UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Design models for parallel programs

Reorganizing computations

- **Task decomposition**: computations are a set of independent tasks that threads can execute in any order.

- **Data decomposition**: the application processes a large collection of data and can compute every element of the data independently.

Task decomposition

- Tasks must be assigned to threads for execution.

- We can allocate tasks to threads in two different ways: static scheduling or dynamic scheduling.

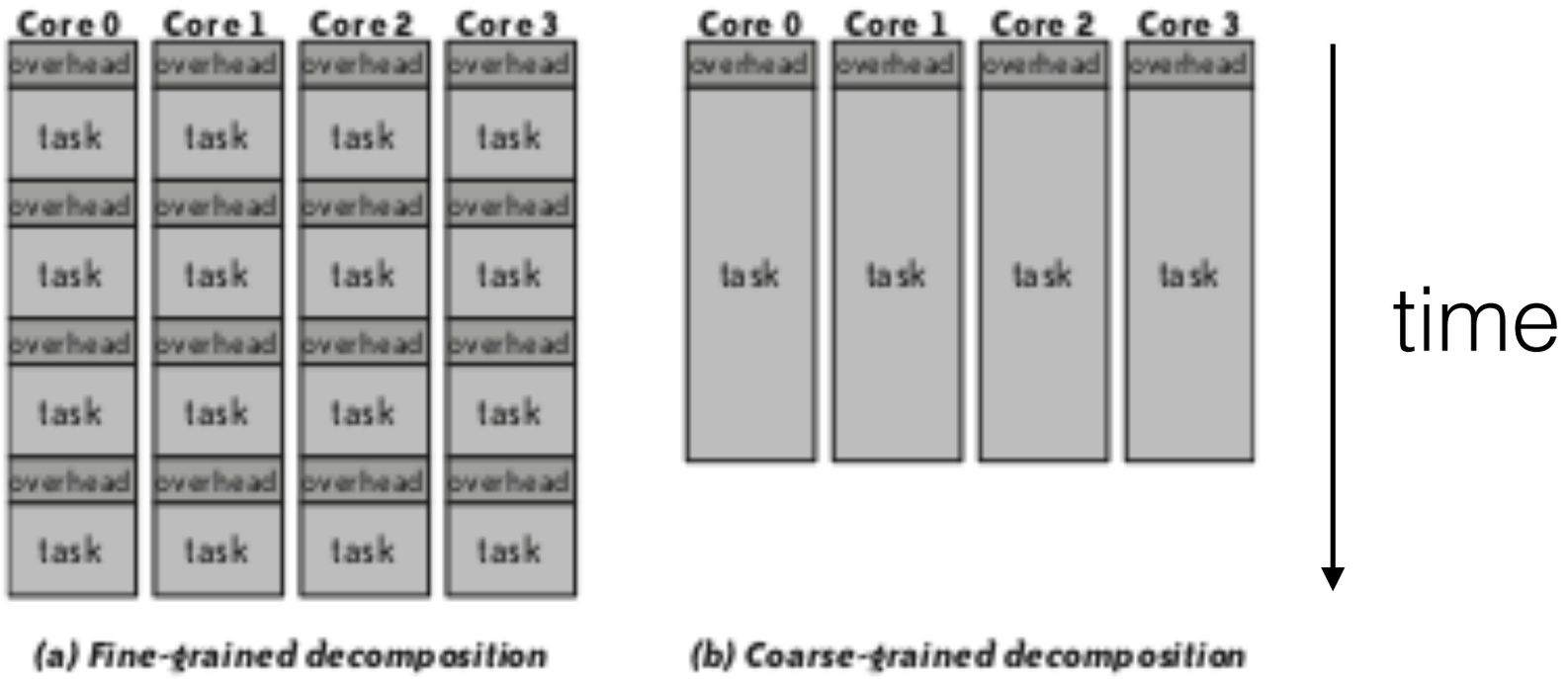
  - **static scheduling**: the division of labor is known at the outset of the computation and doesn't change during the computation.

  - **dynamic scheduling**: assign tasks to threads as the computation proceeds. The goal is to try to balance the load as evenly as possible between threads. Different methods to do this, but they all require a set of many more tasks than threads.

Decomposition criteria

- There should be at least as many tasks as there will be threads.
  Goal: avoid idle threads (or cores) during the execution of the application

- The amount of computation within each task (granularity) must be large enough to offset the overhead that will be needed to manage the tasks and the threads.
  Goal: avoid to write an algorithm that is worse than the sequential version



(a) Fine-grained decomposition *(b) Coarse-grained decomposition*

Data decomposition

- We may identify that execution of a serial program is dominated by a sequence od operations on all elements of one or more large data structures.
  If these are independent we can divide the data assigning portions (*chunks*) to different tasks.

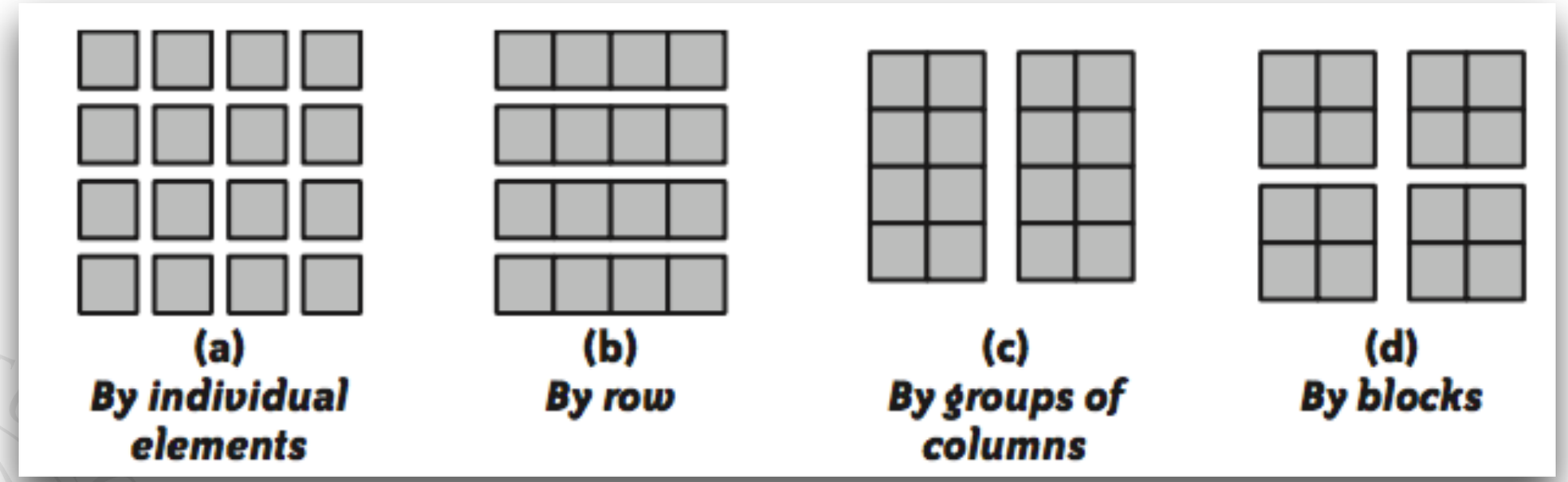
- Key problems:
  - How to divide the data into chunks? Consider shape and granularity…
  - How to ensure that the tasks for each chunk have access to all data required for computations? A thread may need data contained in different thread…
  - How are the data chunks assigned to threads?

