Algorithm Mapping

Summer School on Robotics
Malaga 2011

Architectures for real-time processing for robotics
1. Introduction

2. Parallel Architectures

3. Compute Intensive Algorithms


5. Transformation of Algorithms onto Parallel Code for Processors with Subword-Parallelism (SWP)

6. Practice: Parallel Code Design for 1D/2D-Filter
1. Introduction

Real-time processing

Demand:
• large computational resources
• low power consumption
• demand for mobility
• real-time processing

ROBOTS

Real-time processing

Demand:
• large computational resources
• low power consumption
• demand for mobility
• real-time processing

ROBOT

HARDWARE SOFTWARE

Requirement:
REAL-TIME PROCESSING

RULES

Architectures for real-time processing:

Single Processor
pure sequential computing

Single Processor with Sub-word Parallelism (SWP)
parallel computing

Processor Array
parallel computing

Array of Processors with Sub-word Parallelism
multilevel parallel computing

Prof. Merker, TU Dresden
**What is the problem?**

Algorithms for e.g. processing images, signals, fuzzy rules are given as **sequential** algorithms.

**The solution**

For the implementation of real-time applications, it is crucial to identify which parts of the algorithm can be executed in a **parallel** manner. An analysis of the algorithm is performed to determine the best mapping strategies.

**Example**

**Algorithm:** Image reconstruction in computer tomography

**Equation:**

\[
g(k, m) = \sum_{n=-N/2}^{N/2} p(k+n,m) \cdot h(n)
\]

\[
b(i, j) = \sum_{M=0}^{K-1} \sum_{m=0}^{M-1} a(i, j, k, m) \cdot g(k, m)
\]

**Diagram:**

- **Image reconstruction:**
  - Computer tomograph
  - 360 x 360 pixels
  - FBP ART
  - 512 x 512 pixels

- **Filtered Backprojection (FBP):**
  - 92 Giga Operations (MAC) / Scan
Solution of the Mapping Process:
Processor Array for image reconstruction: **RecoChip**

Systems Integration of the Processor Array:

**Aim of the Lecture**

Strategies for mapping algorithms onto parallel architectures:

1. **compute intensive sequential algorithms**
2. ?
3. mapping strategies
4. ?
5. ?

**Prof. Merker, TU Dresden**
Possible application in the summer school:

- 1D-, 2D-correlation
- 1D-, 2D-filters
- fuzzy rules
- parallel processing of signals & images
2. Parallel Architectures

Architectures for real-time processing:

2.1 Processor Arrays

- 2.2 Prozessor with SWP
- 2.3 Prozessor Array with SWP

2.1 Processor Arrays

Basics

Processor Array:
- array of processing elements (PEs)
- regular interconnection of PEs
- each PE has internal registers (memory L0) and hardware modules
- boundary PEs are connected to I/O ports

Systems Integration of PA:
- PA as part of a heterogeneous System on Chip
- PA implemented as ASIC
- PA implemented as FPGA
- PA as hardware accelerator for compute intensive algorithms

---

ERASMUS IP **2011-1-ES1-ERA10-37080
Example: Processor Array for Fuzzy Control as Hardware Accelerator in a System on Chip

Processor Array (Fuzzy Control)

System on Chip

µP I/O DSP
Mem ASIC

History

Systole: Contraction of the left heart ventricle

Systolic Arrays (1979)

Working scheme of a systolic array (examples)

a) 1D-Array

algorithm: correlation:

\[ y_k = \sum_{i=0}^{N-1} a_i \cdot x_{i+k}, \quad k = 0, \ldots, N-1 \]

systolic array for correlation:

* function of one processing element (PE):

\[ y' = y + ax \]
• processor array (default values: $y_2 = y_1 = y_0 = 0$)

- working scheme:

  - $y_2' = 0 + a_0 x_0$
  - $y_0' = a_0 x_0 + a_1 x_1$
  - $y_1' = a_0 x_1 + a_1 x_2$
  - $y_2' = a_0 x_2 + a_1 x_3$
  - $y_0' = a_0 x_0 + a_1 x_1 + a_2 x_2$
  - $y_1' = a_0 x_1 + a_1 x_2 + a_2 x_3$
  - $y_2' = a_0 x_2 + a_1 x_3 + a_2 x_4$
b) 2D-Array

algorithm: fuzzy inference: $c_{kl} = \max_i \min (a_{ik}, b_{kl})$, $i, l = 1, 2, 3, k = 1, 2$

systolic array for fuzzy inference:

- function of one processing element (PE):

- processor array (default values: $c_{kl} = 0$)

- working scheme:

ERASMUS IP **2011-1-ES1-ERA10-37080
Commercial PAs

HiveFlex Moustique-IC2 Processor von Silicon Hive
picoArray von picoChip
Field Programmable Object Array (FPOA) von Math-Star
Dynamically Reconfigurable Processor (DRP) von NEC

SEAforth-24A von intellaSys:
• System on Chip (SoC)
• one component: PA with 6 x 4 processing elements (PEs)
• each PE processes 18-bit data
• each PE has program and data memory
• boundary PEs are connected to other SoC components
• regular interconnection

ERASMUS IP **2011-1-ES1-ERA10-37080
IBM Cell-Processor:
64 Bit IBM powerarchitektur,
8 Synergistic Processing Elements (SPEs), 1 PowerPC Processing Element (PPE)
(9 CPU cores on chip)

application: Sony Playstation

Model of a Processor Array (PA)

weak programmability & efficiency

2.2 Processor with Sub-word Parallelism

What is Sub-word Parallelism (SWP)?

