

⋮

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 ?

T_k	Execution time of a certain program on a system with k processors, e.g. obtained from measurements
N	Number of processors of the parallel computer system
S_k	Speedup of a certain program on k processors

$$S_N = \frac{T_1}{T_N}$$

⋮

Efficiency

How effective does a parallel program use the parallel resources ?

$$E_N = \frac{S_N}{N}$$

Value range: $1/N \leq E_N \leq 1$

Utilization

- The larger the sequential component of an MIMD-program is, the more dramatically the processor utilization rate drops
- As the utilization drops, so does the parallel speedup compared to a sequential computer system

⋮

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:

$T_k(m)$ Execution time of a program with a problem size m on k 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:

$$SC_k = n/m$$

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.

• • • • •

⋮

Speedup Calculation

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

$$T_N = f * T_1 + (1 - f) * T_1/N$$

Consequently, the calculated (theoretical) speedup is:

$$S_N = \frac{T_1}{T_N} = \frac{N}{1 + f * (N - 1)}$$

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. $f = 0.001$

$$\text{Speedup } S_{1000} = 1000 / (1 + 999/1000) \approx 500$$

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. $f = 0.01$

$$\text{Speedup } S_{1000} = 1000 / (1 + 999/100) \approx 91$$

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

⋮

Maximum Speedup

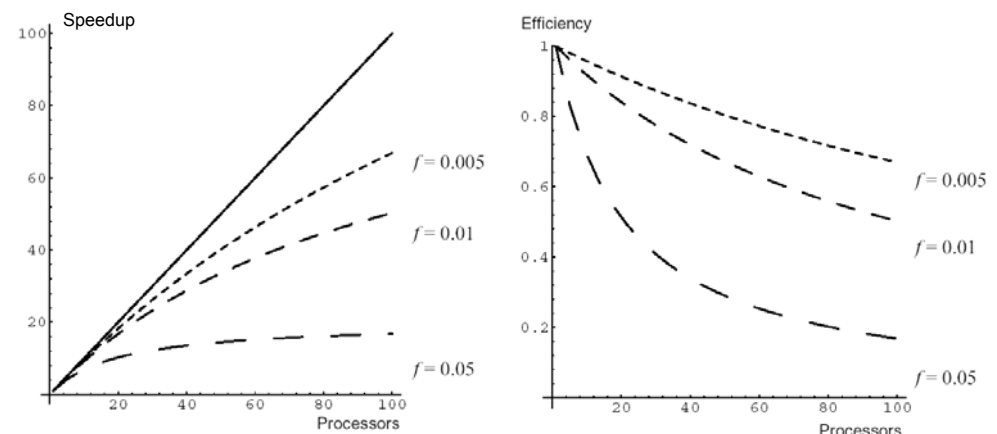
Maximum theoretical speedup
 (independent of the number of processors):

$$\lim_{N \rightarrow \infty} S_N(f) = \frac{N}{1 + f * (N - 1)} = \frac{1}{f}$$

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 !

⋮

Maximum Speedup and Efficiency



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: $p * (n+1 - \log_2 p)$

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:

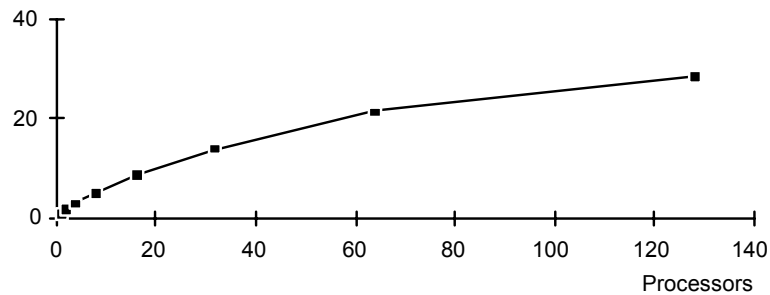
1 PE adds in n time steps $n+1$ data elements
 p PEs add in n time steps $p * (n+1 - \log_2 p)$ data elements, provided $\log_2 p \leq n$

$$\Rightarrow SC_p = \frac{n+1 - \log_2 p}{n+1} * p$$

$$= p - \frac{p * \log_2 p}{n+1} \quad (\text{where } n \text{ is a constant})$$

Scaleup Example: Reduction

Scaleup (for $n = 8$)



MIMD vs. SIMD

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

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

Definition:

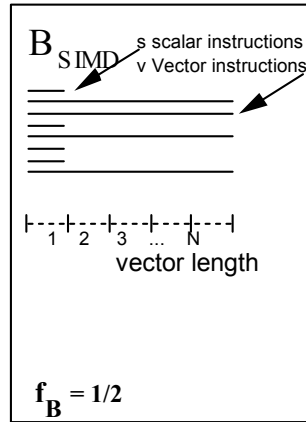
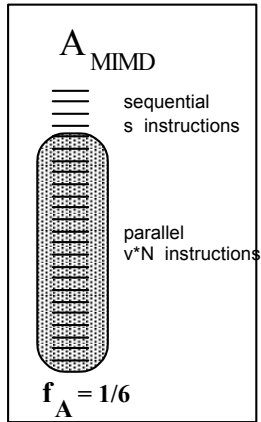
f_A denotes the sequential component in relation to *the number of elementary operations* of a program

f_B denotes the sequential component in relation to *the execution time of the operations* of a program.

MIMD vs. SIMD

$$f_A = \frac{f_B}{N * (1 - f_B) + f_B}$$

by Bräunl 1991:



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*):

$$T_1 = f_B * T_N + (1 - f_B) * N * T_N$$

Speedup-factor for fixed N:

$$S_N = T_1 / T_N = f_B + (1 - f_B) * N$$

MIMD vs. SIMD

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 $S_{1000} = 0.001 + (1 - 0.001) * 1,000 \approx 999$

Maximum peak-performance is approached!!

MIMD vs. SIMD

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:

$$f_A = \frac{f_B}{N * (1 - f_B) + f_B} = \frac{0.001}{1000 * (1 - 0.001) + 0.001} = 10^{-6}$$

Then, according to Amdahl:

$$S_{1000} = \frac{1000}{1 + 10^{-6} * (1000 - 1)} \approx 999$$

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. $f = 0.1$

$$\text{Speedup } S_{1000} = 0.1 + 0.9 * 1000 \approx 900$$

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

MIMD vs. SIMD

Compare to original formula by Amdahl:

$$T_i = f_B * T_N + (1-f_B) * \frac{N * T_N}{i}$$

$$S_i = \frac{T_1}{T_i}$$

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

line	cost
monitor	
10-11	0 units
16-17	0 units
put	
12-13	0 units (assuming there are no waiting periods when entering)
14	1 unit
15	1 unit for returning from subroutine
sum of put:	2 units
worker	
1-3	0 units
4	1 unit
5	51 units for incrementing loop counter
6	51 units, one for each loop iteration
7	51 units for compare of loop counter and branching
8	3 units (1 for entry call plus 2 for entry execution, see above)
9	1 unit for process termination
sum of worker:	158 units
main	
18	100 units for incrementing loop counter 100 units for compare of loop counter and branching 100 units for starting processes plus <i>parallelizable</i> : 100 × process body (100*158 units)

Performance Analysis MIMD

```

1 PROCESS worker(i);
2 VAR s,t: INTEGER;
3 BEGIN
4   t:=0;
5   FOR s:=1 TO i+50 DO
6     t:=t+s;
7   END;
8   mon:put(t);
9 END PROCESS worker;

```

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

$$T_1 = 300 + 15,800 = 16,100$$

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

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

$$E_{100} = \frac{35.2}{100} \approx 35\%$$

```

10 MONITOR mon;
11 VAR s: INTEGER;
12 ENTRY put(e: INTEGER);
13 BEGIN
14   s := s+e;
15 END put;
16 BEGIN (* moitor init *)
17 END MONITOR mon;
18 FOR i:=1 TO 100 DO START(worker(i)); E

```

Bräunl 2004

Performance Analysis SIMD

```

1 SYSTEM xyz;
2 CONFIGURATION grid[1..100],[1..100];
3 CONNECTION right: grid[i,j] <-> grid[i,j+1].left;
4 SCALAR s: INTEGER;
5 VECTOR v: INTEGER;
6 BEGIN
7   FOR s:=1 TO 500 DO
8     PARALLEL
9       v:=2*v;
10      PROPAGATE.right(v);
11    ENDPARALLEL;
12  END;
13 END xyz.