Data decomposition

- We may identify that execution of a serial program is dominated by a sequence od operations on all elements of one or more large data structures.
  If these are independent we can divide the data assigning portions (*chunks*) to different tasks.

- Key problems:

- How to divide the data into chunks? Consider shape and granularity…

- 

-

Data decomposition

- We may identify that execution of a serial program is dominated by a sequence od operations on all elements of one or more large data structures.
  If these are independent we can divide the data assigning portions (*chunks*) to different tasks.

- Key problems:
  - How to divide the data into chunks? Consider shape and granularity…
  - How to ensure that the tasks for each chunk have access to all data required for computations? A thread may need data contained in different thread…
  - How are the data chunks assigned to threads?

Data decomposition

- We may identify that execution of a serial program is dominated by a sequence od operations on all elements of one or more large data structures.
  If these are independent we can divide the data assigning portions (*chunks*) to different tasks.

- Key problems:
  - How to divide the data into chunks? Consider shape and granularity
  - How are the data chunks assigned to threads?

Tasks that are associated with the data chunks can be assigned to threads statically or dynamically. The latter is more complex (coordination) and requires (many) more tasks than threads.

Ensure that the amount of computation that goes along with that chunk is sufficient to warrant breaking out that data as a separate chunk.

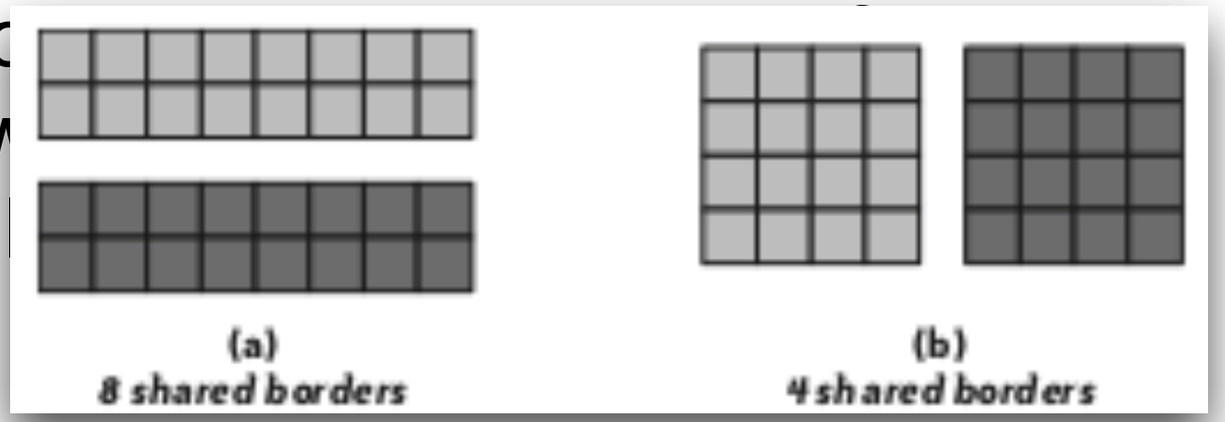
Chunk shape

- The shape of a chunk determines what the neighboring chunks are and how any exchange of data will be handled during the course of the chunk's computations. Reducing the size of the overall border reduces the amount of exchange data required for updating local data elements; reducing the total number of chunks that share a border with a given chunk will make the exchange operation less complicated to code and execute.

- A good rule of thumb is to try to maximize the volume-to-surface ratio. The volume defines the granularity of the computations, and the surface is the border of chunks that require an exchange of data.

Chunk shape

- The shape of a chunk determines what the neighboring chunks are and how any exchange of data will be handled during the course of the chunk's computations. Reducing the size of the overall border reduces the amount of exchange data required for updating local data elements; reduc
that share a border w
exchange operation
execute.



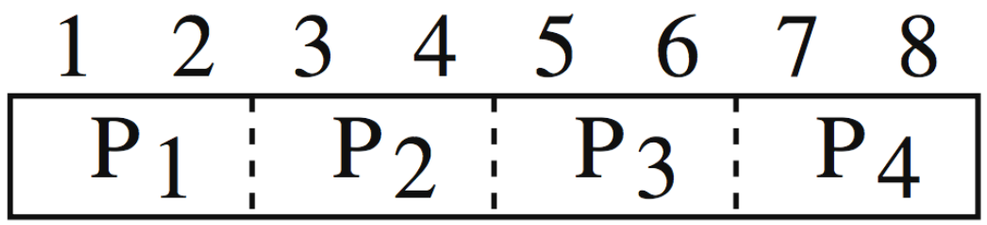
- A good rule of thumb is to try to maximize the volume-to-surface ratio. The volume defines the granularity of the computations, and the surface is the border of chunks that require an exchange of data.

Decomposition example: Data Distributions for Arrays

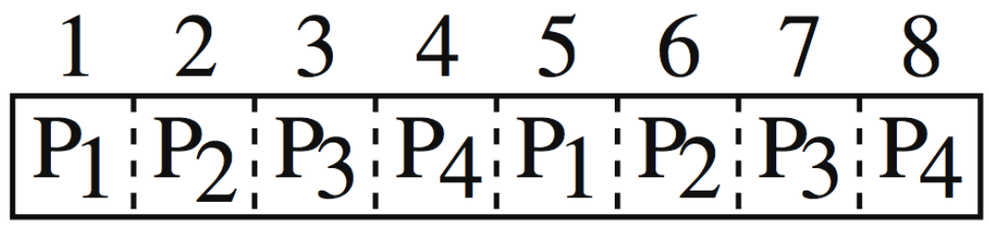
- Let us consider a set of processes $P = \{P_1, ..., P_p\}$
- 1 dimensional arrays
- **Blockwise distribution**: cuts an array *v* of *n* elements into *p* blocks with $\lceil n/p \rceil$ consecutive elements each.
- **Cyclic distribution**: assigns elements to processes in round-robin way, so that v_i is assigned to $P_{(i-1) \bmod p+1}$
- **Block-cyclic**: combination of the two

