5. Performance of Parallel Architectures

What is the gain from using a parallel architecture?

- Speedup
- Efficiency
- Scaleup
- MIMD vs. SIMD

**Speedup**

How much faster will a program be on a parallel computer system versus a sequential one?

\[
S_N = \frac{T_1}{T_N}
\]

**Efficiency**

How effective does a parallel program use the parallel resources?

\[
E_N = \frac{S_N}{N}
\]

Value range: \(1/N \leq E_N \leq 1\)

**Scaleup**

For a number of problems, we do not want to use a parallel architecture to compute the same program faster, but to calculate a scaled-up version of the program in the same time.

E.g.: Weather Forecast

Rather than calculating the same weather forecast program in 1 hour than in 10 hours, it makes more sense to run the forecast program at a higher precision during the full 10 hours, in order to obtain a better result.
Scaleup

Definition for different-size variations of a program:

\[ T_k(m) \quad \text{Execution time of a program with a problem size } m \text{ on } k \text{ processors.} \]

The scaleup of a problem of size \( n \) on \( k \) processors is compared to a smaller problem of size \( m \) \((m<n)\) on 1 processor:

If \( T_1(m) = T_k(n) \) (i.e. execution time for the small program on 1 processor is equal to the time of the large program on \( k \) processors. ) then the Scaleup is defined as:

\[ SC_k = \frac{n}{m} \]

It is always true: \( SC_k \leq k \)

Speedup Calculation

Measuring the achieved speedup comes somewhat late, because we have already:

- Bought an expensive parallel computer system
- Developed a parallel version of our software

Therefore, it would be much better to have a theoretical model, so we can at least estimate the speedup to be achieved in advance.

Note: These models are naturally a simplification of reality, so they do not take into account e.g.:

- Differences between CPUs
- Communication time and other (bookkeeping) overhead time
- Etc ...

Speedup Calculation

Theoretical speedup calculation by Amdahl 1967, here in simplified form:

Definitions for fixed Program A:

- \( P_c \) Maximum parallelization rate (of Program A with parallelism model C)
- \( T_k \) Execution time of Program A with maximum parallelism rate \( P_c \geq k \) on a system with \( k \) processors
- \( N \) Number of processors of the parallel computer system
- \( f \) Sequential program component Percentage part of operations, that cannot be executed in parallel but only sequentially.

Then the following is true for a parallel system with \( N \) processors:

\[ T_N = f \cdot T_1 + (1-f) \cdot \frac{T_1}{N} \]

Consequently, the calculated (theoretical) speedup is:

\[ S_N = \frac{T_1}{T_N} = \frac{N}{1 + f \cdot (N-1)} \]

From \( 0 \leq f \leq 1 \) follows: \( 1 \leq S_N \leq N \). The speedup can never be greater than the number of processors!
Speedup Example

a) 1,000 processors,
Program with maximum parallelization rate of 1,000,
0.1% of the program has to be executed sequentially (e.g. input/output),
i.e. \( f = 0.001 \)

Speedup \( S_{1000} = \frac{1000}{1 + 999/1000} = 500 \)

Only half of possible peak-performance is reached!

Speedup Example

b) 1,000 processors,
Program with maximum parallelization rate of 1,000,
1% of program has to be executed sequentially,
i.e. \( f = 0.01 \)

Speedup \( S_{1000} = \frac{1000}{1 + 999/100} = 91 \)

Only 10% of the total processor capability is utilized, and this with only a small sequential component!!

Maximum Speedup

Maximum theoretical speedup (independent of the number of processors):

\[
\lim_{N \to \infty} S_N(f) = \frac{N}{1 + f \times (N-1)} = \frac{1}{f}
\]

An MIMD-program with 1% scalar component can never achieve a larger speed-up than 100 – no matter if 100, 100-, or 1 million processors are used!
Scaleup Example: Reduction

**Question:** How many data elements can I add in n time steps on p processors?

**Answer:** Because of the tree structure, we need \( \log_2 p \) time steps to combine the results. Any additional time steps can be used to add data elements locally in each processor.

So we can add \((n+1 - \log_2 p)\) data elements in each PE.