Processing Unit (PU) for processing data e.g. with 64bit data width:

register or memory for data with 64bit data width (full length word (FLW))
Sub-word Parallelism (SWP):
processing unit (PU) for data width, e.g. 64 bit, can be used for processing

\[ \text{in parallel} \]

- two operations on 32 bit data (sub-words (SW))
- four operations on 16 bit data (sub-words (SW))

Constraints:
- Sub-words have to be arranged in the register/memory according to the parallel processing scheme
- Sub-words in a resulting FLW have to be stored separately or rearranged in a new FLW (unpacking, packing)

SWP Instruction set:
Instructions for processing of SWs:
- simple instructions (add, sub)
- complex instructions (sum of products, sum of absolute differences)
- packing/unpacking instructions (to arrange the SWs in a FLW, to separate SWs from a FLW)

Advantages/disadvantages of SWP:
+ high data throughput
+ power consumption for SWP-instructions is similar to power consumption of instructions for FLWs
- packing instructions need additional time and energy

Aim:
Parallel program code for PU which
- exploits sub-word parallelism
- has higher processing speed than code without SWP
Architectures with Subword Parallelism

**General purpose processors:**
- Intel Pentium4, Pentium M: 64 bit and 128 bit data word width
- AMD Athlon 64, Opteron: 64 bit and 128 bit data word width

**SWP instruction sets:**

<table>
<thead>
<tr>
<th></th>
<th>MMX</th>
<th>SSE</th>
<th>SSE2</th>
<th>3DNow!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pentium III</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Intel Pentium 4</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>AMD Athlon</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>AMD Athlon MP</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>AMD Athlon 64</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

---

### 2.3 Processor Array with SWP in each Processing Element

**Multilevel parallelism**

PA with multi-level-parallelism:
- **on array level:**
  - PE to PE
- **on PE level:**
  - PE to PE
- **on PU level:**
  - PU to PU

Sub-word Parallelism (SWP)
2.4 Aim of the lecture

Strategies for mapping algorithms onto parallel architectures:

- Single Processor with Subword Parallelism
- Processor Array (PA)
- 3. Mapping strategies
- 4. Compute intensive sequential algorithms
- 5. ?

Mapping strategies:

- 3.
- 4.
- 5.

Mapping strategies for compute intensive sequential algorithms.
3. Compute intensive Algorithms

3.1 Introduction

Examples:

Signal Processing
- FIR-filter
- Correlation
- STAF-algorithm
  (GSM (JPEG))

Multimedia Algorithms
- 2D Filter
- motion estimation (MPEG)
- edge detection algorithm
- filtered backprojection (CT)
- 3D-surface approximation

Fuzzy Algorithms

Neural Net Algorithms
- Hopfield net
- Backpropagation

Examples:

\[
 y(k) = \sum_{i=0}^{N-1} a(i) x(i+k), \quad k = 0, \ldots, N-1
\]

FIR-Filter:
\[
 y(k) = \sum_{i=0}^{K-1} b(l) x(k-l), \quad k = 0, \ldots, K-1
\]

matrix multiplication:
\[
 c(i, j) = \sum_{k=0}^{L-1} a(i, k) b(k, j), \quad i = 0, \ldots, L-1, \quad j = 0, \ldots, K-1
\]

fuzzy inference:
\[
 c(i, l) = \max_{i} \min \left( a(i, k), b(i, l) \right), \quad i = 0, \ldots, L-1, \quad k, l = 0, \ldots, K-1
\]

STAF:
\[
 d_s(k) = s(k), \quad u_s(k) = s(k)
 d_{s'}(k) = d_{s'}(k) + r \cdot u_{s'}(k-1)
 u_s(k) = u_{s'}(k-1) + r \cdot d_{s'}(k)
 0 \leq i < 8, \quad 0 \leq k < 120
\]

Examples in the lecture:

discrete correlation:
1- dimensional: similarity between two signals $a$ and $x$
\[
 y(k) = \sum_{i=0}^{M-1} a(i) x(i+k), \quad k = 0, \ldots, N-1
\]

2- dimensional: image processing, object recognition in images

see lecture of Prof. Janschek
discrete filtering:
1- dimensional: manipulation of signal \( x \) with signal \( b \)

1D-Filter: \( y(k) = \sum_{i=0}^{K-1} b(i) x(k-i) \), \( k = 0, \ldots, K-1 \)

2- dimensional: e.g. noise reduction in images

Aim:

3.2 Affine Indexed Algorithm (AIA)

affine indexed algorithms (AIA):
\( i \in I, j = 1, \ldots, M : y_i(\{f_i(j)\}) = \mathop{OP} F_j(\ldots, y_i(\{f_i(j)\}), \ldots) \)

\( \Pi, \Sigma, \text{min}, \gcd \)

with \( i \)

\( F_j \)

iteration

function

\( i \in I \subset Z^* \)

iteration space

example: correlation

\( k = 0, \ldots, 4 : y(k) = \sum_{i=0}^{K-1} a(i) x(i+k) \),

\( i = \begin{pmatrix} k \\ i \end{pmatrix} \)
affine indexed algorithms (AIA):

\[ y_i(f_{ij}) = \text{OP } F_{ij}(..., y_k(f_{ij}), ...) \]

with iteration:

\[ i = \frac{k}{i} \]

affine indexed variables:

\[ y(f(i)) = y(M \cdot i + a)^T \]

\[ x(i+k) = x(M \cdot i + a) = x \left( \begin{pmatrix} 1 & 1 \end{pmatrix} \frac{k}{i} + 0 \right) \]

instance of a variable:

\[ x(i+k) = x(5) \text{ for } i = \frac{2}{3} \]

AIA as nested loop program (NLP):

\[
\begin{align*}
\text{for } i_1 = l_1 \text{ to } u_1 \\
\text{...} \\
\text{for } i_n = l_n \text{ to } u_n \\
\text{begin} \\
\quad \begin{cases} 
\quad s_1 \\
\quad s_2 \\
\quad \text{...} \\
\quad s_k \\
\end{cases} \\
\text{k statements} \\
\text{end}
\end{align*}
\]

Statements have not a “single assignment form”

What is a “single assignment form”: to each variable only one variable is assigned.