Decomposition example: Data Distributions for Arrays

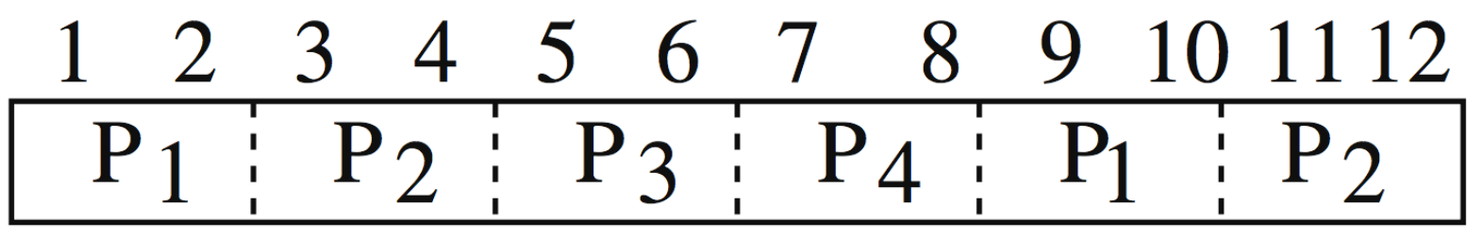
blockwise



cyclic



block–cyclic



in round-robin way, so that v_i is assigned to $P_{(i-1) \bmod p+1}$

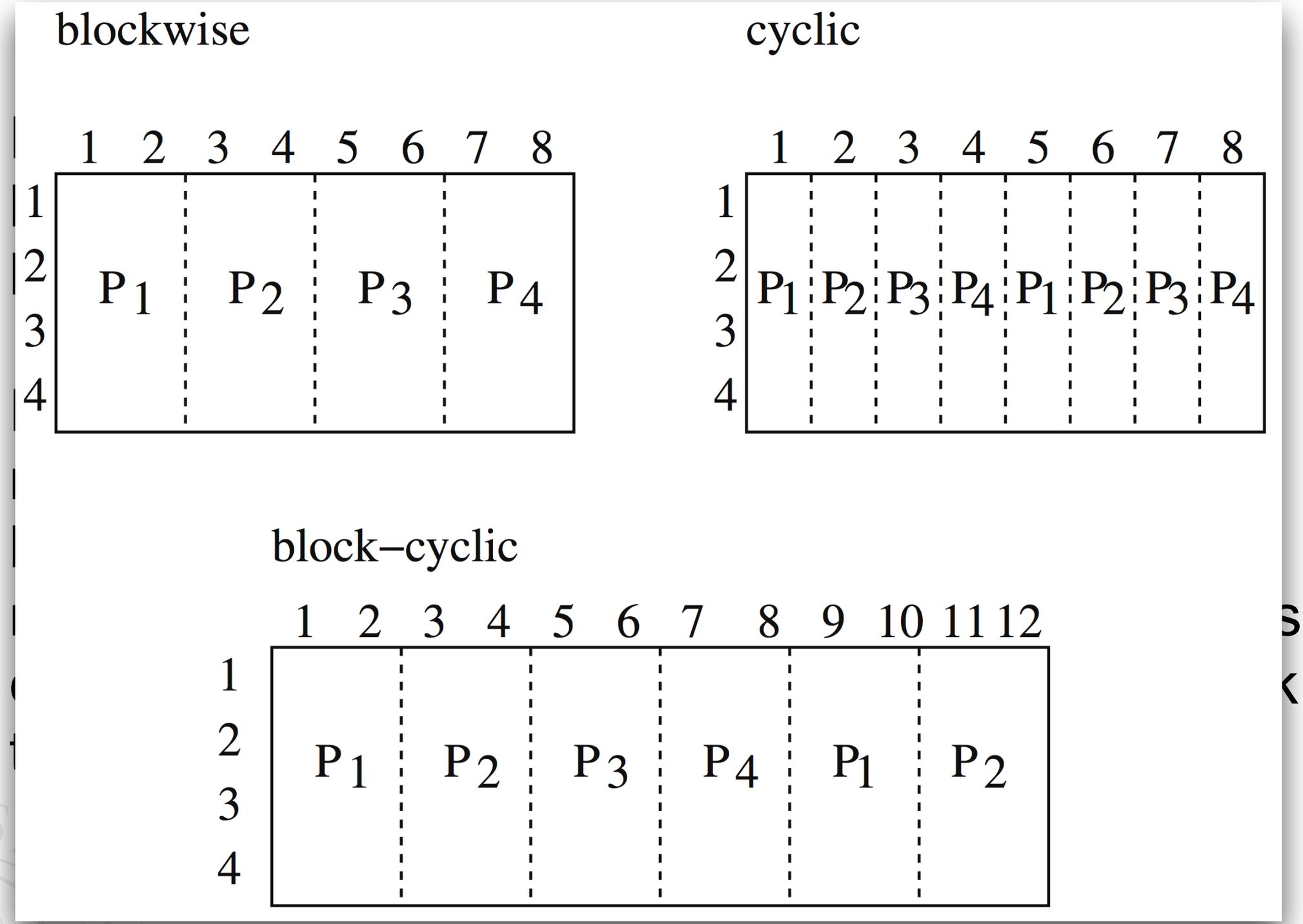
- **Block-cyclic**: combination of the two

Decomposition example: Data Distributions for Arrays

- For two-dimensional arrays, combinations of blockwise and cyclic distributions in only one or both dimensions are used.

- For the distribution in one dimension, columns or rows are distributed in a block- wise, cyclic, or block–cyclic way. The blockwise columnwise (or rowwise) distribution builds *p* blocks of contiguous columns (or rows) of equal size and assigns block i to processor P_i , $i = 1, \ldots, p$.

Decomposition example: Data Distributions for Arrays



Decomposition example: Data Distributions for Arrays

- A distribution of array elements of a two-dimensional array of size $n_1 \times n_2$ in both dimensions uses checkerboard distributions which distinguish between **blockwise cyclic** and **block–cyclic checkerboard** patterns.

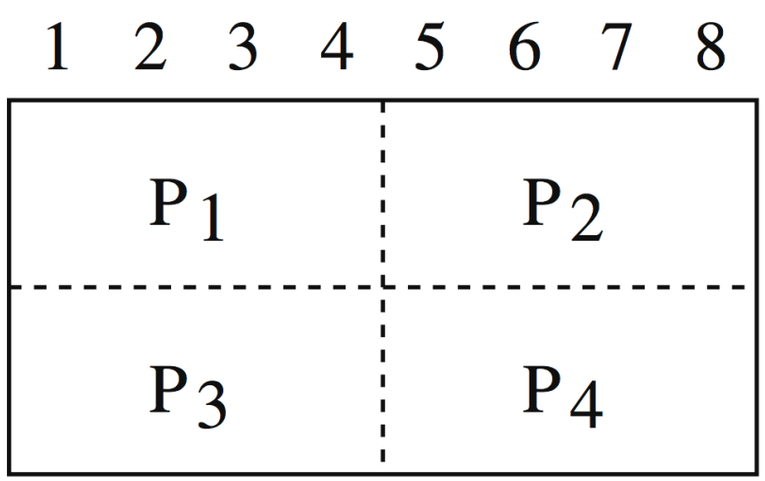
- The processors are arranged in a virtual mesh of size $p_1 \cdot p_2 = p$ where p_1 is the number of rows and p_2 is the number of columns in the mesh. Array elements (k,l) are mapped to processors $P_{i,j}$, $i = 1, \ldots, p_1$, $j = 1, \ldots, p_2$.

