Multiprocessor programming, DV2597/DV2606

Parallel Programming Models (Chapter 3)

Håkan Grahn

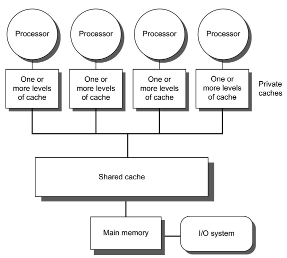
(some slides by G. Karypis, T. Rauber and G. Rünger)

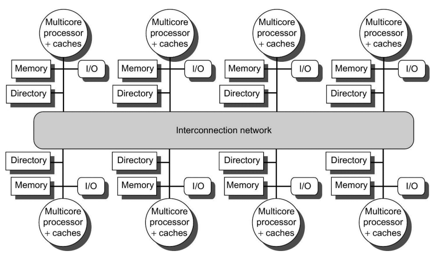
Topic overview

- Models for parallel systems
- Parallelization of programs
- Levels of parallelism
- Data distribution for arrays
- Information exchange
- Parallel Matrix-Vector Multiplication

Shared-Address-Space Platforms

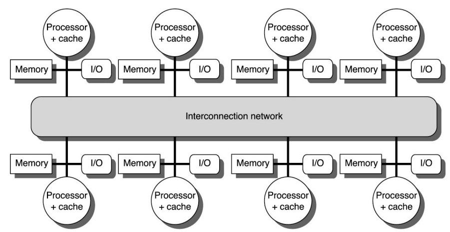
- The memory is accessible to all processors.
  - Processors interact by modifying data objects stored in this shared-address-space => Needs synchronization!
- If the access time of any memory word in the system is equal, the platform is classified as a uniform memory access (UMA), else, a non-uniform memory access (NUMA) machine.





Message-Passing Platforms

- These platforms comprise of a set of processors and their own (exclusive) memory.
  - Examples: Clustered workstations and non-shared-address-space multicomputers.
- These platforms are programmed using (variants of) send and receive primitives.
  - Libraries such as MPI and PVM provide such primitives.



Topic overview

- Models for parallel systems
- Parallelization of programs
- Levels of parallelism
- Data distribution for arrays
- Information exchange
- Parallel Matrix-Vector Multiplication

Models for parallel systems

Distinction according to level of abstraction

- **Parallel machine models:** lowest level of abstraction – hardware related description of the system
- **Parallel architectural models:** Abstraction of machine models – (topology, synchronous or asynchronous operation of the processor, SIMD or MIMD, memory organization
- **Parallel computational models:** Extension of the architectural models, by which algorithms can be constructed and their costs can be considered, e.g., PRAM-Model (parallel random access machine)
- **Parallel programming models:** Description of a parallel system by describing the programming language and environment

Criteria for parallel programming models

- What kind of parallelism from the computation can be used?
  (instruction level parallelism, function level, parallel loops)
- Has the programmer to specify the parallelism and how is the parallelism specified?
  (explicit or implicit specification of parallelism)
- In which way has the programmer to specify the parallelism?
  (e.g., independent tasks, managed by task pools or processes that are generated and have to communicate to each other)
- How is the execution of the parallel units organized?
  (SIMD or SPMD, synchronous or asynchronous)
- How is the information exchange organized?
  (communication with messages or by using shared variables)
- What kind of synchronization can be used?

Topic overview

- Models for parallel systems
- Parallelization of programs
- Levels of parallelism
- Data distribution for arrays
- Information exchange
- Parallel Matrix-Vector Multiplication

Parallelization of programs (I)

1. Decomposition (partitioning) of the computations

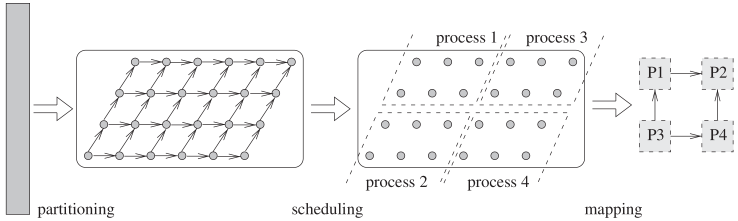
- Decomposition of the algorithm into tasks.
- Specification of task dependencies.
- Tasks include an unrestricted set of computations and
  - access to shared variables (on shared memory systems) or
  - they exchange messages by communication operations (on distributed memory systems)
- **Granularity** of a task: Number of computations performed by a task.