Overall: \( p \times (n+1 - \log_2 p) \)

1 PE adds in 5 time steps 6 values \( \Rightarrow \) \( SC_1 = 1 \) (accord. to definition)
2 PEs add in 5 time steps 10 values \( \Rightarrow \) \( SC_2 = 1.7 \)
4 PEs add in 5 time steps 16 values \( \Rightarrow \) \( SC_4 = 2.7 \)
8 PEs add in 5 time steps 24 values \( \Rightarrow \) \( SC_8 = 4.0 \)
16 PEs add in 5 time steps 32 values \( \Rightarrow \) \( SC_{16} = 5.3 \)

**Scaleup Example: Reduction**

In General:

1 PE adds in n time steps \( n+1 \) data elements
p PEs add in n time steps \( p \times (n+1 - \log_2 p) \) data elements, provided \( \log_2 p \leq n \)

\[ SC_p = \frac{n + 1 - \log_2 p \times p}{n + 1} \]

(Where \( n \) is a constant)

**MIMD vs. SIMD**

Is there a principle difference in performance data between MIMD and SIMD?

Not really, however, consider the following (Bräunl, 1991):

**Definition:**

\( f_A \) denotes the sequential component in relation to the number of elementary operations of a program

\( f_B \) denotes the sequential component in relation to the execution time of the operations of a program.
**MIMD vs. SIMD**

by Bräunl 1991:

\[ f_A = \frac{f_B}{N \cdot (1 - f_B) + f_B} \]

\[ f_B = \frac{1}{2} \]

---

**Example:**

a) 1,000 PEs,

Program with maximum parallelization rate of 1,000, 0.1% of the instructions in SIMD-Program are scalar, i.e. \( f = 0.001 \)

Speedup \( S_{1000} = 0.001 + (1–0.001) \cdot 1,000 \approx 999 \)

**Maximum peak-performance is approached!!**

---

**MIMD vs. SIMD**

Alternative Speedup-Definition (Bräunl 1991):

\( f_B \) Percentage component of the scalar instructions compared to the total number of instructions or:

Percentage component of computing time of sequential operations compared to the total computing time of the parallel program.

Computation for the sequential execution (calculating backwards):

\[ T_1 = f_B \cdot T_N + (1–f_B) \cdot N \cdot T_N \]

Speedup-factor for fixed \( N \):

\[ S_N = \frac{T_1}{T_N} = f_B + (1–f_B) \cdot N \]

---

**Example:**

a) 1,000 PEs,

Program with maximum parallelization rate of 1,000, 0.1% of the instructions in SIMD-Program are scalar, i.e. \( f = 0.001 \)

Calculation according to Amdahl:

\[ f_A = \frac{f_B}{N \cdot (1 - f_B) + f_B} = \frac{0.001}{1000 \cdot (1 - 0.001) + 0.001} = 10^{-6} \]

Then, according to Amdahl:

\[ S_{1000} = \frac{1000}{1 + 10^{-6} \cdot (1000 - 1)} \approx 999 \]
MIMD vs. SIMD

b) 1,000 PEs,
Program with maximum parallelization rate of 1,000
10% of the instructions of the SIMD-Program are scalar, i.e. \( f = 0.1 \)

\[
\text{Speedup } S_{1000} = 0.1 + 0.9 \times 1000 \approx 900
\]

Despite a considerable sequential component still over 90% of peak-performance is reached.

MIMD vs. SIMD

Compare to original formula by Amdahl:

\[
T_i = f_B \times T_N + (1-f_B) \times \frac{N \times T_N}{i}
\]

\[
S_i = \frac{T_i}{T_f}
\]

Comes to the same result!

MIMD vs. SIMD

What have we learned here?

- Two ways of viewing things
- We cannot simply fix a linear percentage to a program
- Looking at different program sizes and processor numbers is mainly academic …

Performance Analysis MIMD

```
1 PROCEDURE worker(i);
2 VAR r,t: INTEGER;
3 BEGIN
4 t:=0;
5 FOR s:=1 TO 1:50 DO
6 t:=t+8;
7 END;
8 mon:put(t);
9 END PROCESS worker;
10 MONITOR mon;
11 VAR a: INTEGER;
12 ENTRY put(e: INTEGER);
13 BEGIN
14 r:=r+e;
15 END put;
16 BEGIN (* monitor init *)
17 MONITOR mon;
18 (* init main program *)
19 FOR i:=1 TO 100 DO START(worker(i)); END;
```

Bräunl 2004
Performance Analysis MIMD

sum of main: 300 units sequentially plus 15,800 units parallel

\[
T_1 = 300 + 15,800 = 16,100
\]

\[
T_{100} = 300 + \frac{15,800}{100} = 458
\]

\[
S_{100} = \frac{16,100}{458} \approx 35.2
\]

\[
E_{100} = \frac{35.2}{100} \approx 35
\]