Decomposition example: Data Distributions for Arrays

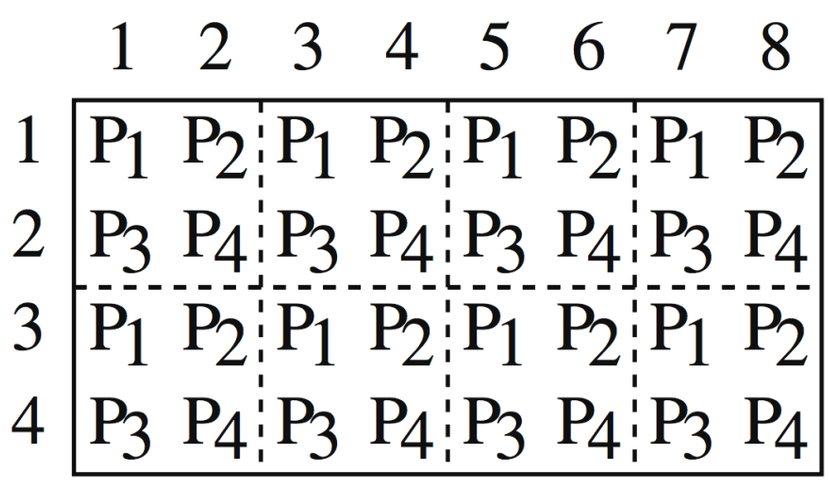
- A
  di ons
  us sh
  be
  cl

- Th of
  si nd
  p
  A
  P

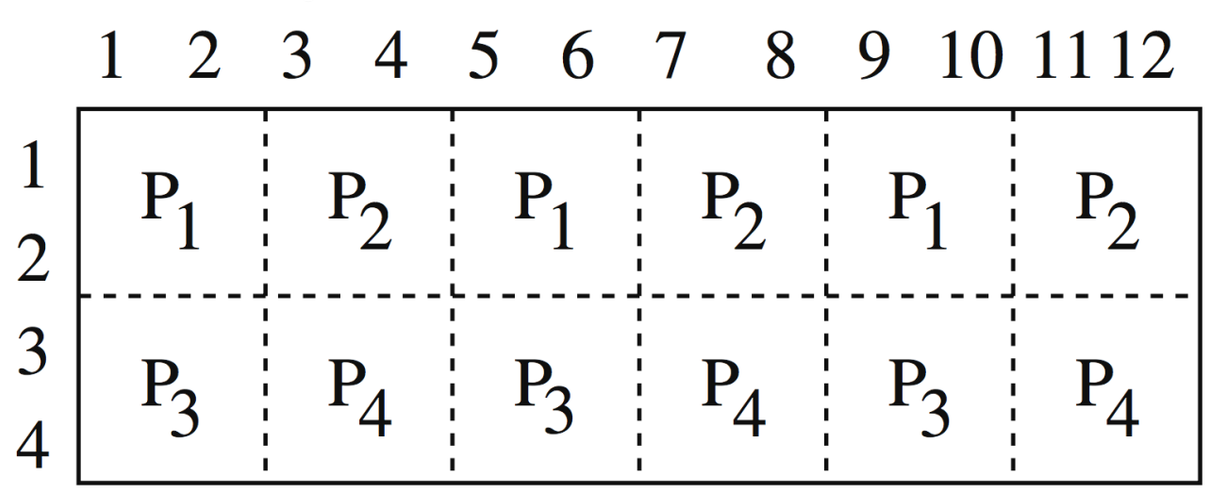
blockwise



cyclic



block–cyclic



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Important properties

Safety and Liveness

- The correctness (i.e. specification and verification of what a given program actually does) of parallel programs, by their very nature, is more complex than that of their sequential counterparts.
  - A modern computer is asynchronous: activities can be halted or delayed without warning by interrupts, preemption, cache misses, failures, and other events. Parallel computing multiplies all this.
- We are interested in two properties:
  - **Safety** Properties
    - Nothing bad happens ever
  - **Liveness** Properties
    - Something good happens eventually

Example

- If two processes need to use a common resource, and this resource can be used only by one process at a certain time, we need a protocol that allows to have these properties:

- **Mutual exclusion**: i.e. both processes never use the resource at the same time.
  This is a **safety** property.

- **No deadlock**: i.e. if one or both the processes want the resource then one gets it.
  This is a **liveness** property.

Example

- If two processes need to use a common resource, and this resource can be used only by one process at a certain time, we need a protocol that allows to have these properties:

- **Mutual exclusion**: i.e. both processes never use the resource at the same time.
  This is a **safety** property.

- **No deadlock**: i.e. if one or both the processes want the resource then one gets it.
  This is a **liveness** property.

Deadlock is used to denote that if processes are stuck then no amount of retry (backoff) will help, while **livelock** means backoff can help

Other properties

- **Starvation freedom**: i.e. if one of the processes wants the resource will it get it eventually ?
- **Waiting**: what happens if a processes is waiting for the other to release the resource, but the process controlling it fails to do so for some reason ?
  This is an example of **fault-tolerance**.
  A mutual exclusion problem implies waiting.

Communication properties

- Two kinds of communication occur naturally in concurrent systems:
  - **Transient communication** requires both parties to participate at the same time. (like speaking)
  - **Persistent communication** allows the sender and receiver to participate at different times. (like writing)
- A protocol capable of achieving mutual exclusion needs persistent communication.
- An interrupt is persistent communication: a process interrupts another setting a bit, the interrupted process will periodically check it, act and then reset the bit.

Communication properties

- Two kinds of communication occur naturally in concurrent systems:
  - **Transient communication** requires both parties to participate at the same time. (like speaking)
  - **Persistent communication** allows the sender and receiver to participate at different times. (like writing)
- A protocol capable of achieving mutual exclusion needs persistent communication.
- An interrupt is persistent communication: a process interrupts another setting a bit, the interrupted process will periodically check it, act and then reset the bit.

An interrupt still is not enough to solve mutual exclusion, though.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Rules of thumbs for designing parallel (multithreaded) applications

Identify Truly Independent Computations

- It’s obvious, but remind that you can’t execute anything concurrently unless the operations that would be executed can be run independently of each other.

- Check the dependencies (e.g. data or loop)

Examples of dependencies

- Recurrences: relations within loops feed information forward from one iteration to the next.

```
for (i = 1; i < N; i++)
    a[i] = a[i-1] + b[i];
```

- Induction variables: variables that are incremented on each trip through a loop, without having a one-to-one relation with loop iterator.

```
i1 = 4;
i2 = 0;
for (k = 1; k < N; k++) {
    B[i1++] = function1(k,q,r);
    i2 += k;
    A[i2] = function2(k,r,q);
}
```

Examples of dependencies

- Recurrences: relations within loops feed information forward from one iteration to the next.

```
for (i = 1; i < N; i++)
    a[i] = a[i-1] + b[i];
```

- Induction variables: variables that are incremented on each trip through a loop, without having a one-to-one relation with loop iterator.

```
i1 = 4;
i2 = 0;
for (k = 1; k < N; k++) {
    B[i1++] = function1(k,q,r);
    i2 += k;
    A[i2] = function2(k,r,q);
}
```

Even if `function1()` and `function2()` are independent, there's no way to transform this code for concurrency without transforming the array index increment expression with a calculation based only on k

Examples of dependencies

- Reduction: takes a collection (e.g. array) of data and reduces it to a single scalar through some combination. If the operation is associative and commutative it can be eliminated.

```
sum = 0;
big = c[0];
for (i = 0; i < N; i++) {
    sum += c[i];
    big = (c[i] > big ? c[i] : big); // maximum element
}
```

Examples of dependencies

In this case we have to compute **sum** and **max** so there's a solution:

1. divide the loop iterations among the threads to be used and simply compute partial results (sum and big in the preceding example) in local storage.
2. combine each partial result into a global result, taking care to synchronize access to the shared variables.

```
sum = 0;
big = c[0];
for (i = 0; i < N; i++) {
    sum += c[i];
    big = (c[i] > big ? c[i] : big); // maximum element
}
```

Examples of dependencies

- Loop-Carried Dependance: occurs when results of some previous iteration are used in the current iteration. **Recurrence** is a special case of a loop-carried dependence where the backward reference is the immediate previous iteration.