Parallelization of programs (II)

2. Assignment of tasks to processes

- Processes execute tasks successively.
- The aim of the assignment of tasks to processes is to execute nearly the same number of computations by each process, such that a good load balance occurs.
- The assignment of tasks to processes is denoted as scheduling.

3. Mapping of processes to physical processors



Topic overview

- Models for parallel systems
- Parallelization of programs
- Levels of parallelism
- Data distribution for arrays
- Information exchange
- Parallel Matrix-Vector Multiplication

Levels of parallelism

- Depending on the level considered, tasks of different **granularity** result
  - Instruction-level parallelism
  - Data parallelism
  - Loop parallelism
  - Functional parallelism

Instruction-level parallelism

- Multiple instructions can be executed in parallel
- **Data dependencies** between instructions I_1 and I_2 limit the parallel execution
  - **Flow dependency** (also called *true dependency*): There is a flow dependency from instruction I_1 to I_2, if I_1 computes a result value in a register or variable which is then used by I_2 as operand.
  - **Anti-dependency**: There is an anti-dependency from I_1 to I_2, if I_1 uses a register or variable as operand which is later used by I_2 to store the result of a computation.
  - **Output dependency**: There is an output dependence from I_1 to I_2, if I_1 and I_2 use the same register or variable to store the result of a computation.

flow dependency	anti dependency	output dependency
I_1: $\underline{R_1} \leftarrow R_2+R_3$	I_1: $R_1 \leftarrow \underline{R_2}+R_3$	I_1: $\underline{R_1} \leftarrow R_2+R_3$
I_2: $R_5 \leftarrow \underline{R_1}+R_4$	I_2: $\underline{R_2} \leftarrow R_4+R_5$	I_2: $\underline{R_1} \leftarrow R_4+R_5$

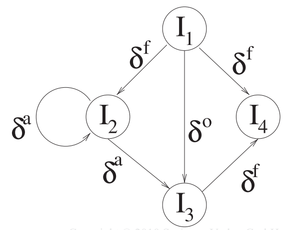
Data dependencies - example

I_1: $R_1 \leftarrow A$

I_2: $R_2 \leftarrow R_2+R_1$

I_3: $R_1 \leftarrow R_3$

I_4: $B \leftarrow R_1$



Data parallelism

- Special constructs to perform the same operation on multiple data elements in parallel (also referred to as SIMD), e.g.,

```
a(1:n) = b(0:n-1) + c(1:n)
```

is the same as

```
for (i = 1:n)
     a(i) = b(i-1) + c(i)
endfor
```

Data parallelism

- Data parallelism is often used also in MIMD, e.g., by using the **Single Program Multiple Data** (SPMD) model
  - One parallel program executed in parallel on all processors
  - In practice, many parallel programs are SPMD
  - For example, each processor calculates a part of an array

```
local_size = size/p;
local_lower = me * local_size;
local_upper = (me+1) * local_size - 1;
local_sum = 0;

for (i=local_lower; i<=local_upper; i++)
       local_sum += x[i] * y[i];

Reduce(&local_sum, &global_sum, 0, SUM);
```

Loop parallelism

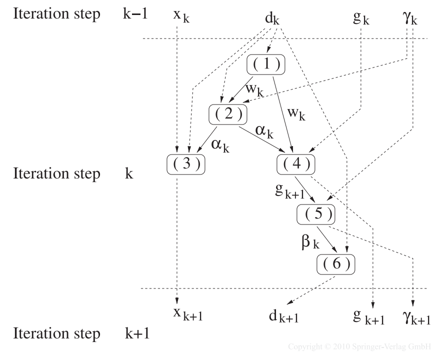
- Execute loop iterations in parallel
  - Example: forall and doall
  - Careful, since they may behave differently (check language etc.)

```
for (i=1:4)              forall (i=1:4)           dopar (i=1:4)
  a(i)=a(i)+1              a(i)=a(i)+1              a(i)=a(i)+1
  b(i)=a(i-1)+a(i+1)       b(i)=a(i-1)+a(i+1)       b(i)=a(i-1)+a(i+1)
endfor                   endforall                enddopar
```

start values			after for-loop	after forall-loop	after dopar-loop
a(0)	1				
a(1)	2	b(1)	4	5	4
a(2)	3	b(2)	7	8	6
a(3)	4	b(3)	9	10	8
a(4)	5	b(4)	11	11	10
a(5)	6				