AIM: Transformation of the Algorithm in form of AIA in a description in single assignment form

3.3 Transformation of the AIA into a single assignment form: System of Uniform Recurrence Equations (SUREs)
given: sequential Algorithm (AIA)

- as a formula:
  \[ y(k) = \sum_{i=0}^{k} a(i) \cdot x(i+k) . \]

- as a nested loop program:
  ```plaintext
  for k = 0 to N-1
      y(k) = 0
  end
  for i = 0 to N-1
      begin
          s: y(k) = y(k) + a(i) \cdot x(i+k)
      end
  ```

sought after:
Transformation of the sequential Algorithm (AIA) in a single assignment form:
System of Uniform Recurrence Equations (SUREs)

automatic MAPPING
Example: correlation \( k = 0, \ldots, 3 \): \( y(k) = \sum_{i=0}^{3} a(i) x(i+k) \), as SUREs:

\[
\begin{align*}
I_{\text{iteration}} &= \left\{ i = \binom{k}{i} \mid 0 \leq k, i \leq 3 \right\} \\
\text{computation per iteration } i &= \binom{k}{i}:
\begin{align*}
A(i) &= \left\{ \begin{array}{ll}
A(i-d_i), & \text{if } \underline{i-d_i} \in I \\
a(i), & \text{else}
\end{array} \right.
X(i) &= \left\{ \begin{array}{ll}
X(i-d_i), & \text{if } \underline{i-d_i} \in I \\
x(i+k), & \text{else}
\end{array} \right.
Y(i) &= \left\{ \begin{array}{ll}
Y(i-d_i) + A(i)X(i), & \text{if } \underline{i-d_i} \in I \\
a(i)X(i+k), & \text{else}
\end{array} \right.
y(k) &= Y(k)
\end{align*}
\]

\[
\begin{pmatrix}
(0) \\
(1) \\
(2) \\
(3)
\end{pmatrix}
\]

Example:

**Definition of a system of uniform recurrence equations (SUREs):**

given: \( E\) Euclidean \( n \)-dim. space, \( L \) lattice of this space
\( i \in \mathbb{L} \), \( i = n \)-dim. vector with integer coordinates

\[
\begin{pmatrix}
L \subset \mathbb{L}
\end{pmatrix}
\]

in each \( i \in \mathbb{L} \):

a set of statements \( s \):

\[
\begin{align*}
s_i &: y_i(i) = f_i(y_{i-d_1}, \ldots, y_{i-d_m}, \ldots) \\
d^*_i &: \text{constant vector independent of } i
\end{align*}
\]

**3.4 Steps for derivating SUREs from AIA**

Aim: Description of the algorithm in a single assignment form

Starting point: Nested Loop Program for AIA:

\[
\begin{align*}
\text{for } i_1 = l_1 \text{ to } u_1 \\
\vdots \\
\text{for } i_x = l_x \text{ to } u_x \\
\text{begin} \\
\quad s_1 \\
\quad \vdots \\
\quad s_k \\
\text{end}
\end{align*}
\]

**1. Step: Determination of the iteration \( i \)**

Iteration \( i = \binom{k}{i} \) contains all loop indices

**Example:**

**2. Step: Determination of the iteration space \( I \)**

Iteration space \( I \) contains all iterations \( i \)

**Example:**

ERASMUS IP **2011-1-ES1-ERA10-37080**
3. Step: Derivation of the computation per iteration \( i \)

a) Derivation of the dependence vectors \( d_i \) of variables \( v(f(i)) \) in the statements

1. \( a1) \) dependent variables:
   - on right and left sight of one statement, 
   - are computed recursively from iteration to iteration 

   **Example:** dependent variable: \( y(k) \)

   **Dependence vectors** \( d_i \) of dependent variables:
   - difference between two successive iterations in the NLP for computing dependent variables

   **Example:** \( y(k) \) is computed in the sequence of iterations:

   \[
   i = \begin{pmatrix} k \\ 0 \end{pmatrix}, \begin{pmatrix} k \\ 2 \end{pmatrix}, \ldots, \begin{pmatrix} k \\ N-1 \end{pmatrix}, \quad k = 0, 
   \ldots, N-1
   \]

   Dependence vector of \( y(k) \): 
   \[
   d_i = \begin{pmatrix} 0 \\ 1 \end{pmatrix}
   \]

2. \( a2) \) input variables:
   - variables only on right sight of one statement

   **Example correlation:**

   **Dependence vectors** \( d_i \) of input variables:
   - The vector \( d_i \) is a dependence vector of an input variable \( v(f(i)) \), if

   \[
   (1) \quad d_i \in \text{null}(M) \text{ with } f(i) = (M \cdot i + a)^T, \\
   \text{null}(M) = \{ d | M \cdot d = 0 \} \text{ is the null space of } M
   \]

   \[
   (2) \quad \gcd(d_i^n, \ldots, d_i^0) = 1 \text{ with } d_i = (d_i^n, \ldots, d_i^0)
   \]

   \[
   (3) \quad d_i \neq 0, \text{ if } \text{possible}
   \]