  Dividing such loop iterations into tasks presents the problem of requiring extra synchronization to ensure that the backward references have been computed before they are used in computation of the current iteration.

```
for (k = 5; k < N; k++) {
    b[k] = DoSomething(k);
    a[k] = b[k-5] + MoreStuff(k);
}
```

Examples of dependencies

Sometimes loop-carried dependance is not immediately visible, e.g. hidden by a variable:

```
wrap = a[0] * b[0];
for (i = 1; i < N; i++) {
    c[i] = wrap;
    wrap = a[i] * b[i];
    d[i] = 2 * wrap;
}
```

Examples of dependencies

Sometimes loop-carried dependance is not immediately visible, e.g. hidden by a variable:

```
wrap = a[0] * b[0];
for (i = 1; i < N; i++) {
    c[i] = wrap;
    wrap = a[i] * b[i];
    d[i] = 2 * wrap;
}
```

Luckily this case can be solved:

```
for (i = 1; i < N; i++) {
    wrap = a[i-1] * b[i-1];
    c[i] = wrap;
    wrap = a[i] * b[i];
    d[i] = 2 * wrap;
}
```

Implement Concurrency at the Highest Level Possible

- Identify the hotspots in the code, then examine if it's possible to parallelize the higher level.
  - This allows to parallelize with a larger granularity
    - E.g.: we may identify that the hotspot of a video coding application is related to coding macroblocks, but perhaps the application is required to encode many videos, so it's better to parallelize at the video processing level
- The objective of this rule is to find the highest level where concurrency can be implemented so that your hotspot of code will be executed concurrently.

Plan Early for Scalability to Take Advantage of Increasing Numbers of Cores

- Designing and implementing concurrency by data decomposition methods will give you more scalable solutions.

- Task decomposition solutions will suffer from the fact that the number of independent functions or code segments in an application is likely limited and fixed during execution.
After each independent task has a thread and core to execute on, increasing the number of threads to take advantage of more cores will not increase performance of the application.

Use of Thread-Safe Libraries Wherever Possible

- Check that the library you use is thread-safe. It may contain some shared variable that causes data races.

- A library function is thread-safe if it can be called by different threads concurrently, without performing additional operations to avoid race conditions.

Use the Right Threading Model

- There are different libraries and APIs to implement multi-threaded programs. Use the correct one for the task, i.e. do not go low-level and reinvent the wheel if it's not necessary.
  - E.g. in C++ you can use:
    - Pthreads
    - C++11 threads
    - OpenMP
    - Intel TBB
    - Cilk++

Never Assume a Particular Order of Execution

- Execution order of threads is nondeterministic and controlled by the OS scheduler:
  - no reliable way to predict their execution order
  - this is the motivation of the risk of data races
- Don’t try to enforce a particular order of execution unless it is absolutely necessary.
  Recognize those times when it is absolutely necessary, and implement some form of synchronization to coordinate the execution order of threads relative to each other.

Never Assume a Particular Order of Execution

Data races are a direct result of this scheduling nondeterminism.
Do not assume that one thread will write a value into a shared variable before another thread will read that value.

- no reliable way to predict their execution order
  - this is the motivation of the risk of data races

- Don’t try to enforce a particular order of execution unless it is absolutely necessary.
  Recognize those times when it is absolutely necessary, and implement some form of synchronization to coordinate the execution order of threads relative to each other.

Use Thread-Local Storage Whenever Possible or Associate Locks to Specific Data

- Synchronization is overhead: do not use it except to guarantee the correct answers are produced from the parallel execution.

- Use temporary work variables allocated locally to each thread.

- Use thread-local storage (TLS) APIs to enable persistence of data local to threads (similar to the concept of static variables in C functions).

- If the above two options are not viable then use shared and synchronized data.

Dare to Change the Algorithm for a Better Chance of Concurrency

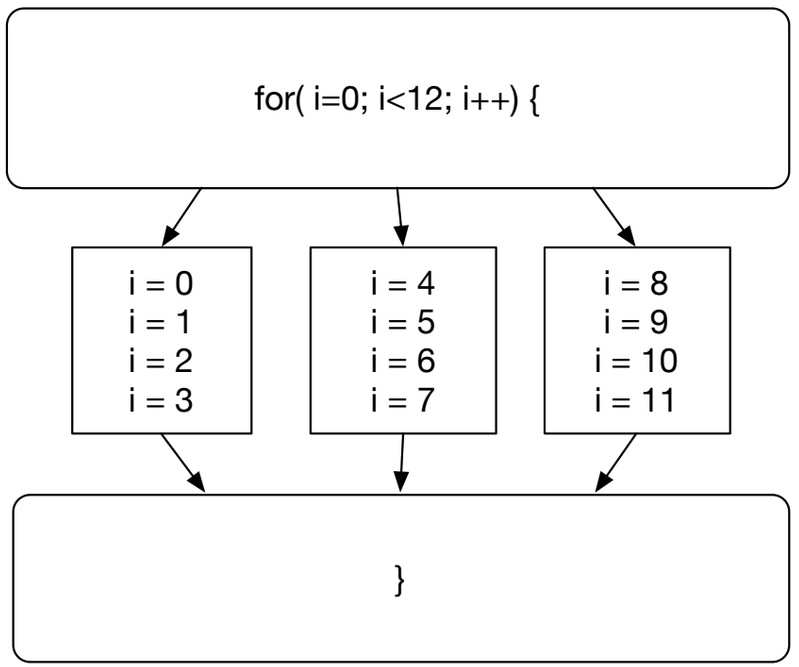
- Some algorithm with higher complexity may be more amenable to parallelization than other algorithms with better complexity.
- E.g. simple $O(n^3)$ matrix multiplication is easily parallelizable, while optimized algorithms like Strassen and Coppersmith-Winograd may be unpractical.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Design patterns for parallel programming

Loop level parallelism

- Many programs are expressed using iterative constructs:
  - Assign loop iteration to units of execution (i.e. threads and processes)
  - Especially good when code cannot be massively restructured