Functional parallelism

- Functional parallelism (a.k.a. task parallelism) relies on dividing the program into independent tasks
- The tasks and their dependencies can be represented in a task graph
- Tasks can be executed in parallel as long as their task dependencies are maintained
- Tasks are scheduled either static or dynamically on multiple processors
  - Task pool is a popular model for dynamic task scheduling



Arrangement of tasks / threads

Parallel design patterns (= structures of coordination of the threads) can be used for organizing the cooperation of tasks / threads of a program:

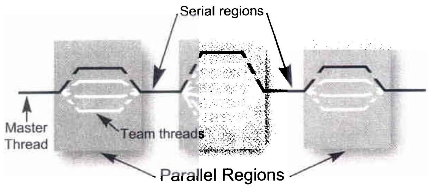
- Creation of threads
- Fork-Join
- Parbegin-Parend
- SPMD and SIMD
- Master-Slave or Master-Worker
- Client-Server-Model
- Pipelining
- Taskpools
- Producer-Consumer-Threads

Creation of threads

- The creation of processes or threads can be carried out *statically or dynamically*.
  - **Static thread creation:** A *fixed number* of processes or threads are created at program start, all processes or threads exist during the entire execution of the parallel program, and are terminated when program execution is finished.
  - **Dynamic thread creation:** Allow creation and termination of processes or threads dynamically at arbitrary points during program execution, i.e., at *run-time*.

Fork-Join

- An existing thread T1 creates a second thread T2, or a group of threads
- Arbitrary nesting
- Specific characteristics in different parallel programming languages and environments



Parbegin-Parend

- Simultaneous creation and destruction of several threads (structured variant of thread creation)
- The statements included by Parbegin-Parend are mapped to separate threads;
  The statements following the Parend statement are executed after all additional threads are destroyed
- Actual parallel execution depends on implementation
- Specific characteristics in individual programming languages and environments (e.g., parallel sections in OpenMP)

SPMD and SIMD

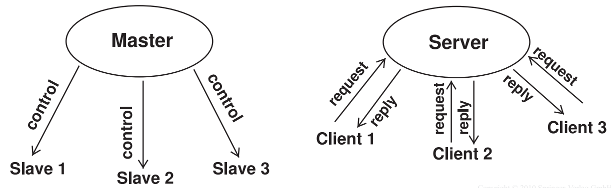
- SIMD - **S**ingle **I**nstruction, **M**ultiple **D**ata
- SPMD - **S**ingle **P**rogram, **M**ultiple **D**ata

- All threads execute the same program but with different data
  - SIMD: *synchronously*, i.e., all threads execute the same instruction simultaneously. This means data-parallelism in the strict sense.
  - SPMD: *asynchronously*, i.e., at a time different threads execute different program statements at the same time.

Master-Slave or Master-Worker, vs. Client-Server model

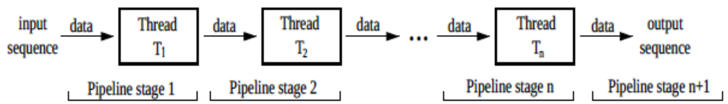
- **Master-Slave:** One single thread controls the whole computations of a program; creates mostly similar Worker- or Slave-threads which get computations assigned

- **Client-Server-Model:** Several client threads make requests to the server thread; server thread handles client requests concurrently/parallel (Extensions: Several server threads or threads which are client and server at the same time)



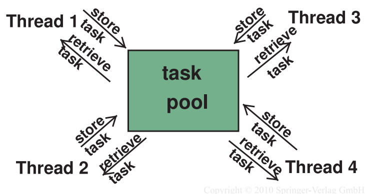
Pipelining

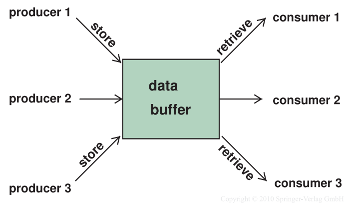
- Threads $T_1, \ldots, T_n$ are logically ordered in a specified order;
  - Thread T_i gets the output of thread T_{i-1} as input and computes its output, that will be used by thread T_{i+1}, $i = 2, \ldots, n-1$, as input;
  - Thread T_1 gets its input from other program parts; T_n provides its output to other program parts
- Parallelism despite of data dependencies



Taskpools and Producer-Consumer

- **Taskpool:** *Data structure* managing program parts in form of *functions* (tasks) that have to by *executed*; execution by a fixed number of threads, that access the taskpool for extraction and storage of tasks.
- **Producer-Consumer:** Producer threads create data and consumer threads use the data; common *data structure* of fixed size of the storage of *data*.