   **Example:**

---

**ERASMUS IP **2011-1-ES1-ERA10-37080**
b) Derivation of the computation per iteration \( i \)

b1) Introduction of new variables \( V(i) \) for each iteration \( i \):

\[
\begin{align*}
a(i) & \rightarrow A(i) \\
x(i+k) & \rightarrow X(i) \\
y(k) & \rightarrow Y(i)
\end{align*}
\]

b2) Derivation of computation per iteration \( i \):

- Equations for propagating of input variables
  
  \[
  A(i) = \begin{cases} 
  A(i-d_i), & \text{if } i-d_i \in \mathbb{Z} \\
  a(i), & \text{else}
  \end{cases}, \quad d_i \in \begin{bmatrix} 1 & -1 \\
  0 & 0 \end{bmatrix}
  \]
  
  \[
  X(i) = \begin{cases} 
  X(i-k), & \text{if } i-k \in \mathbb{Z} \\
  x(i+k), & \text{else}
  \end{cases}, \quad d_i \in \begin{bmatrix} 1 & -1 \\
  -1 & 1 \end{bmatrix}
  \]

- Equations for computing dependent variables
  
  \[
  Y(i) = \begin{cases} 
  Y(i-d_i) + A(i) \cdot X(i), & \text{if } i-d_i \in \mathbb{Z} \\
  a(i) \cdot x(i+k), & \text{else}
  \end{cases}
  \]

- Output equation
  
  \[
  y(k) = \begin{cases} 
  1, & \text{if } k \in \mathbb{Z} \\
  0, & \text{else}
  \end{cases}
  \]

---

c) Resulting SURE for the Algorithm

SURE:

- Iteration space
- Equations for propagation of input variables
- Equations for computing dependent variables
- Output equation

---

Graph representation:
example: SUREs for the correlation:

\[ L = \{ i = k \mid i \leq k \leq N - 1 \} \]

\[ A(i) = \begin{cases} A(i - d), & \text{if } i - d \in L \\ a(i) & \text{else} \end{cases} \]

\[ X(i) = \begin{cases} X(i - d), & \text{if } i - d \in L \\ x(i + k) & \text{else} \end{cases} \]

\[ Y(i) = Y(i - d) + A(i)X(i), \quad \text{if } i - d \in L \]

\[ y(k) = \begin{bmatrix} k \\ N - 1 \end{bmatrix} \]

### 3.5 Exercise

Derivation of the SUREs for FIR-filter: \( y(k) = \sum_{i=0}^{3} b(i) x(k - i), \quad k = 0, \ldots, 3 \)

with the NLP:

\[
\begin{align*}
& y(k) = \sum_{i=0}^{3} b(i) x(k - i), \quad k = 0, \ldots, 3 \\
& \text{for } i = 0 \text{ to } 3 \\
& \text{begin} \\
& \quad y(k) = y(k) + b(i) x(k - i) \\
& \text{end}
\end{align*}
\]

1. Step: Determination of the iteration \( i \)
2. Step: Determination of the iteration space \( I \)
3. Step: Derivation of the computation per iteration \( i \)
   a) dependence vectors

4.1 Basic Principles

AIM of mapping strategies:
• each iteration is assigned to a PE: **ALLOCATION**
• each iteration is assigned to a time step: **SCHEDULING**
  in consideration of several constraints

Constraints of the algorithm (causality constraints):

\[ y(k) \] (dependent variable)

\[ Y(i) = Y(i-d_y) + \ldots \]

\[ Y(i-d_y) = \ldots \]

SCHEDULING

\[ t_1 \]

with \( t_1 > t_2 \)

\[ t_2 \]
4.2 Main Steps from Algorithm (SUREs) to the Processor Array (PA)

a) Partitioning of the iteration space

- **ALLOKATION** by partitioning the iteration space
- **SCHEDULING**
b) Partitioning Approach

given: $n$ – dimensional iteration space $I$

Partitioning the iteration space with matrix $\Theta = \text{diag} \left( \vartheta \right) = \begin{pmatrix} \vartheta_1 & 0 \\ \vdots & \ddots \\ 0 & \vartheta_n \end{pmatrix}$

$\vartheta_i$ : extension of partition in $i$-th direction of iteration space

example 1:

$L = \{ i = (i, j, k)^T \mid 1 \leq i, j, k \leq 3 \}$

$\Theta = \text{diag} \left( \vartheta \right) = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 3 \end{pmatrix}$

example 2:

$L = \{ i = (i, i, i)^T \mid 0 \leq i, i, i \leq 5 \}$

$\Theta = \text{diag} \left( \vartheta \right) = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$

simple partitioning:

To each iteration $i$ two new coordinates are assigned as follows:

$i = \Theta \hat{\kappa} + \kappa$

with $\Theta \in N^{\max}$

$\hat{\kappa}$ (partition) $\in \hat{\mathcal{K}}$ (set of partitions of iteration space)

$\kappa$ (position in the partition) $\in \mathcal{K}$ (set of all iterations in a partition)

$0 \leq \kappa < \vartheta$

example 2:
Two strategies with respect to allocation and scheduling

- Locally sequential globally parallel (LSGP)- partitioning:
  - Each partition corresponds with one PE
  - Iterations of partitions are computed in sequential
  - Partitions (PEs) are computed in parallel

\[ i = \Omega^{LS} \kappa^{LS} + \kappa^{LS} \]

- Locally parallel globally sequential (LPGS)- partitioning:
  - Each partition corresponds with the PA
  - Iterations of partitions are computed in parallel
  - Partitions are computed in sequential on the PA

\[ i = \Omega^{LP} \kappa^{LP} + \kappa^{LP} \]

Evaluation of LPGS- and LSGP-partitioning

LSGP
  - Number of PEs
  - Arrangement of PEs
  - I/O communication restrictions of PEs
  - PA (i.e. number of local memory (L0) in the PEs) depends on algorithm size