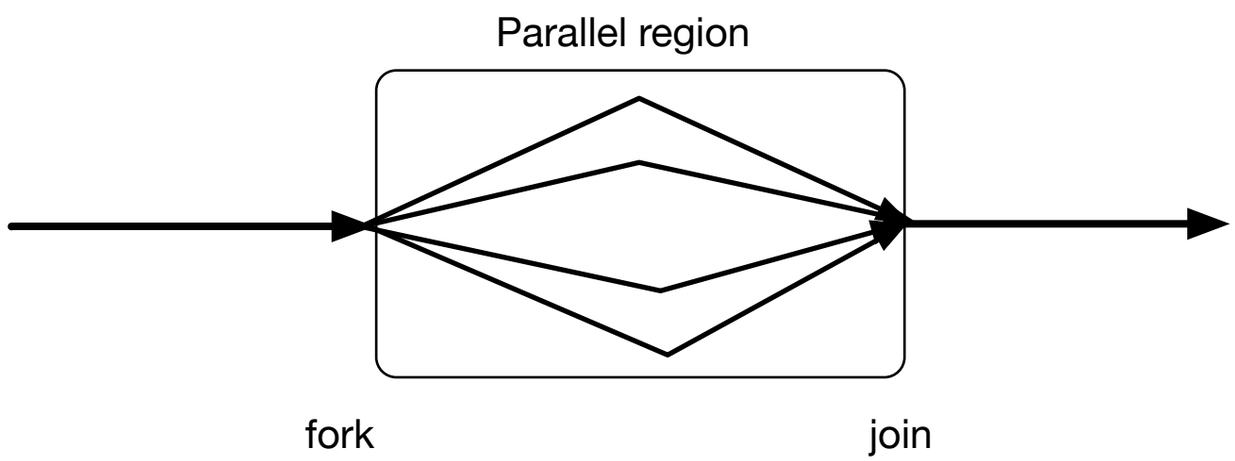
Task parallelism

- Task parallelism focuses on distributing tasks – concretely performed by processes or threads – across different parallel computing nodes.
  - Can be applied to shared and distributed memory systems.
  - In a multi-processor/multi-core system, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data.
    The threads may execute the same or different code.
    In the general case, different execution threads communicate with one another as they work.
    Communication usually takes place by passing data from one thread to the next as part of a workflow.

Fork-Join parallelism

- A main process/thread forks some number of processes/threads that then continue in parallel to accomplish some portion of the overall work.

- Parent tasks creates new task (fork) then waits until all they complete (join) before continuing on with the computation



Pipelining

- Special form of coordination of different threads in which data elements are forwarded from thread to thread to perform different processing steps.
  - can be considered as a special form of functional decomposition where the pipeline threads process the computations of an application algorithm one after another.
  A parallel execution is obtained by partitioning the data into a stream of data elements which flow through the pipeline stages one after another.

SPMD

- Single program, multiple data
- All units of execution execute the same program in parallel, but each has its own set of data.
  - Initialize
  - Obtain a unique identifier
  - Run the same program on each processor
  - Distribute data/tasks based on ID
  - Finalize

Mostly used in distributed memory systems, rather than shared memory…
…but it's still applicable, either using processes or threads

Master-worker

- A **master** process or thread set up a pool of **worker** processes of threads and a bag of tasks (often managed with a queue).

- The workers execute concurrently, with each worker repeatedly removing a tasks from the bag of the tasks.

- Workers request a new task as they finish their assigned work: load is automatically balanced between a collection of workers.

- Appropriate for “embarrassingly parallel problems”

- Other names: producer-consumer, master-slave

Master-worker

- A **master** process or thread set up a pool of **worker** processes of threads and a bag of tasks (often managed with a queue).

- The workers execute concurrently, with each worker repeatedly removing a tasks from the bag of the tasks.

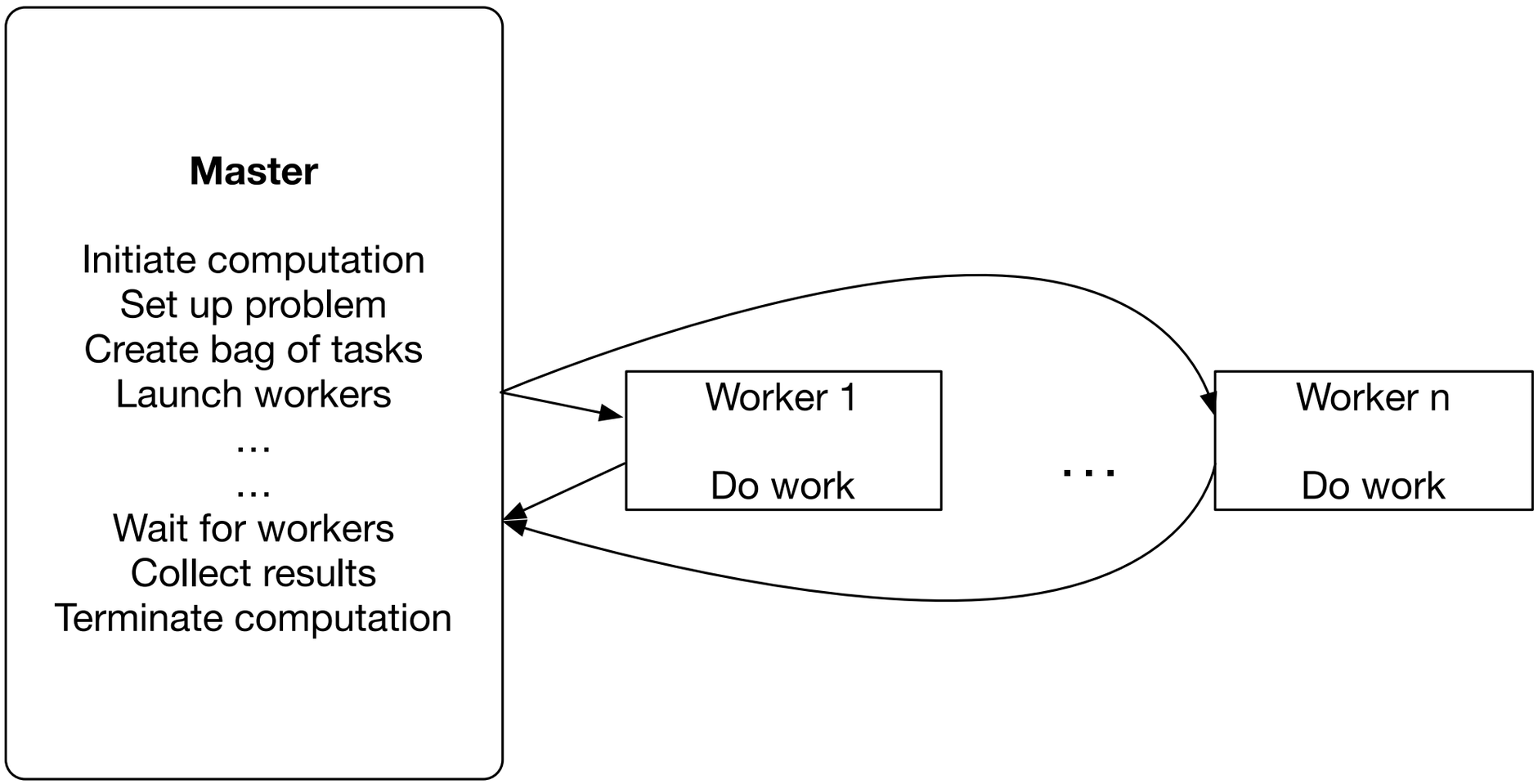
- Workers request a new task as they finish their assigned work: load is automatically balanced between a collection of workers.