Topic overview

- Models for parallel systems
- Parallelization of programs
- Levels of parallelism
- Data distribution for arrays
- Information exchange
- Parallel Matrix-Vector Multiplication

Data distribution for arrays

- Data distribution, data decomposition, and data partitioning: data are partitioned into smaller pieces that are distributed to the processor for execution
  - **Distributed memory:** data assigned to a processor are stored in the local memory and can only be accessed by this processor (owner)
  - **Shared memory:** data are stored in the same shared memory and processors access different data according to the data distribution pattern → no access conflicts
- In the following:
  - data distribution for one-dimensional arrays
  - data distribution for two-dimensional arrays

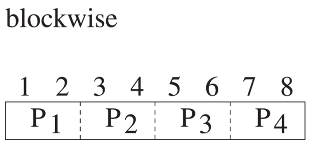
Data distribution for one-dimensional arrays

Blockwise data distribution for p processors:

- Array $v = (v_1,...,v_n)$ of length n
- Decomposition of array v into p blocks with $\lceil n/p \rceil$ consecutive elements each:
  - Block j contains the consecutive elements with indices: $(j-1) \cdot \lceil n/p \rceil + 1, ..., j \cdot \lceil n/p \rceil$ for $1 \le j < p$
  - Block p contains the elements with indices: $(j-1) \cdot \lceil n/p \rceil + 1, ..., n$
- Block j is assigned to processor j, $(1 \le j \le p)$.

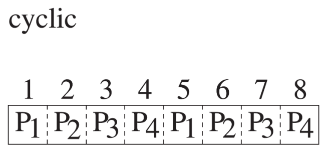
Example: For $n = 14$ and $p = 4$

- P1: v_1, v_2, v_3, v_4,
- P2: v_5, v_6, v_7, v_8,
- P3: v_9, v_{10}, v_{11}, v_{12},
- P4: v_{13}, v_{14}.



Cyclic data distribution

- Elements of an array $v = (v_1,...,v_n)$ are assigned to p processors in a **round robin** way, i.e., v_i is assigned to processor $P_{(i-1)} \bmod p + 1$, $i = 1,...,n$
- Processor P_j owns the array elements $j, j+p, \ldots, j+p \cdot (\lceil n/p \rceil - 1)$, for $j \le n \bmod p$
- Processor P_j owns the array elements $j, j+p, \ldots, j+p \cdot (\lceil n/p \rceil - 2)$, for $n \bmod p < j \le p$.

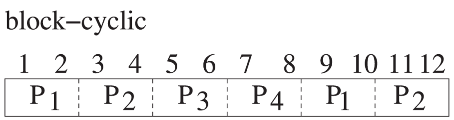


Example: For $n = 14$ and $p = 4$ the following results:

- $n \bmod p = 14 \bmod 4 = 2$
- For $1 \le j \le 2$: P_j owns array elements j, j+4, j+4∗2, j+4∗(4−1)
  For $2 < j \le 4$: Pj owns array elements j, j+4, j+4∗(4−2)
  - P1: v1, v5, v9, v13,
  - P2: v2, v6, v10, v14,
  - P3: v3, v7, v11,
  - P4: v4, v8, v12.

Block-cyclic data distribution

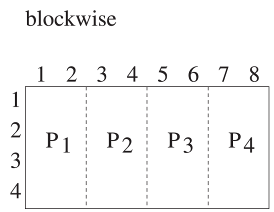
- Combination of the blockwise and the cyclic distribution
- Subdivision of array $v = (v_1,...,v_n)$ into blocks of size b; usually $b \ll n/p$.
- Cyclic distribution of the blocks to the processors $P_1,...,P_p$.