```

line	cost
1-6	0 time units
7	500 time units for incrementing loop counter (seq.)
8	0 time units
9	500 time units, one for each loop iteration (par.)
10	5000 time units (a data exchange is assumed to cost 10 time units) (par.)
11	0 time units
12	500 time units for compare of loop counter and branching (seq.)
13	0 time units
sum :	1.000 time units for sequential statements
plus	5.500 time units for parallel statements

Bräunl 2004

26

Performance Analysis SIMD

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

$$T_1 = 1,000 + 5,500 * 10,000 = 55,001,000$$

$$T_{10,000} = 1,000 + 5,500 = 6,500$$

Speedup

$$S_{10,000} = \frac{T_1}{T_{10,000}} \approx 8,462$$

Efficiency

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

Bräunl 2004

27

Performance Analysis SIMD

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

T_1 remains unchanged

$$T_{100} = 1,000 + 5,500 * 100 + 200 = 551,200$$

Speedup

$$S_{100} = \frac{T_1}{T_{100}} \approx 99,8$$

Efficiency

$$E_{100} = \frac{T_{100}}{100} \approx 100\%$$

Bräunl 2004

28



Performance Analysis SIMD

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

T_1 remains unchanged

$$T_{106} = T_{10,000} = 6.500$$

Speedup

$$S_{106} = \frac{T_1}{T_{1,000,000}} = S_{10,000} \approx 8,462$$

Efficiency

$$E_{106} = \frac{T_{1,000,000}}{1,000,000} \approx 0.85\% \text{ (i.e. } < 1\% \text{ !)}$$



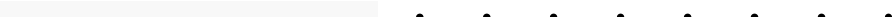
Analysis of Performance Data

- Performance data is always application dependent
- Speedup refers to a single processor in a parallel computer, however, this processor is potentially considerable 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)



⋮

Automatic Vectorization of for-loop

Sequential:

```

for i:=1 to n do
  A[i] := B[i] + C[i];  (* statement in loop .. *)
  D[i] := A[i] * 5;    (* statement in loop .. *)
end;

```

Parallel (Fortran 90):

```

A(1:n) = B(1:n) + C(1:n) (* .. becomes vector statement *)
D(1:n) = A(1:n) * 5      (* .. becomes vector statement *)

```



⋮

Automatic Vectorization of for-loop

Sequential:

```

for i:=1 to n do
  A[i] := B[i] + C[i];
  D[i] := A[i+1] * 5;  (* uses old value of A *)
end;

```

Parallel (Fortran 90):

```

A(1:n) = B(1:n) + C(1:n)
D(1:n) = A(2:n+1) * 5  (* incorrect: uses new value of A *)

```



⋮

Automatic Vectorization of for-loop

Sequential:

```

for i:=1 to n do
  A[i] := B[i] + C[i];
  D[i] := A[i+1] * 5;  (* uses old value of A *)
end;

```

Parallel (Fortran 90):

```

D(1:n) = A(2:n+1) * 5 (* correct: change execution order *)
A(1:n) = B(1:n) + C(1:n)

```



⋮

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;

```

$$IN(S) = \{A2, A3, A4, A5, A6, B\}$$

$$OUT(S) = \{X1, X2, X3, X4, X5\}$$

Data Dependence

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

(1) $\exists x: x \in OUT(S_1) \wedge x \in IN(S_2) \wedge S_1 \Theta S_2 \wedge \neg \exists k: (S_1 \Theta S_k \Theta S_2 \wedge x \in OUT(S_k))$
 $\iff S_2$ is **flow-dependent** on S_1 : $S_1 \delta S_2$

S_2 uses the value x , which was calculated in S_1

(2) $\exists x: x \in IN(S_1) \wedge x \in OUT(S_2) \wedge S_1 \Theta S_2 \wedge \neg \exists k: (S_1 \Theta S_k \Theta S_2 \wedge x \in OUT(S_k))$
 $\iff S_2$ is **anti-dependent** on S_1 : $S_1 \bar{\delta} S_2$

S_1 uses the value x , before it is altered by S_2

(3) $\exists x: x \in OUT(S_1) \wedge x \in OUT(S_2) \wedge S_1 \Theta S_2 \wedge \neg \exists k: (S_1 \Theta S_k \Theta S_2 \wedge x \in OUT(S_k))$
 $\iff S_2$ is **output-dependent** on S_1 : $S_1 \delta^o S_2$

S_2 overwrites the value x , which was calculated earlier in S_1

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

Data Dependence

Definition: Indirect Data Dependence

S_2 is **data-dependent** on S_1 :
 $S_1 \delta^* S_2 \iff S_1 \delta S_2 \vee S_1 \bar{\delta} S_2 \vee S_1 \delta^o S_2$

S_2 is **indirectly data-dependent** on S_1 :
 $S_1 \Delta S_2 \iff \exists S_{k1}, S_{k2}, \dots, S_{kn} (n \geq 0):$
 $S_1 \delta^* S_{k1} \delta^* S_{k2} \delta^* \dots \delta^* S_{kn} \delta^* S_2$

Data Dependence

Example:

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

Dependencies:

```
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 δ̄ S3      (due to A)
S1 δ° S3     (due to A)
```

Indirect Data Dependence

S_2 is **data-dependent** on S_1 :

$$S_1 \delta^* S_2 \iff S_1 \delta S_2 \vee S_1 \bar{\delta} S_2 \vee S_1 \delta^\circ S_2$$

S_2 is **indirectly data-dependent** on S_1 :

$$S_1 \Delta S_2 \iff \exists S_{k1}, S_{k2}, \dots, S_{kn} \ (n \geq 0): \\ S_1 \delta^* S_{k1} \delta^* S_{k2} \delta^* \dots \delta^* S_{kn} \delta^* S_2$$

Data Dependence Direction

Assume S_1 and S_2 are embedded in loops with indices i_1, \dots, i_d

If there exist two specific loop instances $I = (i_1', \dots, i_d')$ and $I'' = (i_1'', \dots, i_d'')$ for the loop indices i_1, \dots, i_d such that the following holds for the appropriate instances of S_1 and S_2 :

$$S_1^{i_1' \dots i_d'} \delta^* S_2^{i_1'' \dots i_d''}$$

and if further the following relation is valid for the index vectors (loop instances):

$$I \Psi I''$$

(that is: $\Psi = (\psi_1, \dots, \psi_d)$ where $\psi_i \in \{<, =, \leq, >, \geq, \neq, ?\}$,
['?' represents an unknown relation]

with $i_1' \psi_1 i_1''$
 $i_2' \psi_2 i_2''$
...
 $i_d' \psi_d i_d''$)

then the following definition holds:

$$\iff S_2 \text{ is data-dependent with direction } \Psi \text{ on } S_1: S_1 \delta_\Psi^* S_2$$

Data Dependence Direction

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: $S_1 \delta_\Psi S_2$ (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 S_1 and S_2 exists the following directed data dependence:

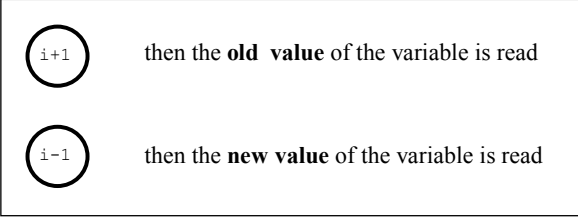
$$S_1 \delta_{(=, <)} S_2$$

⋮

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:



⋮

Loop Vectorization

Example:

```

for i:=1 to n do
S1:  A[i]:= B[i] + C[i];
S2:  D[i]:= A[i+1] + 1;
S3:  C[i]:= D[i];
end;

```

Data Dependences: *Sequence in vectorized code:*

- $S_1 \bar{\delta}_{(=)} S_3$ (due to C) \Rightarrow S_1 before S_3
- $S_2 \bar{\delta}_{(<)} S_1$ (due to A) \Rightarrow S_2 before S_1
- $S_2 \bar{\delta}_{(=)} S_3$ (due to D) \Rightarrow S_2 before S_3



⋮

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 ">" in potentially existing *enclosing* 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.



⋮

Loop Vectorization

Example:

```

for i:=1 to n do
S1:  A[i]:= B[i] + C[i];
S2:  D[i]:= A[i+1] + 1;
S3:  C[i]:= D[i];
end;

```

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

```

S2:  D(1:N) = A(2:N+1) + 1
S1:  A(1:N) = B(1:N) + C(1:N)
S3:  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