Performance Analysis SIMD

Case A: Number of physical PEs = Number of virtual PEs

\[
T_1 = 1,000 + 5,500 \times 10,000 = 55,001,000
\]

\[
T_{10,000} = 1,000 + 5,500 = 6,500
\]

Speedup

\[
S_{10,000} = \frac{T_1}{T_{10,000}} = 8,462
\]

Efficiency

\[
E_{10,000} = \frac{T_{10,000}}{10,000} \approx 85\%
\]

Case B: Number of physical PEs < Number of virtual PEs

\[
T_1 \text{ remains unchanged}
\]

\[
T_{100} = 1,000 + 5,500 \times 100 + 200 = 551,200
\]

\[
\text{Speedup}
\]

\[
S_{100} = \frac{T_{100}}{T_{100}} \approx 99.8
\]

Efficiency

\[
E_{100} = \frac{T_{100}}{100} \approx 100\%
\]
Performance Analysis SIMD

**Case C:** Number of physical PEs > Number of virtual PEs

\[
T_1 \quad \text{remains unchanged}
\]

\[
T_{10^6} = T_{10,000} = 6.500
\]

\[
\text{Speedup} \quad \quad S_{10^6} = \frac{T_1}{T_{1,000,000}} = S_{10,000} \approx 8.462
\]

\[
\text{Efficiency} \quad \quad E_{100} = \frac{T_{1,000,000}}{1,000,000} \approx 0.85\% \quad (\text{i.e.} \ < 1\% !)
\]

Analysis of Performance Data

- Performance data is always application dependent
- Speedup refers to a single processor in a parallel computer, however, this processor is potentially considerably less powerful than a processor in a sequential computer
- Processor utilization is top priority on an MIMD-system, but not on an SIMD-system
- Peak-Performance-Data only has limited meaning
- Benchmarks should cover a wide variety of applications and should be relevant for the proposed application (e.g. NAS Parallel Benchmark "Numerical Aerodynamic Simulation")

Analysis of Performance Data

- Architecture-independent programming languages cannot utilize all capabilities of a parallel system. Warning: the compiler is automatically included in the overall evaluation!
- Assembler-Programming (or low level programming) often results in better performance
- Including input/output in the evaluation can cause totally different performance results (bottleneck due to sequential I/O)
- When comparing sequential-parallel (or SIMD-MIMD) one must potentially compare **different algorithms** (whatever is best suited for the particular computer)

6. Automatic Parallelization/Vectorization

- Software development is expensive
- Parallel software development is **really** expensive
- For many applications, companies have **existing** software solutions
- **Idea:** Just re-compile source code with automatic parallelization (MIMD) or automatic vectorization (SIMD)
6.1 Data Dependence

(by Michael Wolfe)

Data dependences can exist between the instructions of a (sequential) program.

Auxiliary definitions:
Input Set of instruction S:
IN(S) = "set of all input elements, whose value is read by S"

Output Set of instruction S:
OUT(S) = "set of all output elements, whose value is changed by S"
Data Dependence

Example:

```
for i:=1 to 5 do
  S: X[i]:=A[i+1]*B
end;
```

\[ \text{IN}(S) = \{A_2, A_3, A_4, A_5, A_6, B\} \]

\[ \text{OUT}(S) = \{X_1, X_2, X_3, X_4, X_5\} \]

Definition: Execution Sequence

If statement \( S \) is enclosed in a loop with index \( i \), then \( S_i \) denotes the instance \( S \) during loop \( i=i' \).

Definition:

\( S_1 \Theta S_2 \iff \) An instance of \( S_1 \) can be executed during program execution before instance \( S_2 \).

\( S_1i' \Theta S_2i'' \iff S_1, S_2 \) are both enclosed in a loop and \( S_1i' \) can be executed before \( S_2i'' \).

Assumption:

a) Loop-increment is always 1
b) For simplicity only assignment instructions are considered

Data Dependence

Definition: Indirect Data Dependence

(1) \[ \exists x: x \in \text{OUT}(S_1) \land x \in \text{IN}(S_2) \land S_1 \Theta S_2 \land \neg \exists k: \( S_1 \Theta S_k \Theta S_2 \land x \in \text{OUT}(S_k) \) \]
\[ \iff S_2 \text{ is flow-dependent on } S_1: \quad S_1 \delta S_2 \]