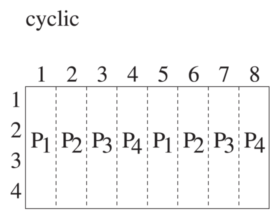


Data distribution for two-dimensional arrays

Distribution in only one of the two dimensions

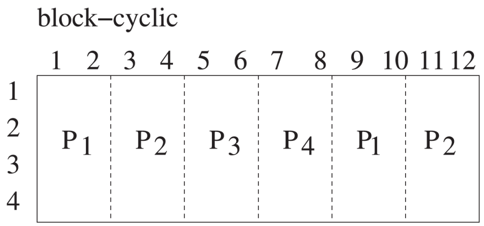
- **blockwise columnwise** (or blockwise rowwise):
  - A block of contiguous columns (or rows) of equal size; block i is assigned to P_i, $i = 1, \ldots, p$
- **cyclic columnwise** (or cyclic rowwise):
  - round robin distribution of columns (or rows) to processors





Data distribution for two-dimensional arrays

- **block-cyclic columnwise** (or rowwise):
  - blocks of contiguous columns (or rows) are assigned to the processors in a cyclic way to processors P_1 , … , P_p



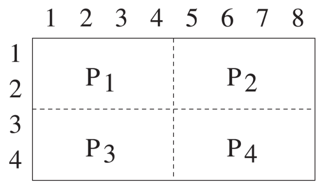
Checkerboard distribution of two-dimensional arrays

- The processors are arranged in a virtual mesh of size p_1 x p_2 = p
- Distribution of the data along both dimensions
- Distribution of the elements of an array of size $n_1 \times n_2$ in blockwise, cyclic, and block-cyclic checkerboard pattern

Checkerboard distribution of two-dimensional arrays

Blockwise checkerboard distribution:

- Decomposition of the array into p_1 x p_2 = p blocks
- Block (i,j), $1 \le i \le p_1$, $1 \le j \le p_2$ contains the elements (k,l) with $k = (i - 1) \cdot \lceil n_1/p_1 \rceil + 1, \ldots, i \cdot \lceil n_1/p_1 \rceil$ and $l = (j - 1) \cdot \lceil n_2/p_2 \rceil + 1, \ldots, j \cdot \lceil n_2/p_2 \rceil$.
- Block (i,j) is assigned to processor (i,j) in the processor grid



Checkerboard distribution of two-dimensional arrays

Cyclic checkerboard distribution:

- Array element (k,l) is assigned to the processor with mesh position $((k-1) \bmod p_1 + 1, (l-1) \bmod p_2 + 1)$.
- The processor at position (i,j) owns all array elements (k,l) with $k = i+s \cdot p_1$ and $l = j+t \cdot p_2$ for $0 \le s < n_1/p_1$ and $0 \le t < n_2/p_2$.

	1	2	3	4	5	6	7	8
1	P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2
2	P_3	P_4	P_3	P_4	P_3	P_4	P_3	P_4
3	P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2
4	P_3	P_4	P_3	P_4	P_3	P_4	P_3	P_4

Checkerboard distribution of two-dimensional arrays

Block-cyclic checkerboard distribution:

- Decomposition of the array into blocks of size $b_1 \times b_2$
- Array element (m,n) belongs to block (k,l) with $k = \lceil m/b_1 \rceil$ and $l = \lceil n/b_2 \rceil$.
- Block (k,l) is assigned to processor at mesh position $((k-1) \bmod p_1 + 1, (l-1) \bmod p_2 + 1)$.
- Special cases:
  $b_1 = b_2 = 1$ cyclic checkerboard distribution
  $b_1 = n_1/p_1$ and $b_2 = n_2/p_2$ blockwise checkerboard distribution

	1 2	3 4	5 6	7 8	9 10	11 12
1 2	P_1	P_2	P_1	P_2	P_1	P_2
3 4	P_3	P_4	P_3	P_4	P_3	P_4

Topic overview

- Models for parallel systems
- Parallelization of programs
- Levels of parallelism
- Data distribution for arrays
- Information exchange
- Parallel Matrix-Vector Multiplication

Information exchange

- The **information exchange** between processors of a parallel systems depends on the **organization of the memory subsystem**:
  - shared address space: shared variables
  - distributed address space: explicit communication operations
- **Shared variables:** Concurrent accesses through several processor to the same address is protected by *synchronization operations*
  → Serialization of concurrent accesses
  → Prevention of race conditions
  Simple synchronization by using (lock/unlock).
- **Communication operations:** Information exchange by **sending messages** (message passing);
  - Differentiation between *point-to-point* communication and *global* communication

Overview of communication operations: Send-Receive / Broadcast

- **Point-to-Point transfer:** A processor P_i (sender) sends a message to another processor P_j (receiver).
  - The **sender** executes a send operation (with the specification of a *send buffer*) and with the identification number of the receiver.
  - The **receiver** executes a corresponding receive operation with the specification of an *receive buffer* and with the specification of the identification number of the sender.
- **Single Broadcast:** A specific processor P_i (root) sends the same message to all other processors. Depiction:

$$\begin{array}{ll} P_1 : \boxed{x} & \\ P_2 : \boxed{-} & \\ \vdots & \\ P_p : \boxed{-} & \end{array} \quad \stackrel{broadcast}{\Longrightarrow} \quad \begin{array}{l} P_1 : \boxed{x} \\ P_2 : \boxed{x} \\ \vdots \\ P_p : \boxed{x} \end{array}$$