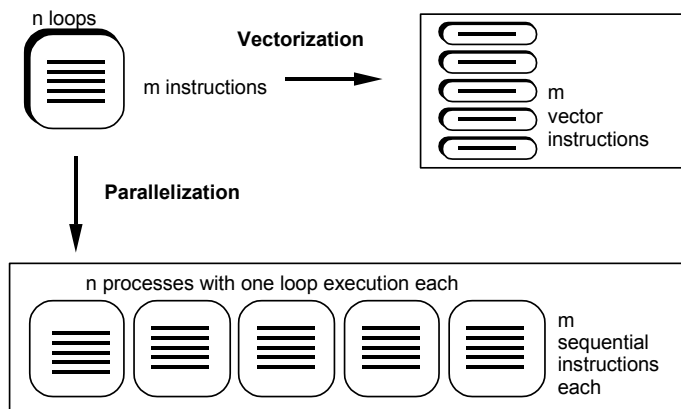
"**doacross**": allocate individual loop-iterations to different processors.
Additional synchronization is required depending on data dependences.

Loop Parallelization Rules

- Data dependencies with direction "=" for the loop to be parallelized do not need to be synchronized.
- Data dependencies with direction "<" or ">" in potentially existing loops *enclosing* the loop to be parallelized can be ignored.
- Potentially existing *inner* loops do not need to be considered when parallelizing; they are completely taken over into the parallelizing code.
- Every other data dependency has to be synchronized between processes via their own arrays of semaphores.
- 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 .

Vectorization and Parallelization

Sequential Loop



Loop Parallelization

Example of Parallelization of a Loop (case a):

```
for i:=1 to n do
  S1: A[i]:= C[i];
  S2: 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

```
doacross i:=1 to n do
  S1: A[i]:= C[i];
  S2: B[i]:= A[i];
enddoacross;
```

⋮

Loop Parallelization

Example of Parallelization of a Loop (case b):

The inner loop is to be parallelized:

```

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

```

Data dependency: S₁ δ (<,<) S₂ (due to A[i,j])

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



⋮

Loop Parallelization

Parallelized Loop:

```

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

```



⋮

Loop Parallelization

Example of Parallelization of a Loop (case c):

The outer loop is to be parallelized:

```

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

```

Determination of all data dependencies:

$$\left. \begin{array}{l} S_1 \delta_{(=,<)} S_2 \quad (\text{due to } A) \\ S_1 \bar{\delta}_{(=,=)} S_2 \quad (\text{due to } B) \end{array} \right\} \Rightarrow \text{no synchronization requirements}$$



⋮

Loop Parallelization

Parallelized Loop:

```

doacross i:=1 to n do
  for j:=1 to n do
    S1: A[i,j]:=B[i,j];
    S2: B[i,j]:=A[i,j-1];
  end;
enddoacross;

```



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:

- S₁ δ(=) S₂ (due to A[i]) ← no synch. required
- S₃ δ(=) S₄ (due to E[i]) ← no synch. required
- S₃ δ(<) S₂ (due to E[i]) ← **synch. required because of direction** "<<"

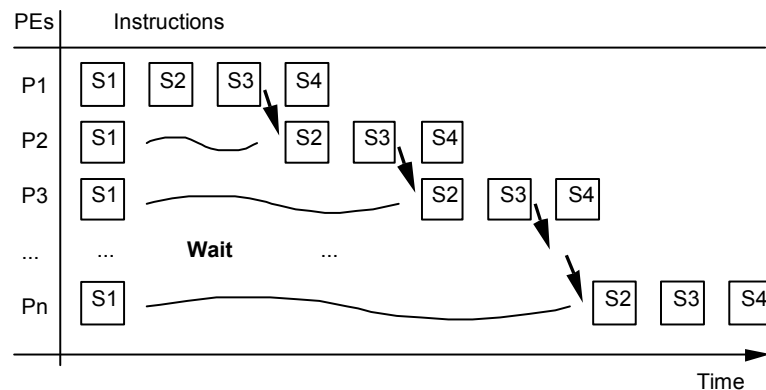
Loop Parallelization with Synchronization

```
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];

  S3: e[i] := e[i] + 2 * b[i];
  V(sync[i]);
  S4: f[i] := e[i] + 1;
enddoacross;
```

Loop Parallelization with Synchronization



Loop Parallelization with Synchronization

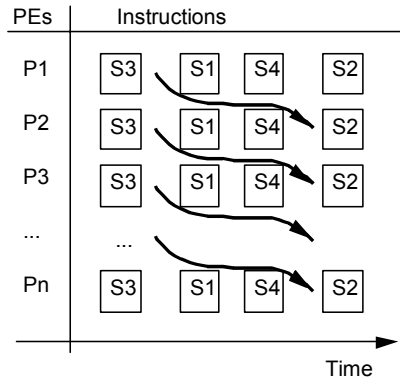
More efficient solution:

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

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

Loop Parallelization with Synchronization

More efficient solution:



6.4 Complex Data Dependences

Circular dependency:

```

for i:=1 to n do
  S1: A[i]:= B[i];
  S2: B[i]:= A[i+1];
end;

```

$$S_2 \bar{\delta}(<) S_1 \quad (\text{due to } A) \Rightarrow S_2 \text{ before } S_1$$

$$S_1 \bar{\delta}(=) S_2 \quad (\text{due to } B) \Rightarrow S_1 \text{ before } S_2$$

No re-arrangement possible !

Complex Data Dependences

Breaking circular dependency via auxiliary variable:

```

for i:=1 to n do
  SH: Aux[i] := B[i];
  S1: A[i] := Aux[i];
  S2: B[i] := A[i+1];
end;

```

The data dependencies change as follows:

$$S_A \bar{\delta}(=) S_1 \quad (\text{due to } Aux) \Rightarrow S_A \text{ before } S_1$$

$$S_A \bar{\delta}(=) S_2 \quad (\text{due to } B) \Rightarrow S_A \text{ before } S_2$$

$$S_2 \bar{\delta}(<) S_1 \quad (\text{due to } A) \Rightarrow S_2 \text{ before } S_1$$

Vectorization is now possible!

Complex Data Dependences

Solution

```

SH: Aux(1:N) = B(1:N);
S2: B(1:N) = A(2:N+1);
S1: A(1:N) = 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

by Wolfe in [Hwang, DeGroot 89]

```

for i:=1 to n do
  for j:=1 to n do
    A[i,j] := A[i,j-1] + A[i,j+1];
  end; (* j *)
end; (* i *)

```

The data dependences are:

$$S_1 \delta_{(=,<)} S_1 \text{ (due to } A[i,j-1]) \Rightarrow S_1 \text{ before } S_1$$

$$S_1 \bar{\delta}_{(=,<)} S_1 \text{ (due to } A[i,j+1]) \Rightarrow S_1 \text{ before } S_1$$

Cannot be vectorized directly!



Loop Interchanging Example

Interchanging inner and outer loop:

```

for j:=1 to n do
  for i:=1 to n do
    A[i,j] := A[i,j-1] + A[i,j+1];
  end; (* i *)
end; (* j *)

```

Data dependences for changed program:

$$S_1 \delta_{(<=)} S_1 \text{ (due to } A[i,j-1])$$

$$S_1 \bar{\delta}_{(=,<)} S_1 \text{ (due to } A[i,j+1])$$

$$\left. \begin{matrix} S_1 \delta_{(<=)} S_1 \text{ (due to } A[i,j-1]) \\ S_1 \bar{\delta}_{(=,<)} S_1 \text{ (due to } A[i,j+1]) \end{matrix} \right\} \Rightarrow \text{no restriction, since surrounding loop has direction '<'}$$

Vectorization possible!



Loop Interchanging Example

Vectorized Solution:

```

do j=1,n
  A(1:n,j) = A(1:n,j-1) + A(1:n,j+1)
end do

```

⋮

Loop Interchanging

Things get more complex if the loop limits are dependable:

```

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

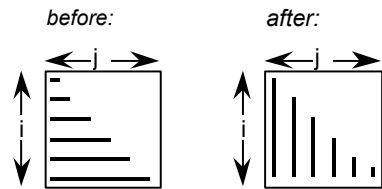
```

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:

```

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

```



⋮

⋮

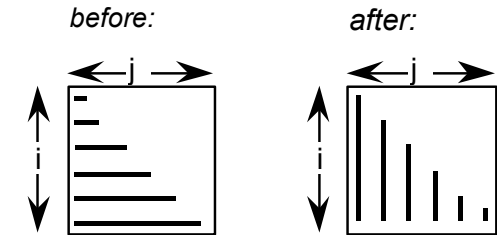
Loop Interchanging

Vectorized Program:

```

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

```



⋮

⋮

Complex Dependences

Double-indexed Access:

```

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

```

It is not possible to resolve this conflict!

⋮

⋮

Complex Dependences

Data dependencies within an instruction:

```

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

```

During the vectorization this is not a dependency:

```

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

```

⋮