(2) \[ \exists x: x \in \text{IN}(S_1) \land x \in \text{OUT}(S_2) \land S_1 \Theta S_2 \land \neg \exists k: \( S_1 \Theta S_k \Theta S_2 \land x \in \text{OUT}(S_k) \) \]
\[ \iff S_2 \text{ is anti-dependent on } S_1: \quad S_1 \bar{\delta} S_2 \]

(3) \[ \exists x: x \in \text{OUT}(S_1) \land x \in \text{OUT}(S_2) \land S_1 \Theta S_2 \land \neg \exists k: \( S_1 \Theta S_k \Theta S_2 \land x \in \text{OUT}(S_k) \) \]
\[ \iff S_2 \text{ is output-dependent on } S_1: \quad S_1 \delta^o S_2 \]

If neither (1) nor (2) nor (3) holds, then \( S_1 \) and \( S_2 \) are data-independent.
Data Dependence

Example:
S1 : A := B + D;
S2 : C := A * 3;
S3 : A := A + C;
S4 : E := A / 2;

Dependences:
S1 δ S2    (due to A)
S1 δ S3    (due to A)
S2 δ S3    (due to C)
S3 δ S4    (due to A)
S2 δ S3    (due to A)
S1 δ° S3   (due to A)

Indirect Data Dependence

Example:
for i:=1 to n do
for j:=2 to m do
S1: A[i,j] := B[i,j];
S2: C[i,j] := A[i,j-1];
end;
end;

Data dependence: S1 δ S2 (because of A)
Direction:
for the enclosing loop with index i: 2 = 2
for the inner loop with index j: 2 < 3
This means that between S1 and S2 exists the following directed data dependence:
S1 δ(=,<) S2

Data Dependence Direction

Assume S1 and S2 are embedded in loops with indices $i_1, ..., i_d$
If there exist two specific loop instances $I' = (i_1', ..., i_d')$ and $I'' = (i_1'', ..., i_d'')$
for the loop indices $i_1, ..., i_d$ such that the following holds for the appropriate instances of S1 and S2:
$S_1 \; i_1'...i_d' \; \delta \; S_2 \; i_1''...i_d''$
and if further the following relation is valid for the index vectors (loop instances):
$I' \; \psi \; I''$
(that is: $\psi=(\psi_1, ..., \psi_d)$ where $\psi_r \in \{<,=,>,\geq,\leq,?,\}$
["?" represents an unknown relation]
$\psi_r$ $i_r' \; \psi$ $i_r''$
$\psi_r$ $\psi_2 \; i_2''$
... $\psi_d \; \psi_d \; i_d''$
then the following definition holds:
$\Leftrightarrow$: $S_2$ is data-dependent with direction $\psi$ on $S_1$: $S_1 \; \delta \; S_2$
Data Dependence

Memorizing Rule
If for a data dependency the index of a variable is i for write access, and for read access it is:

\[ i+1 \]
then the old value of the variable is read

\[ i-1 \]
then the new value of the variable is read

6.2 Loop Vectorization

Rules

a) If a data dependency \( S_x \delta^* S_y \) exists with the loop to be vectorized, then the vectorized code must execute instruction \( S_x \) before instruction \( S_y \) (if necessary by changing the execution sequence).

b) Data dependencies with direction "<" or "">
   loop do not have to be taken into account.

c) If multiple data dependencies prevent a consistent instruction sequence, then the loop cannot be vectorized directly.

Example:

\[ \text{for } i:=1 \text{ to } n \text{ do} \]
\[ S_1: \quad A[i]:= B[i] + C[i]; \]
\[ S_2: \quad D[i]:= A[i+1] + 1; \]
\[ S_3: \quad C[i]:= D[i]; \]
\[ \text{end} \]

Data Dependences:

\[ S_1 \delta (-) S_3 \text{ (due to } C) \]
\[ \Rightarrow \quad S_1 \text{ before } S_3 \]
\[ S_2 \delta (-) S_1 \text{ (due to } A) \]
\[ \Rightarrow \quad S_2 \text{ before } S_1 \]
\[ S_2 \delta (-) S_3 \text{ (due to } D) \]
\[ \Rightarrow \quad S_2 \text{ before } S_3 \]

Solution in Fortran-90 notation:
(after re-arrangement of instructions)

\[ S_2: \quad D(1:N) = A(2:N+1) + 1 \]
\[ S_1: \quad A(1:N) = B(1:N) + C(1:N) \]
\[ S_3: \quad C(1:N) = D(1:N) \]
6.3 Loop Parallelization

So far: Vectorization of loop for SIMD or Vector system
Now: Parallelization of loop for MIMD system

Parallelization of a Loop
"doacross": allocate individual loop-iterations to different processors. Additional synchronization is required depending on data dependences.

