++)
   a[j][i] = a[j][i+1] + 5;
 }
return 0;
}
```

Assembly Code

```
main:
movl $a+4, %edx
movdqa .LC0, %xmm1
.L2:
leal -4(%edx), %ecx
xorl %eax, %eax
.L3:
movdqu (%edx,%eax), %xmm0
paddd %xmm1, %xmm0
movdqa %xmm0, (%ecx,%eax)
addl $16, %eax
cmpl $480, %eax
jne .L3
addl $1024, %edx
cmpl $a+204804, %edx
jne .L2
```

```
.LC0:
.long 5
&.long 5
.long 5
.long 5
```

Loop Interchange

Consider the following C program:

```
for (i=0; i<120; i++)
{
for (j = 0; j < 200; j++)
a[i][j] = a[i+1][j] + 5;
}
```

- Compile the program for vectorization and examine the relevant dumps. Comment on the vector code and explain with respect to the source
- Is the program semantically equivalent to other two programs given? If YES, find the best vector code among the three, else give reasons for your answer.

Loop Interchange

Loop Interchange for Parallelization

Original Code

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

- Outer Loop - dependence on i, can not be parallelized
- Inner Loop - parallelizable, but synchronization barrier required

Total number of synchronisation's required = n

Loop Interchange

Loop Interchange for Parallelization

Original Code

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

After Interchange

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

- Outer Loop - parallelizable

Total number of Synchronization required = 1

Loop Distribution

Original Code

```
for (i=0; i<230; i++) {
   S1 : a[i+3] = a[i];
   S2 : b[i] = c[i];
}
```

- True dependence in S_1, no dependence in S_2
- Loop cannot be vectorized or parallelized, but S_2 can be vectorized and parallelized independently

Loop Distribution

Original Code

```
for (i=0; i<230; i++) {
   S1 : a[i+3] = a[i];
   S2 : b[i] = c[i];
}
```

- True dependence in S_1, no dependence in S_2
- Loop cannot be vectorized or parallelized, but S_2 can be vectorized and parallelized independently

Compile with

```
gcc -O2 -ftree-loop-distribution -fdump-tree-ldist
```

Loop Distribution

Control Flow Graph	Distributed Control Flow Graph

```
<bb 3>:
  D.1819 = i + 3;
  D.1820 = a[i];
  a[D.1819] = D.1820;
  D.1821 = c[i];
  b[i] = D.1821;
  i = i $+ 1$;

<bb 4>:
  if (i <= 229)
    goto <bb 3>;
  else
    goto <bb 5>;
```

```
<bb 6>:
  # i_12 = PHI <i_19(7), 0(2)>
  D.1855_13 = i_12 $+ 3$;
  D.1856_8 = a[i_12];
  a[D.1855_13] = D.1856_8;
  i_19 = i_12 $+ 1$;
  if (i_19 != 230)
    goto <bb 7>;

<bb 3>:
  # i_14 = PHI <i_6(4), 0(8)>
  D.1857_5 = c[i_14];
  b[i_14] = D.1857_5;
  i_6 = i_14 $+ 1$;
  if (i_6 != 230)
    goto <bb 4>;
```

Loop Distribution

After Distribution

```
for (i=0; i<230; i++)
    S1 : a[i+3] = a[i];
for (i=0; i<230; i++)
    S2 : b[i] = c[i];
```

- S_2 can now be independently parallelized or vectorized
- S_1 runs sequentially

Loop Fusion for Locality

Original Code

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i][j] = b[i];
for (k=0; k<n; k++)
    for (l=0; l<n; l++)
        b[k] = a[k][l];
```

- Large reuse distance for array a and b, high chances of cache miss
- If loops i and k are parallelized, 2 synchronisations required
- Outer loops i and k can be fused

Loop Fusion for Locality

Original Code

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i][j] = b[i];
for (k=0; k<n; k++)
    for (l=0; l<n; l++)
        b[k] = a[k][l];
```

Fused Code

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

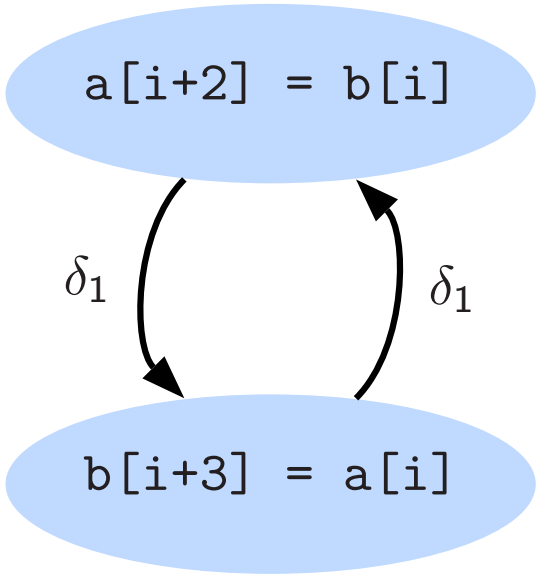
- Reduced reuse distance for array `a` and `b`, low chances of cache miss
- If outer loop `i` is parallelized, only 1 synchronization required
- Whether fusion of inner loops `j` and `l` is also possible? Reason your claim?

Loop Peeling

Peeling for Vectorization

Original Code