Overview of communication operations: Reduction / single-accumulation

- **Single-Accumulation Operation:** *Each process* sends a message to a specific processor P_i (*root*) with data of the same type.
- The messages are combined elementwise with a specific **reduction operation**.
  → the result on the root process P_i is a single (composed) message.
- Each process specifies a buffer with the data to be combined and the reduction operation which to be used.

$$\begin{array}{ll} P_1 : \boxed{x_1} \\ P_2 : \boxed{x_2} \\ \vdots & \\ P_p : \boxed{x_p} \end{array} \stackrel{accumulation}{\Longrightarrow} \begin{array}{l} P_1 : \boxed{x_1 + x_2 + \ldots + x_p} \\ P_2 : \boxed{x_2} \\ \vdots \\ P_p : \boxed{x_p} \end{array}$$

Overview of communication operations: Gather

- **Gather:** *Each* process sends to a specific processor (*root*) a message. The root processor collects the messages *without any reduction*.
- Each process specifies a buffer storing the data to be sent. The root process specifies an *additional* buffer for the collected messages.

$$\begin{array}{l} P_1 : \boxed{x_1} \\ P_2 : \boxed{x_2} \\ \vdots \\ P_p : \boxed{x_p} \end{array} \stackrel{gather}{\Longrightarrow} \begin{array}{l} P_1 : \boxed{x_1 \parallel x_2 \parallel \ldots \parallel x_p} \\ P_2 : \boxed{x_2} \\ \vdots \\ P_p : \boxed{x_p} \end{array}$$

Overview of communication operations: Scatter

- **Scatter:** A specific processor P_i (*root*) sends to the other processors a message, which might be different for each receiver.

$$\begin{array}{l} P_1 : \boxed{x_1 \parallel x_2 \parallel \ldots \parallel x_p} \\ P_2 : \boxed{-} \\ \vdots \\ P_p : \boxed{-} \end{array} \stackrel{scatter}{\Longrightarrow} \begin{array}{l} P_1 : \boxed{x_1} \\ P_2 : \boxed{x_2} \\ \vdots \\ P_p : \boxed{x_p} \end{array}$$

Overview of communication operations: Multi-Broadcast

- **Multi-Broadcast Operation:** Each processor executes a single broadcast operation, i.e., *each* processor sends *each other* processor the *same message*.
  - Contrary, each processor receives a message from each other processor, where the different receivers receive from one sender the same message.
  - Note: There is *no specific* root process

$$\begin{array}{l} P_1 : \boxed{x_1} \\ P_2 : \boxed{x_2} \\ \vdots \\ P_p : \boxed{x_p} \end{array} \stackrel{multi-broadcast}{\Longrightarrow} \begin{array}{l} P_1 : \boxed{x_1 \parallel x_2 \parallel \ldots \parallel x_p} \\ P_2 : \boxed{x_1 \parallel x_2 \parallel \ldots \parallel x_p} \\ \vdots \\ P_p : \boxed{x_1 \parallel x_2 \parallel \ldots \parallel x_p} \end{array}$$

Overview of communication operations: Multi-Accumulation

- **Multi-Accumulation Operation:** *Each* processor executes a single accumulation operation, i.e., *each process* provides for *each other* process a possible different message.
  - The messages specific for each receiver are combined with a *reduction operation*, such that each receiver gets a combined message.
  - Note: There is *no specific* root process

$$
\begin{array}{ll}
P_1: & \boxed{x_{11} \parallel x_{12} \parallel \ldots \parallel x_{1p}} \\
P_2: & \boxed{x_{21} \parallel x_{22} \parallel \ldots \parallel x_{2p}} \\
\vdots & \\
P_p: & \boxed{x_{p1} \parallel x_{p2} \parallel \ldots \parallel x_{pp}}
\end{array}
\quad \overset{multi-accumulation}{\Longrightarrow} \quad
\begin{array}{ll}
P_1: & \boxed{x_{11} + x_{21} + \ldots + x_{p1}} \\
P_2: & \boxed{x_{12} + x_{22} + \ldots + x_{p2}} \\
\vdots & \\
P_p: & \boxed{x_{1p} + x_{2p} + \ldots + x_{pp}}
\end{array}
$$