LPGS
  - Number of PEs
  - Arrangement of PEs
  - PA independent of algorithm size
  - I/O communication of the array depends on partition size

Disadvantage:
No balance of I/O communication and local memory size (L0) in the PA
4.3 Special Case: Mapping SUREs to PA using LPGS-partitioning

a) LPGS- partitioning, allocation and scheduling

Allocation:

Scheduling:
Scheduling:

Determination of a sequence of the LPGS-partitions, i.e. mapping a time step \( t \) to each partition \( \mathbf{k} \) as follows:

\[
\mathbf{t} = \mathbf{c} \mathbf{k}
\]

with

\[
\mathbf{c} : n \text{-dimensional scheduling vector with}
\]

- \( \forall \mathbf{k}_1, \mathbf{k}_2 \in \mathbf{K} : \mathbf{k}_1 \neq \mathbf{k}_2 \Rightarrow \mathbf{c} \mathbf{k}_1 \neq \mathbf{c} \mathbf{k}_2 \)
  (partitions have different time steps)

- \( \max_{\mathbf{k}_2 \in \mathbf{K}} \mathbf{c} \mathbf{k} - \min_{\mathbf{k}_2 \in \mathbf{K}} \mathbf{c} \mathbf{k} = |\mathbf{k}| - 1 \)
  (compact schedule)

How can we determine the scheduling vector \( \mathbf{c} \)?

given: \( n \)-dimensional iteration space

\( \rightarrow \) \( 2^n \) directions in the iteration space according to the \( 2^n \) unit vectors: \( \mathbf{e}_1 = (0, \ldots, 0, +1, 0, \ldots, 0) \), \( \mathbf{e}_2 = (0, \ldots, 0, -1, 0, \ldots, 0) \),

example:

\[
\begin{align*}
\mathbf{e}_1 &= (1, 0), & \mathbf{e}_2 &= (-1, 0), \\
\mathbf{e}_3 &= (0, 1), & \mathbf{e}_4 &= (0, -1).
\end{align*}
\]

Succession vector \( \mathbf{a} \):

Succession vector \( \mathbf{a} = (\alpha_1, \ldots, \alpha_n) \) describes the sequence of directions in the \( n \)-dimensional iteration space.

(\( n! \cdot 2^n \) possibilities)

Example for succession vectors:

\[
\begin{align*}
\mathbf{a} &= (\alpha_1, \alpha_2) = (-1, 2) \\
\mathbf{a} &= (\alpha_1, \alpha_2) = (2, 1)
\end{align*}
\]
Mapping Strategies of Algorithms onto Parallel Architectures

Scheduling vector $\tau$:

To each succession vector $\alpha = (\alpha_1, \ldots, \alpha_n)$ of the iteration space partitioned with $\Theta$

the scheduling vector $\tau = (\tau_1, \ldots, \tau_n)$ can be determined as follows:

\[
\tau_{|k|} = \begin{cases} 
\text{sign} (\alpha_k), & \text{if } \hat{\delta}_{|k|} > 1 \land k = 1 \\
\text{sign} (\prod_{i=1}^{k-1} \hat{\delta}_{|k|}), & \text{if } \hat{\delta}_{|k|} > 1 \land k > 1 \\
0, & \text{if } \hat{\delta}_{|k|} = 1 
\end{cases}
\]

$1 \leq k \leq n$

with

\[
\hat{\delta} = \begin{pmatrix} \hat{\delta}_1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \hat{\delta}_n \end{pmatrix}
\]

$\hat{\delta}_i$ number of LP-partitions in the $i$-th direction of iteration space

Examples for determining the scheduling vector:

\[
\begin{array}{c|c}
\hline
\alpha = (\alpha_1, \alpha_2) = (-1, 2) & \alpha = (\alpha_1, \alpha_2) = (2, 1) \\
\hline
\hat{\delta} = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix} & \hat{\delta} = \begin{pmatrix} \hat{\delta}_1 \\ \vdots \\ \hat{\delta}_{i_2} \end{pmatrix} \\
\hline
\tau = (\tau_1, \tau_2) = (-1, 3) & \tau = (\tau_1, \tau_2) = (2, 1) \\
\hline
\end{array}
\]

time steps for the partitions: $t = \tau \hat{K}$

b) Mapping of Algorithms (SUREs) to Processor Arrays using LPGS-Partitioning

Allocation:

Selection of a partitioning matrix: $\Theta = \begin{pmatrix} \delta_1 & 0 \\ \vdots & \ddots \\ 0 & \delta_n \end{pmatrix}$

Scheduling:

To each iteration $i$ a time step $t$ is mapped as follows:

\[
t = \tau \hat{K} + \tau^{\text{off}} \hat{K}
\]

with

- $\tau$ scheduling vector
- $\tau^{\text{off}}$ offset vector to avoid causality constraints
Constraints for scheduling vector $\tau$:

given:

- partitions
- dependence vector $\uparrow$ of the dependent variable

Causality constraints between partitions:

Find a scheduling vector so that $t_2 > t_1$ $t_4 > t_3$ $t_4 > t_3 > t_2 > t_1$

Causality constraints inside partitions:

<table>
<thead>
<tr>
<th>causality conflicts!</th>
<th>no causality conflicts!</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau^{off} \neq (0 \ 0)$</td>
<td>$\tau^{off}$ arbitrary</td>
</tr>
</tbody>
</table>

Steps for Mapping SUREs into Processor Arrays (PA)

**Correlation (SUREs):**

\[ y(k) = \sum_{i=0}^{i=N} a(i) x(i+k) \]

**example:** $k = 0, ..., 3$:

\[ y(k) = \sum_{i=0}^{i=N} a(i) x(i+k) \]