- Appropriate for “embarrassingly parallel problems”

-

Remind: an “embarrassingly parallel problem”, is one for which little or no effort is required to separate the problem into a number of parallel tasks. This is often the case where there exists no dependency (or communication) between those parallel tasks.

Master-worker



Client-Server

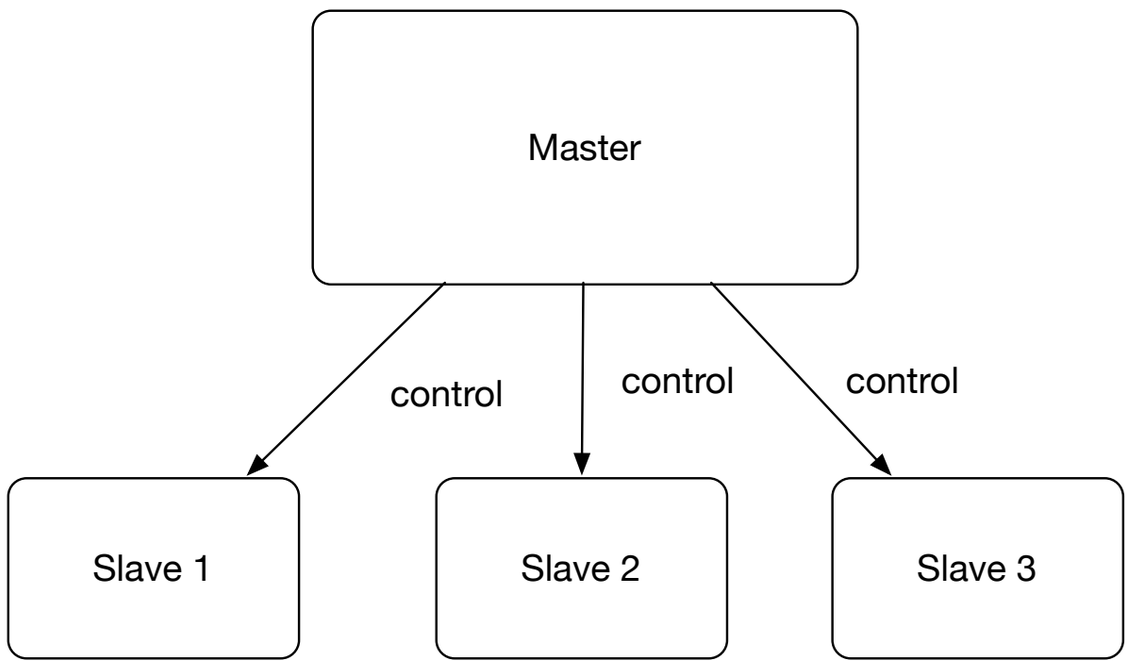
- The client–server model is similar to the general MPMD (multiple-program multiple-data) model.

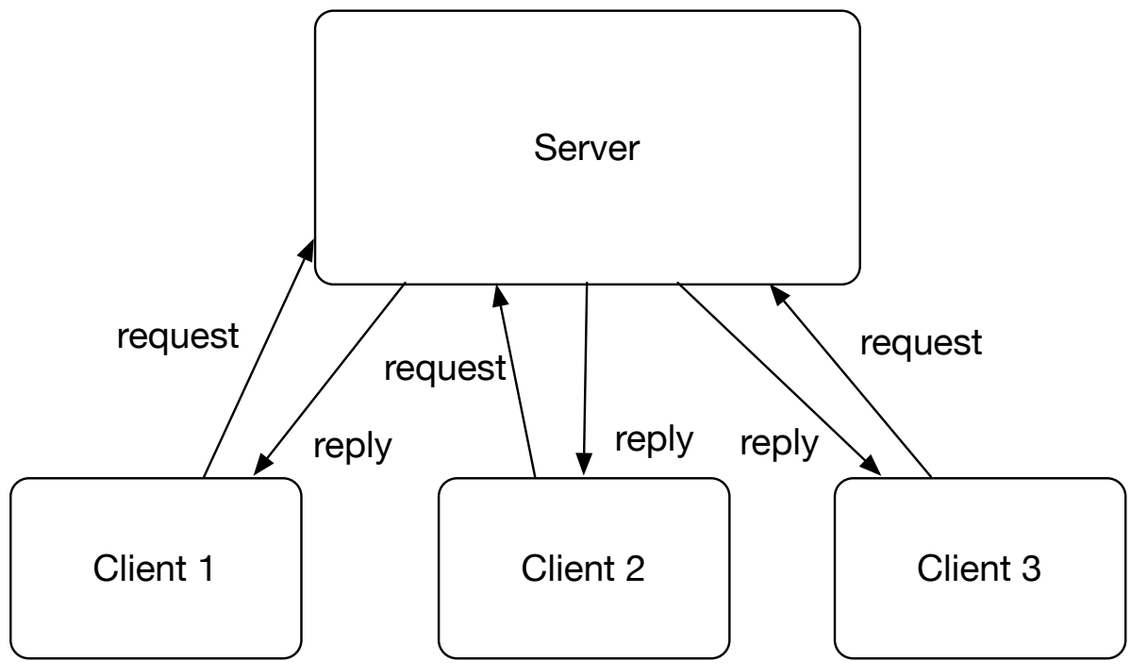
- This model originally comes from distributed computing

- Parallelism can be used by computing requests from different clients concurrently or even by using multiple threads to compute a single request if this includes enough work.

- There may be several server threads or the threads of a parallel program may play the role of both clients and servers, generating requests to other threads and processing requests from other threads.

Client-server vs Master-slave





Task pool

- A **task pool** is a data structure in which tasks to be performed are stored and from which they can be retrieved for execution. A task comprises computations to be executed and a specification of the data to which the computations should be applied. The computations are often specified as a function call.

- A fixed number of threads is used for the processing of the tasks. The threads are created at program start by the main thread and they are terminated not before all tasks have been processed. For the threads, the task pool is a common data structure which they can access to retrieve tasks for execution

- Access to the task pool must be synchronized to avoid race conditions.