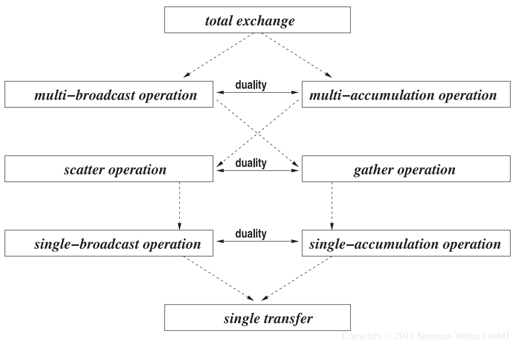
Overview of communication operations: Total Exchange

- **Total Exchange:** *Each* processor send to *each other* processor a possibly *different message*, without using a reduction operation, i.e., each processor executes a scatter operation.
  - Contrary, each processor receives from each other processor a possibly different messages, i.e., each processor executes a gather operation.
  - Note: There is *no specific* root process

$$
\begin{array}{ll}
P_1: & \boxed{x_{11} \parallel x_{12} \parallel \ldots \parallel x_{1p}} \\
P_2: & \boxed{x_{21} \parallel x_{22} \parallel \ldots \parallel x_{2p}} \\
\vdots & \\
P_p: & \boxed{x_{p1} \parallel x_{p2} \parallel \ldots \parallel x_{pp}}
\end{array}
\quad \overset{total\ exchange}{\Longrightarrow} \quad
\begin{array}{ll}
P_1: & \boxed{x_{11} \parallel x_{21} \parallel \ldots \parallel x_{p1}} \\
P_2: & \boxed{x_{12} \parallel x_{22} \parallel \ldots \parallel x_{p2}} \\
\vdots & \\
P_p: & \boxed{x_{1p} \parallel x_{2p} \parallel \ldots \parallel x_{pp}}
\end{array}
$$

Hierarchy of Communication Operations

- The communication operations result from a *stepwise specialization* from the most general operation (total exchange)
  → Representation as a hierarchy is possible:



Topic overview

- Models for parallel systems
- Parallelization of programs
- Levels of parallelism
- Data distribution for arrays
- Information exchange
- Parallel Matrix-Vector Multiplication

Parallel Matrix-Vector Multiplication

- Multiplication of
  - a dense $n \times m$-matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\mathbf{A} = (a_{ij})_{i=1,\dots,n, j=1,\dots,m}$
  - and a vector $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{b} = (b_1, \dots, b_m)$
  - with result vector $\mathbf{c} = (c_1, \dots, c_n) \in \mathbb{R}^n$

$$c_i = \sum_{j=1}^{m} a_{ij} b_j, \quad i = 1, \dots, n,$$

- There exist two implementation variants differing in the loop order over i and j, $i, j = 1, \dots, n$.
  - computation of n scalar products.
  - linear combination of columns

Matrix-Vector Product using scalar products

- Computation of n scalar products (vector-vector-multiplication of rows $\mathbf{a}_1, \dots, \mathbf{a}_n$ von $\mathbf{A}$ with vector $\mathbf{b}$:

$$\mathbf{A} \cdot \mathbf{b} = \begin{pmatrix} (\mathbf{a}_1, \mathbf{b}) \\ \vdots \\ (\mathbf{a}_n, \mathbf{b}) \end{pmatrix},$$

- A scalar product is defined as:
  $(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{m} x_j y_j$
  for two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ with $\mathbf{x} = (x_1, \dots, x_m)$ and $\mathbf{y} = (y_1, \dots, y_m)$
- Corresponding sequential algorithm in C-notation:

```
for (i=0; i<n; i++) c[i] = 0;
for (i=0; i<n; i++)
   for (j=0; j<m; j++)
      c[i] = c[i] + A[i][j] * b[j];
```

  with a two-dimensional array A and one-dimensional array b, c. (The indices start with 0 as usual in C).