**Correlation (SUREs):**

\[ y(k) = \sum_{i=0}^{i=N} a(i) x(i+k) \]

**special case:** $0 \leq i, k \leq 3$

**general approach:** $0 \leq i, k \leq N-1, N > 4$

\[ d_i = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 0 \end{bmatrix} \]

\[ d_i = \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1 \end{bmatrix} \]

\[ d_i = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \]

**1. Step:**

- Selection of a partitioning matrix $\Theta^P = \Theta = \begin{pmatrix} \alpha_1 & \alpha_2 \\ \beta_1 & \beta_2 \end{pmatrix}$

  **version 1:**

  \[ \Theta = \begin{pmatrix} 4 & 0 \\ 0 & 1 \end{pmatrix} \Rightarrow \hat{\Theta} = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix} \]

  **version 2:**

  \[ \Theta = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix} \Rightarrow \hat{\Theta} = \begin{pmatrix} 4 & 0 \\ 0 & 1 \end{pmatrix} \]

- Selection of a scheduling vector $\tau$

  succession vector: $\alpha = (\alpha_1 \ \alpha_2) = (1 \ 2)$

  scheduling vector: $\tau = (\tau_1 \ \tau_2) = (0 \ 1)$

  scheduling vector: $\tau = (\tau_1 \ \tau_2) = (1 \ 0)$
2. Step: Selection of an offset vector $\tau^{\text{off}}$

version 1:

<table>
<thead>
<tr>
<th>$\tau^{\text{off}} = (0 \ 0)$</th>
<th>$\tau^{\text{off}} \neq (0 \ 0)$</th>
</tr>
</thead>
</table>

version 2:

<table>
<thead>
<tr>
<th>$\tau^{\text{off}}$</th>
<th>$\tau^{\text{off}}$</th>
</tr>
</thead>
</table>

version 1, case 1:

\[
\tau = (0 \ 1) \\
\tau^{\text{off}} = (0 \ 0)
\]

\[t = \tau \hat{k}\]

version 1, case 2:

\[
\tau = (0 \ 1) \\
\tau^{\text{off}} = (1 \ 0)
\]

\[t = \tau \hat{k} + \tau^{\text{off}} \hat{k}\]

version 2:

\[
\tau = (1 \ 0) \\
\tau^{\text{off}} = (0 \ 2)
\]

\[t = \tau \hat{k} + \tau^{\text{off}} \hat{k}\]

3. Step:

Selection of the dependence vectors $d_v$ of the input variables $v$ so that

- $d_v$ connects two iterations with the same time step $t$
- or $d_v$ is directed from iteration $i_1$ with time step $t_1$ to iteration $i_2$ with time step $t_2$, where $t_1 < t_2$. 

4. Step: Derivation of the processor array (PA)

version 1, case1:
version 1, case 2:

PA:

version 2:

PA:

General Approach (correlation $0 \leq i, k \leq N-1, N > 4$, variant 1, case 1)
Exercise:

Derivation of a Processor Array for FIR-filter: \( y(k) = \sum_{l=0}^{L-1} b(l) x(k-l), \quad k = 0, \ldots, K-1 \)

1. special case: \( 0 \leq l, k \leq 3 \)

\[ \Theta = \begin{pmatrix} 4 & 0 \\ 0 & 1 \end{pmatrix} \]

\( \alpha = (2,1), \quad \tau = (0,1) \)

\( \Sigma^{\text{off}} = (0,0) \)

PA with 4 PEs

2. general approach: \( 0 \leq l \leq L-1, 0 \leq k \leq K-1 \)

Solution:

1. special case: \( 0 \leq l, k \leq 3 \)
2. general approach: $0 \leq l \leq L - 1, 0 \leq k \leq K - 1$ with $K \mod 4 = 0, L \mod 4 = 0$
5. Transformation of Algorithms into Parallel Code

5.1 From algorithm to parallel code - basic principles

**ALGORITHM:**
\[ y(k) = \sum_{i=0}^{N-1} a(i)x(i+k) \]

\[ k = 0, \ldots, N - 1 \]

**ARCHITECTURE:** single processor with SWP

**Processor Array**
(N = 4)

**SWP on a single processor:**

**Instructions**

**for data packing/unpacking**

**for processing sub-words in parallel**

**FLW:** full length word
Streaming SIMD Extensions (SSE)
SSE intrinsics
(SSE assembler instructions wrapped in highlevel C - functions)

Instructions:

• __m128 : virtueller SSE Register

  128 bit
  32 bit  32 bit  32 bit  32 bit

• __m128 a : virtueller SSE Register for a-data

  128 bit
  a(4)  a(3)  a(2)  a(1)

Beispiel:

Selected instructions:

register r:
• r = _mm_load1_ps (& a[i])
  a(i)  a(i)  a(i)  a(i)
loads one floating point value from a given memory adress (& a[i]: pointer at adress a[i]) and broadcasts the value in the whole __m128 register r

register r:
• r = _mm_loadu_ps (& a[i])
  a[i+3]  a[i+2]  a[i+1]  a(i)
loads four floating point values beginning from the given memory adress (& a[i]) into the __m128 register r

  from lower to higher indices!!