Loop Parallelization Rules

a) Data dependencies with direction "=" for the loop to be parallelized do not need to be synchronized.
b) Data dependencies with direction "<" or ">" in potentially existing loops enclosing the loop to be parallelized can be ignored.
c) Potentially existing inner loops do not need to be considered when parallelizing; they are completely taken over into the parallelizing code.
d) Every other data dependency has to be synchronized between processes via their own arrays of semaphores.
e) To increase efficiency the execution sequence of instructions may be changed within the limits of the data dependencies:
   However, if a data dependency $S_x \delta^*(=) S_y$ exists within the loop, then the parallelizing code must execute instruction $S_x$ before instruction $S_y$.

Example of Parallelization of a Loop (case a):

```c
for i:=1 to n do
  S_1: A[i]:= C[i];
  S_2: B[i]:= A[i];
end;
```

Data dependency $S_1 \delta (=) S_2$ (due to $A[i]$). In the same loop iteration, hence synchronization is not required.

```c
doacross i:=1 to n do
  S_1: A[i]:= C[i];
  S_2: B[i]:= A[i];
enddoacross;
```
Loop Parallelization

Example of Parallelization of a Loop (case b):
The inner loop is to be parallelized:

```plaintext
for i:=1 to n do
  for j:=1 to m do
    S_1: A[i,j]:= C[i,j];
    S_2: B[i,j]:= A[i-1,j-1];
  end;
end;
```

Data dependency: \( S_1 \delta (.<,) S_2 \) (due to \( A[i,j] \))

Synchronization required: **NONE** due to dependency direction "<"

Parallelized Loop:

```plaintext
for i:=1 to n do
  doacross j:=1 to m do
    S_1: A[i,j]:= C[i,j];
    S_2: B[i,j]:= A[i-1,j-1];
  endacross;
end;
```

Example of Parallelization of a Loop (case c):
The outer loop is to be parallelized:

```plaintext
for i:=1 to n do
  for j:=1 to n do
    S_1: A[i,j]:= B[i,j];
    S_2: B[i,j]:= A[i,j-1];
  end;
end;
```

Determination of all data dependencies:

\[ S_1 \delta (.<,) S_2 \] (due to \( A[i,j] \))

\[ S_1 \delta (.<,) S_2 \] (due to \( A[i,j] \)) \[ \Rightarrow \] no synchronization requirements
**Loop Parallelization**

**Example (cases d+e):**

by Wolfe in [Hwang, DeGroot 89]

```
for i:= 1 to n do
  S1: A[i] := B[i] + C[i];
  S2: D[i] := A[i] + E[i-1];
  S3: E[i] := E[i] + 2 * B[i];
  S4: F[i] := E[i] + 1;
end;
```

Data Dependences:

- $S_1 \delta(=) S_2$ (due to $A[i]$) ← no synch. required
- $S_1 \delta(=) S_4$ (due to $E[i]$) ← no synch. required
- $S_1 \delta(<) S_2$ (due to $E[i]$) ← synch. required because of direction “<“

---

**Loop Parallelization with Synchronization**

```
var sync: array [1..n] of semaphore[0];

doacross i := 1 to n do
  S1: a[i] := b[i] + c[i];
  if i>1 then P(sync[i-1]) end;
  S2: d[i] := a[i] + e[i-1];
  V(sync[i]);
  S3: e[i] := e[i] + 2 * b[i];
  S4: f[i] := e[i] + 1;
endoacross;
```

More efficient solution:

```
var sync: array [1..n] of semaphore[0];

doacross i := 1 to n do
  S1: a[i] := b[i] + c[i];
  V(sync[i]);
  S2: d[i] := a[i] + c[i];
  if i>1 then P(sync[i-1]) end;
  S3: e[i] := e[i] + 2 * b[i];
  S4: f[i] := e[i] + 1;
endoacross;
```
Loop Parallelization with Synchronization

More efficient solution:

<table>
<thead>
<tr>
<th>PE</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>S3, S1, S4, S2</td>
</tr>
<tr>
<td>P2</td>
<td>S3, S1, S4, S2</td>
</tr>
<tr>
<td>P3</td>
<td>S3, S1, S4, S2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Pn</td>
<td>S3, S1, S4, S2</td>
</tr>
</tbody>
</table>

6.4 Complex Data Dependences

Circular dependency:

\[ \text{for } i := 1 \text{ to } n \text{ do} \]
\[ S_1: A[i] := B[i]; \]
\[ S_2: B[i] := A[i+1]; \]
\[ \text{end;} \]

\[ S_2 \circlearrowleft (\neq) S_1 \quad \text{(due to A)} \Rightarrow S_2 \text{ before } S_1 \]
\[ S_1 \circlearrowright (\neq) S_2 \quad \text{(due to B)} \Rightarrow S_1 \text{ before } S_2 \]

No re-arrangement possible!