Matrix-Vector Product based on linear combinations

- Computation of a linear combination of columns $\tilde{\mathbf{a}}_1, \dots, \tilde{\mathbf{a}}_m$ of $\mathbf{A}$, with coefficients $(b_1, \dots, b_m)$, i.e.

$$\mathbf{A} \cdot \mathbf{b} = \sum_{j=1}^{m} b_j \tilde{\mathbf{a}}_j \, .$$

- Corresponding sequntial allgorithm in C-notation:

```
for (i=0; i<n; i++) c[i] = 0;
for (j=0; j<m; j++)
   for (i=0; i<n; i++)
      c[i] = c[i] + A[i][j] * b[j] ;
```

  - For each $j = 0, \dots, n-1$ a column $\tilde{\mathbf{a}}_j$ is added to the linear combination.
  - This sequential program is equivalent to the previous one, since the loops over i and j can be exchanged due to data independence.

Parallel Matrix vector-Product

The two sequential representations give rise to two different parallel implementations

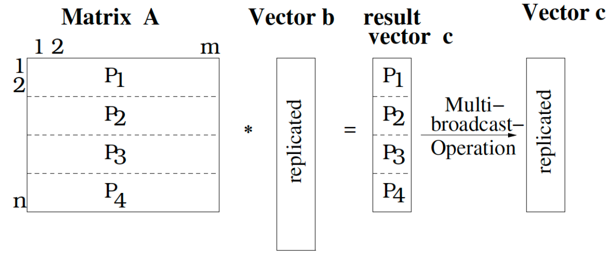
(a) Row-oriented representation of matrix A and the computation of n scalar products:
Parallel implementation in which each processor computes about n/p scalar products (p denotes the number of processors)

(b) Column-oriented representation of matrix A and computation of a linear combination:
Parallel implementation in which each processor computes a part of the linear combination using about n/p column vector

Matrix-Vector Multiplication: distribution of rows (1)

- Matrix-vector multiplication based on **scalar products**
- Set of p processors P_k, $k = 1, \ldots, p$ with **distributed memory**
- Each processor computes that scalar product for which it own the corresponding row of matrix **A**.
- Data distribution:
  - Row-oriented blockwise distribution of matrix **A**:: Processor P_k stores the rows $\mathbf{a}_i$ for $i = n/p \cdot (k-1) + 1, \ldots, n/p \cdot k$ in its local memory, $k = 1, \ldots, p$
  - Result vector $\mathbf{c} = (c_1, \ldots, c_n)$ has a block wise distribution.

Matrix-Vector-Multiplication: distribution of row (2)



- When the matrix-vector-product is used within a large algorithm like iteration methods, a certain distribution of c might be required. Example: Vector c should have the same distribution as b.
  - Each processor P_k, $k = 1, \ldots, p$ sends its block $(c_{n/p\cdot(k-1)+1}, \ldots, c_{n/p\cdot k})$ to all other processors by a **multi-broadcast** operation.

Matrix-Vector Multiplication: distribution of rows (3)

- Parallel implementation SPMD program for processor P_k mit $k = 1, \ldots, p$
- Row-wise distribution
  - Each processor has a local array `local_A` of size `local_n` $\times$ `m`
  - Processor P_k stores the following data:

$$\mathtt{local\_A[i][j]} = \mathtt{A[i+(k-1)*n/p][j]}$$

- Result vector: local array `local_c` of size `local_n`
- After the multi-broadcast operation

$$\mathtt{c[i+(k-1)*n/p]} = \mathtt{local\_c[i]}$$

Matrix-Vector-Multiplication: distribution of rows (4)

- Program fragment in C-notation and MPI comunication operation

```
local_n = n/p;
for (i=0; i<local_n; i++) local_c[i] = 0;
for (i=0; i<local_n; i++)
    for (j=0; j<m; j++)
        local_c[i] = local_c[i] + local_A[i][j] * b[j];
MPI_Allgather(local_c,local_n,MPI_DOUBLE,
              global_c,local_n,MPI_DOUBLE,comm);
```

Parallel Matrix-Vector-Multiplication: Shared Memory

- SPMD program with a distribution of the computations
- No explicit distribution of data
- Each process accesses a different part of the matrix A
- Each process computes n/p components of the result vector c and uses the corresponding n/p rows of matrix A
  $\rightarrow$ No access conflict

Parallel Matrix-Vector-Multiplication: Shared Memory

- SPMD program with private variable k holding the processor ID.

```
local_n = n/p;
for (i=0; i<local_n; i++) c[i+(k-1)*local_n] = 0;
for (i=0; i<local_n; i++)
    for (j=0; j<m; j++)
        c[i+(k-1)*local_n] =
        c[i+(k-1)*local_n] + A[i+(k-1)*local_n][j] * b[j];
synch();
```