```
for (i=0; i<n; i++)
{
    S1: a[i+2] = b[i];
    S2: b[i+3] = a[i];
}
```



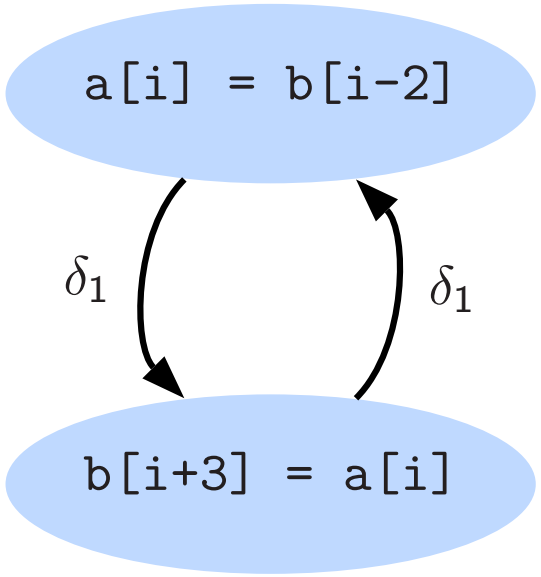
- Cyclic Dependence, both dependence distances are $<$ VF
- Cannot vectorize

Loop Peeling

Peeling for Vectorization

Transformed Code

```
for (i=0; i<2; i++)
    S2: b[i+3] = a[i];
for (i=2; i<n; i++) {
    S1: a[i] = b[i-2];
    S2: b[i+3] = a[i];
}
for (i=n; i<n+2; i++)
    S1: a[i] = b[i-2];
```



- Cyclic Dependence, dependence distance from S_2 to S_1 is $5 \geq$ VF
- Can vectorize

Loop Peeling

Peeling for Parallelization

Original Code

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

- dependence on i, can not be parallelized

Total number of synchronisations required = n

Loop Peeling

Peeling for Parallelization

Original Code

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

Transformed Code

```
c[1] = a[0];
for (i=1; i<n-1; i++) {
    S1: a[i] = b[i];
    S2: c[i+1] = a[i];
}
a[n-1] = b[n-1];
```

- Outer Loop parallelizable

Total number of synchronisations required = 1

Loop Alignment

```
int a[256], b[256];
int main ()
{
     int i;
     for (i=0; i<203; i++)
          a[i+2] = b[i+2];
}
```

Loop Alignment

```
int a[256], b[256];
int main ()
{
     int i;
     for (i=0; i<203; i++)
          a[i+2] = b[i+2];
}
```

```
7: misalign = 8 bytes of ref b[D.2302_4]
7: misalign = 8 bytes of ref a[D.2302_4]
```

Loop Alignment

Observing the final decision about alignment

```
7: Try peeling by 2
7: Alignment of access forced using peeling.
7: Peeling for alignment will be applied.

7: vectorization_factor = 4, niters = 203
   prologue iterations: 2
   epilogue iterations: 1
```

Loop Alignment

An aligned vectorized code can consist of three parts

- Peeled Prologue - Scalar code for alignment
- Vectorized body - Iterations that are vectorized
- Epilogue - Residual scalar iterations

Loop Alignment

Control Flow Graph

```
<bb 3>:
  D.1821 = i + 2;
  D.1822 = i + 2;
  D.1823 = b[D.1822];
  a[D.1821] = D.1823;
  i = i + 1;

<bb 4>:
  if (i <= 202)
    goto <bb 3>;
  else
    goto <bb 5>;
```

Vectorized Control Flow Graph

```
<bb 3>:
  # i_24 = PHI <0(2)>
  # ivtmp.4_15 = PHI <203(2)>
  # ivtmp.8_75 = PHI <0(2)>

<bb 17>:
  # i_16 = PHI <i_24(3), i_22(4)>
  # ivtmp.4_18 = PHI <ivtmp.4_15(3), ivtmp.4_23(4)>
  # ivtmp.8_35 = PHI <ivtmp.8_75(3), ivtmp.8_36(4)>
  D.2302_19 = i_16 + 2;
  D.2303_20 = b[D.2302_19];
  a[D.2302_19] = D.2303_20;
  i_22 = i_16 + 1;
  ivtmp.4_23 = ivtmp.4_18 - 1;
  ivtmp.8_36 = ivtmp.8_35 + 1;
  if (ivtmp.8_36 <= 1)
    goto <bb 4>;
```

2 Iterations of Prologue

Loop Alignment

Control Flow Graph