Complex Data Dependences

Breaking circular dependency via auxiliary variable:

\[ \text{for } i := 1 \text{ to } n \text{ do} \]
\[ S_H: \text{Aux}[i] := B[i]; \]
\[ S_1: A[i] := \text{Aux}[i]; \]
\[ S_2: B[i] := A[i+1]; \]
\[ \text{end;} \]

The data dependencies change as follows:

\[ S_A \circlearrowleft (\neq) S_1 \quad \text{(due to Aux)} \Rightarrow S_A \text{ before } S_1 \]
\[ S_A \circlearrowright (\neq) S_2 \quad \text{(due to A)} \Rightarrow S_A \text{ before } S_2 \]
\[ S_2 \circlearrowleft (\neq) S_1 \quad \text{(due to B)} \Rightarrow S_2 \text{ before } S_1 \]

Vectorization is now possible!

Complex Data Dependences

Solution

\[ S_H: \text{Aux}(1:N) = B(1:N); \]
\[ S_2: B(1:N) = A(2:N+1); \]
\[ S_1: A(1:N) = \text{Aux}(1:N); \]
Complex Data Dependences

Loop Interchanging

- Swapping of inner and outer loops
- E.g. if data dependencies require it
- E.g. if vectorizing the outer loop promises better parallelism
- Is possible if there are no data dependencies in direction (<,>)

Loop Interchanging Example

Interchanging inner and outer loop:

\[
\begin{align*}
&\text{for } j=1, n \\
&\text{for } i=1, n \\
&\text{end;} \quad (* \quad i \quad *) \\
&\text{end;} \quad (* \quad j \quad *)
\end{align*}
\]

Data dependences for changed program:

\[
\begin{align*}
S_1 \quad \delta_{(<,)} \quad S_1 \quad \text{(due to } A_{i,j-1}) \quad \\ S_1 \quad \delta_{(>,)} \quad S_1 \quad \text{(due to } A_{i,j+1})
\end{align*}
\]

\[\Rightarrow \text{ no restriction, since surrounding loop has direction } <^\ast\]

Vectorization possible!

Loop Interchanging Example

by Wolfe in [Hwang, DeGroot 89]

\[
\begin{align*}
&\text{for } i=1 \text{ to } n \\
&\text{for } j=1 \text{ to } n \\
&\text{end;} \quad (* \quad j \quad *) \\
&\text{end;} \quad (* \quad i \quad *)
\end{align*}
\]

The data dependences are:

\[
\begin{align*}
S_1 \quad \delta_{(<,)} \quad S_1 \quad \text{(due to } A_{i,j-1}) \quad \Rightarrow \quad S_1 \quad \text{before } S_1 \\
S_1 \quad \delta_{(>,)} \quad S_1 \quad \text{(due to } A_{i,j+1}) \quad \Rightarrow \quad S_1 \quad \text{before } S_1
\end{align*}
\]

Cannot be vectorized directly!
Loop Interchanging

Things get more complex if the loop limits are dependable:

```plaintext
for i:=1 to n do
    for j:=1 to i do
        S1: A[i,j] := A[j,i];
        end; (* j *)
    end; (* i *)
end
```

Here the index `j` of the inner loop depends on the value of the index `i` of the outer loop. During the loop swap these limits need to be adjusted appropriately:

```plaintext
for j:=1 to n do
    for i:=j to n do
        S1: A[i,j] := A[j,i];
        end; (* i *)
    end; (* j *)
end
```

Before:

![Before Diagram]

After:

![After Diagram]

Vectorized Program:

```plaintext
do j=1,n
    A(j:n,j) = A(j,j:n);
end do
```

Complex Dependences

**Double-indexed Access:**

```plaintext
for i:= 1 to n do
    S1: A[C[i]] := B[i]
    end;
```

It is not possible to resolve this conflict!

**Data dependencies within an instruction:**

```plaintext
for i:= 1 to n do
    S1: A[i] := A[i+1]
    end;
```

During the vectorization this is not a dependency:

```plaintext
S1: A(1:n) = A(2:n+1)
```