• r_y = _mm_set1_ps( 0.0 )
setting of the four floating point values of register r_y to 0.0

• _mm_storeu_ps ( &y[i], r )
saves the four floating point values of the __m128 register r in an memory array defined by the starting adress &y[i]

register r:

  y(i+3)  y(i+2)  y(i+1)  y(i)

memory array:

  y[i]  
  *  
  y(i)  y(i+1)  y(i+2)  y(i+3)
5.2 Design steps from SUREs to parallel code for SWP

ALGORITHM (SUREs):

a) selection of instructions for PU
b) data packing/unpacking in registers
   register optimization

c) code

ARCHITECTURE:
single processor with SWP
a) Selection of instructions for parallel processing in the PU

Example: Correlation  \( y(k) = \sum_{i=0}^{N-1} a(i) x(i + k), \quad k = 0, \ldots, N-1 \)

**SUREs:**

\[
A(i) = \begin{cases} 
A(i-d), \text{ if } i-d \in I \\
A(i), \text{ else }
\end{cases}
\]

\[
X(i) = \begin{cases} 
X(i-d), \text{ if } i-d \in I \\
x(i+k), \text{ else }
\end{cases}
\]

\[
Y(i) = \begin{cases} 
Y(i-d) + A(i) X(i), \text{ if } i-d \in I \\
A(i) X(i+k)
\end{cases}
\]

\[
y(k) = Y \left( i = \left\lfloor \frac{k}{M-1} \right\rfloor \right)
\]

\[
I = \left\{ i = \left\lfloor \frac{k}{M} \right\rfloor \mid 0 \leq k \leq N-1, 0 \leq i \leq M-1 \right\}
\]

**Instructions:**

- parallel MUL & parallel ADD

**b) Data packing/unpacking of sub-words in the registers**

**Basics:**

**ALGORITHM (SUREs):**

Correlation

\[
y(k) = \sum_{i=0}^{3} a(i) x(i + k), \quad k = 0, \ldots, 3
\]

**Design of an 1-dim. PA:**

**Data packing in the registers:**

- \( y \) - data: no packing necessary (data reuse in the same subword)
- \( x \) - data: pipelining of one sub-word to right form FLW to FLW
- \( a \) - data: broadcasting, all sub-words in a FLW are the same

**Prof. Merker, TU Dresden**
Size adaptation

General case:
\[ y(k) = \sum_{i=0}^{N-1} a(i)x(i + k) , \quad k = 0, \ldots, N-1 \]

Constraints:
Using of the above determined packing instructions

Method: LPGS-Partitioning

4 parallel units

LPGS-Partitioning of the iteration space:

Register:
\[ x(3) \quad x(2) \quad x(1) \quad x(0) \]

Reflection of the iteration space on \( i \)-axis:

ERASMUS IP **2011-1-ES1-ERA10-37080
c) Parallel program code

**Transformation of Algorithms into Parallel Code**

ERASMUS IP **2011-1-ES1-ERA10-37080**

Correlation without using SSE (sequential code):

```c
bool correlation ( float *y, float *a, float *x, int M, int N )
{
    for ( int k = 0; k < N; k++ )
        for ( int i = 0; i < M; i++ )
            y[k] = a[i] * x[k+i];
    return true;
}
```

version 1 of parallel code: direct data reading is possible (unaligned access):

```c
bool correlation_sse_direct ( float *y, float *a, float *x, int N, int M )
{
    __m128 r_y, r_a, r_x;
    int k = 0;
    for ( ; k < N; k+=4 )
    {
        r_y = _mm_set1_ps ( 0.0 );
        for ( int i = 0; i < M; i++ )
        {
            r_a = _mm_load1_ps ( &a[i] );
            r_x = _mm_loadu_ps ( &x[k+i] );
            r_y = _mm_add_ps ( r_y, r_x * a[i] );
        }
        _mm_storeu_ps ( &y[k], r_y );
    }
    return true;
}
```