```
<bb 3>:
  D.1821 = i + 2;
  D.1822 = i + 2;
  D.1823 = b[D.1822];
  a[D.1821] = D.1823;
  i = i + 1;

<bb 4>:
  if (i <= 202)
    goto <bb 3>;
  else
    goto <bb 5>;
```

Vectorized Control Flow Graph

```
<bb 8>:
 # i_13 = PHI <i_6(13), i_26(7)>
 # ivtmp.4_12 = PHI <ivtmp.4_11(13), ivtmp.4_29(7)>
 # vect_pb.14_64=PHI <vect_pb.14_65(13),vect_pb.17_63(7)>
 # vect_pa.19_70=PHI <vect_pa.19_71(13),vect_pa.22_69(7)>
 # ivtmp.23_72 = PHI <ivtmp.23_73(13), 0(7)>
 D.2302_4 = i_13 + 2;
 vect_var_.18_66 = MEM[(int[256] *)vect_pb.14_64];
 D.2303_5 = b[D.2302_4];
 MEM[(int[256] *)vect_pa.19_70] = vect_var_.18_66;
 i_6 = i_13 + 1;
 ivtmp.4_11 = ivtmp.4_12 - 1;
 vect_pb.14_65 = vect_pb.14_64 + 16;
 vect_pa.19_71 = vect_pa.19_70 + 16;
 ivtmp.23_73 = ivtmp.23_72 + 1;
 if (ivtmp.23_73 < bnd.10_38)
   goto <bb 13>;
```

200 Iterations of Vector Code

Loop Alignment

Control Flow Graph

```
<bb 3>:
  D.1821 = i + 2;
  D.1822 = i + 2;
  D.1823 = b[D.1822];
  a[D.1821] = D.1823;
  i = i + 1;

<bb 4>:
  if (i <= 202)
    goto <bb 3>;
  else
    goto <bb 5>;
```

Vectorized Control Flow Graph

```
<bb 10>:
  # i_49 = PHI <tmp.12_59(9), i_26(6)>
  # ivtmp.4_52 = PHI <tmp.13_60(9), ivtmp.4_29(6)>

<bb 11>:
  # i_40 = PHI <i_46(12), i_49(10)>
  # ivtmp.4_42 = PHI <ivtmp.4_47(12), ivtmp.4_52(10)>
  D.2302_43 = i_40 + 2;
  D.2303_44 = b[D.2302_43];
  a[D.2302_43] = D.2303_44;
  i_46 = i_40 + 1;
  ivtmp.4_47 = ivtmp.4_42 - 1;
  if (ivtmp.4_47 != 0)
    goto <bb 12>;
```

1 Iteration of Epilogue

Loop Alignment

Strategy for Peeling

```
int a[256];
int main ()
{
     int i;
     for (i=4; i<253; i++)
         a[i-3] = a[i-3] + a[i+2];

}
```

Loop Alignment

Strategy for Peeling

```
int a[256];
int main ()
{
     int i;
     for (i=4; i<253; i++)
         a[i-3] = a[i-3] + a[i+2];

}
```

```
a[1] = a[1] + a[6]
```

Loop Alignment

Strategy for Peeling

```
int a[256];
int main ()
{
     int i;
     for (i=4; i<253; i++)
          a[i-3] = a[i-3] + a[i+2];

}
```

a[1] = a[1] + a[6]

Peel Factor = 3

Loop Alignment

Strategy for Peeling

```
int a[256];
int main ()
{
     int i;
     for (i=4; i<253; i++)
          a[i-3] = a[i-3] + a[i+2];

}
```

```
a[1] = a[1] + a[6]
```

Peel Factor = 3

Loop Alignment

Strategy for Peeling

```
int a[256];
int main ()
{
     int i;
     for (i=4; i<253; i++)
         a[i-3] = a[i-3] + a[i+2];

}
```

```
a[1] = a[1] + a[6]
```

Peel Factor = 2

Loop Alignment

Strategy for Peeling

```
int a[256];
int main ()
{
     int i;
     for (i=4; i<253; i++)
         a[i-3] = a[i-3] + a[i+2];

}
```

```
a[1] = a[1] + a[6]
```

Maximize alignment with minimal peel factor

Loop Alignment

Strategy for Peeling

```
int a[256];
int main ()
{
     int i;
     for (i=4; i<253; i++)
         a[i-3] = a[i-3] + a[i+2];

}
```

Peel the loop by 3

Parallelization and Vectorization in GCC : Conclusions

- Renewed interest in the area because of the current state of hardware
- Three major issues in automatic translation of sequential code
  1. detection of parallelism
  2. an intermediate representation for expressing parallelism detected
  3. transformations for enhancing parallelism
  4. generating good quality code for contemporary architectures
- Throws up challenging problems at theoretical and engineering levels
- Need to learn from successful manual translation efforts
- Outline of the GCC framework
  1. under evolution : attempts to address the first three issues
  2. a long way to go to catch up with the success of GCC achieved for sequential machines