version 2: direct data reading is not possible (aligned access)

```c
bool correlation_sse_shift ( float *y, float *a, float *x, int M, int N )
{
    __m128 r_y, r_a, r_x, r_z, r_t;
    int k = 0;
    for ( ; k < N; k+=4 )
    {
        r_y = _mm_set1_ps ( 0.0 );
        for ( int i = 0; i < M; i++ )
        {
            r_a = _mm_load1_ps ( &a[i] );
            r_z = _mm_loadu_ps ( &x[i+k+3] );
            if ( i%4 == 1 )
                r_x = _mm_loadu_ps ( &x[i+k] );
            else
            {
                r_t = _mm_shuffle_ps ( r_z, r_x, _MM_SHUFFLE ( 2,3, 0,1 ) );
                r_x = _mm_shuffle_ps ( r_x, r_t, _MM_SHUFFLE ( 1,2, 1,2 ) );
            }
            r_z = _mm_shuffle_ps ( r_z, r_z, _MM_SHUFFLE ( 0,3,2, 1 ) );
            r_x  = _mm_shuffle_ps ( r_x, r_t, _MM_SHUFFLE ( 1,2 ,2,1 ) );
            r_z = _mm_shuffle_ps ( r_z, r_z, _MM_SHUFFLE ( 0,3,2, 1 ) );
            if ( i%4 == 1 )
                r_x = _mm_loadu_ps ( &x[i+k] );
            else
            {
                r_t = _mm_shuffle_ps ( r_z, r_x, _MM_SHUFFLE ( 2,3,0,1 ) );
                r_x = _mm_shuffle_ps ( r_x, r_t, _MM_SHUFFLE ( 1,2,1,2 ) );
                r_z = _mm_shuffle_ps ( r_z, r_z, _MM_SHUFFLE ( 0,3,2,1 ) );
            }
            r_t = _mm_mul_ps ( r_x, r_a );
            r_y = _mm_add_ps ( r_y, r_t );
        }
        _mm_storeu_ps ( &y[k], r_y );
    }
    return true;
}
6. Practice: Parallel Code Design for 1D/2D-Filter

real world application:

underwater robot
(Stanford University)

underwater camera ——> images with noise

Gaussian filter

images without noise

Gaussian filter:

application of Gaussian Filter at images

Gaussian distribution curve (bell-shaped):

Each pixel is replaced by a new value where the adjacent pixels are weighted with a Gaussian curve.

The little structures are lost, the bigger coarse ones remain in the image:

smoothing soft-focus-effect

Usage:

• Improvement of an image
• Reduction of noise
• Matching the informative parts

Practice:

1. Developing a parallel program for 1-D-filtering

2. Putting it in a given software environment for 2-D-filtering of images using 2 times 1-D-filtering (as the bell-shaped curve is symmetric):

3. Measuring the time periods for the sequential and parallel 2-D-filtering
1. Developing a parallel program for 1-D-filtering:

\[
y(k) = \sum_{l=0}^{L-1} b(l)x(k-l), \quad k = 0, \ldots, K-1
\]

4 parallel units

a) Selection of instructions for parallel processing

SUREs for FIR-filter:

\[
B[i] = \begin{cases} 
B[i-d_i] & \text{if } i-d_i \in I \\
b(i) & \text{else} 
\end{cases} \\
X[i] = \begin{cases} 
X[i-d_i] & \text{if } i-d_i \in I \\
x(k-l) & \text{else} 
\end{cases} \\
Y[i] = \begin{cases} 
Y[i-d_i] + B[i]X[i] & \text{if } i-d_i \in I \\
B(i)x(k-l) & \text{else} 
\end{cases}
\]

\[y(k) = Y[k, L-1]\]

\[
\begin{align*}
&\text{Parallel MUL} \\
&\text{Parallel ADD} \\
&\text{Instructions:} \\
r_t = _mm\_mul\_ps(r_b, r_x) \\
r_y = _mm\_add\_ps(r_t, r_y)
\end{align*}
\]

b) Data packing

FIR-Filter → Design of an 1 – dim. PA:

1 – dim. PA:
Data packing in the registers:

- **y** - data: no packing necessary (data reuse in the same subword)
- **x** - data: pipelining of one sub-word to left form FLW to FLW
- **b** - data: broadcasting, all sub-words in a FLW are the same

### Size adaption

**General case:**

\[ y_l = \sum_{i=0}^{L-1} b_l x_{a-i}, \quad k = 0, \ldots, K-1 \]

**Constraints:**

Using of the above determined packing instructions

**Method:** LPGS-Partitioning

**4 parallel units**

### LPGS-Partitioning of the iteration space:

\[ a \]

\[ b \]

\[ c \]

\[ d \]

\[ x(3) \quad x(2) \quad x(1) \quad x(0) \]

\[ b(3) \quad b(2) \quad b(1) \quad b(0) \]
c) Parallel program code

sequential code:

```c
bool filter ( float *y, float *x, int K, float *b, int L ) {
    for ( int k = 0; k < K; k++ )
        for ( int l = 0; l < L; l++ )
            y[k] += b[l]*x[k-l];
    return true;
}
```

version 1: parallel code (direct data reading is possible (unaligned access))
Approach to the Labwork „Algorithm Mapping“

1. Start of the project Teilwortparallelitaet using the file Teilwortparallelitaet.sln

2. Get familiar with Visual Studio using the subproject correlation of the project teilwortparallelitaet.

The subproject correlation has to be defined as starting project using the right mouse bottom.

Use again the right mouse bottom of the subproject and activate:

- bereinigen (purge) and
- neu erstellen (create new) and
- Define the Program-arguments in Properties/Debugging/Instruction-arguments:
  - Overlapping of the signal(M)<Length of the output signal (N)><computing method (Orig, SSEdirect, SSEshift, SSEshift2, SSEshift3)> (z.B. 300000 2000 Orig)
- Release: measuring the time periods for computing the correlation with the different computing methods
3. Get familiar with the Shuffle-instructions:

Define the subproject `test_instruction` as starting project using the right mouse bottom.

Use again the right mouse bottom of the subproject `test_instruction` and activate

- bereinigen (purge) and
- neu erstellen (create new).

Afterwards you can test and play with the Shuffle-instructions.

4. Derive the parallel programs for 1-D-Filtering

(Steps are described in the script analogous to the correlation)

5. Insert the parallel programs for 1-D-Filtering into the subproject `filtering-students`:

- The subproject `filtering-students` has to be defined as starting project using the right mouse bottom.
- Fill the parallel programs in the prepared gaps in the file `filter.h`:
  - Program Orig (sequenzielles Programm) exists already
  - Program SSEdirect (the subwords can read directly from the memory)
  - Programm SSEshift (to avoid direct memory access the subwords are packed using packing operations)
  - Program SSEshift2 (can be an optimized version of SSEshift)
- Activate subproject `filtering-students` using the right mouse bottom:
  - Bereinigen (purge) and
  - neu erstellen (create new) and

The further files in subproject `filtering-students` are used as follows:

- `smoth_color.cpp`: main data file,
  - read the received image file (file format: BMP) and write the image pixels into an 3D-array.
  - Filtering of the image (1D-Filtering) in X-direction for all three color spaces (RGB) using the defined parameters and method
  - Transpose the image after 1D-Filtering
  - Filtering of the image (1D-Filtering) in Y-direction for all three color spaces (RGB) using the defined parameters and method
  - Transpose the image resulting after the twofold filtering.
  - Write the resulting image in an output file (BMP).
- Output of the computing time
  - Filtering of the image (1D-Filtering) in X-direction for all three color spaces (RGB) using the defined parameters and the method „Orig“.
  - Compare if the images resulting from 1D_Filtering in X-direction using method „Orig“ or the defined method respectively are identical
- `bitmap.h`: Funktions for handling BMP-images
- `img.h`: Funktions for reading and writing of the images and for creating and deleting of 2D- und 3D-Arrays