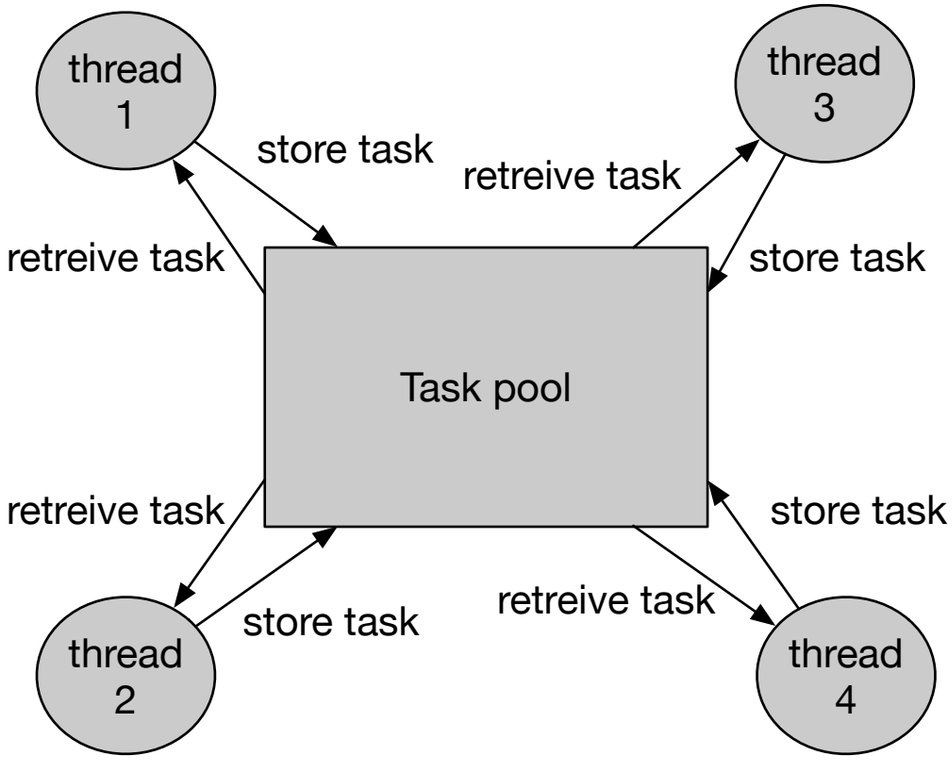
Producer-Consumer

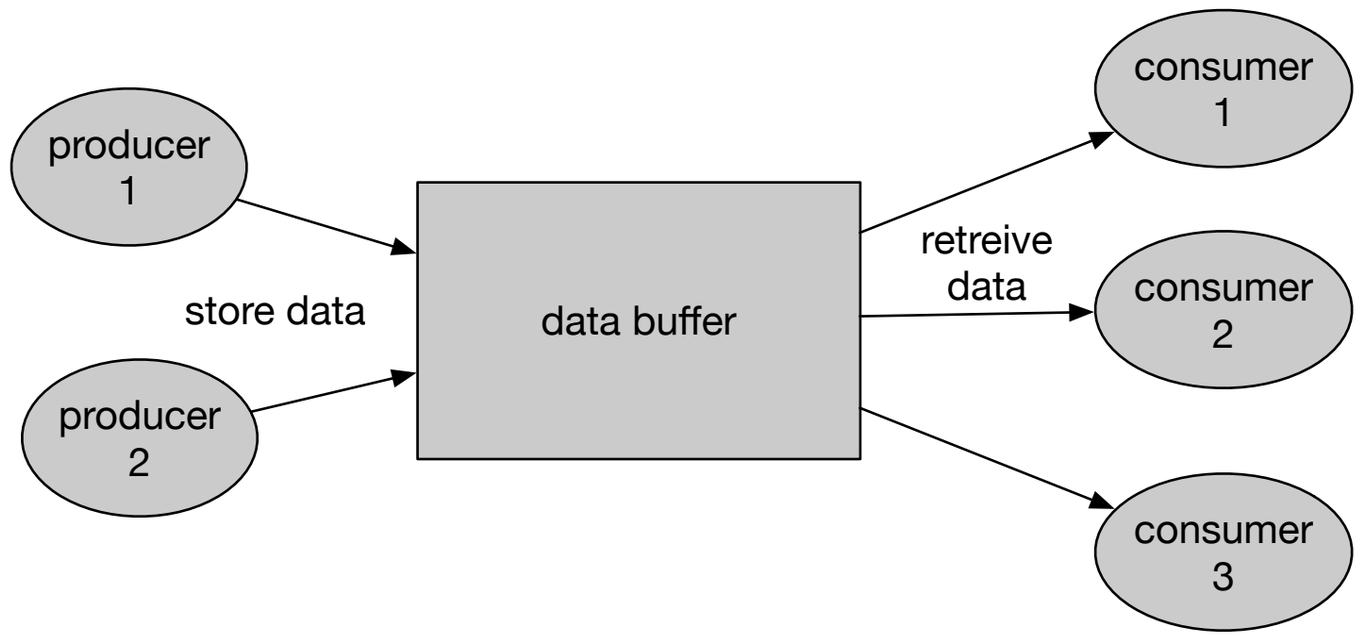
- **Producer** threads produce data which are used as input by **consumer** threads.

- Data is transferred from producers to consumers, using a common data structure (e.g. a data buffer of fixed length, accessible by both types of threads). Producers store the data elements generated into the buffer, consumers retrieve data elements from the buffer for further processing

- Synchronization has to be used to ensure a correct coordination between producer and consumer threads.

Task pool vs. producer-consumer





Work queue

- A FIFO, LIFO, priority order queue is typically used in the producer-consumer paradigm

- Pay attention to the granularity of the data inserted in the queue to avoid to pay an excessive overhead needed to access the data in the queue.

- Using multiple queues reduces contention, but then there is need to balance work among queues.

  - A solution is to allow processes to perform **work stealing** from other queues than that assigned to them.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Information exchange

Information exchange

- To control the coordination of the different parts of a parallel program, information must be exchanged between the executing processors.

- The implementation depends changes if we are dealing with **shared** or **distributed memory** systems

Shared memory

- Each thread can access shared data in the global memory. Such shared data can be stored in **shared variables** which can be accessed as normal variables.
  A thread may also have private data stored in private variables, which cannot be accessed by other threads.

- To coordinate access by different threads to the same shared variable we need a sequentialization mechanism.

Shared memory: race condition

- The term **race condition** describes the effect that the result of a parallel execution of a program part by multiple execution units depends on the order in which the statements of the program part are executed by the different units.

- This may lead to **non-deterministic behavior**, since, depending on the execution order, different results are possible, and the exact outcome cannot be predicted.

- We need **mutual exclusion** to allow the execution of **critical sections** of code accessing the shared variables, using **lock** mechanisms.

Distributed memory

- Exchange of data and information between the processors is performed by communication operations which are explicitly called by the participating processors.
- The actual data exchange is realized by the transfer of messages between the participating processors. The corresponding programming models are therefore called **message-passing** programming models.
- There are **point-to-point** and **global** communication operations.

Communication operations

- **Single transfer**: for a single transfer operation, a processor P_i (sender) sends a message to processor P_j (receiver) with `j!=i`. For each send operation, there must be a corresponding receive operation, and vice versa. Otherwise, deadlocks may occur

- **Single-broadcast**: for a single-broadcast operation, a specific processor P_i sends the same data block to all other processors.

- **Single-accumulation**: for a single-accumulation operation, each processor provides a block of data with the same type and size. By performing the operation, a given reduction operation is applied element by element to the data blocks provided by the processors, and the resulting accumulated data block of the same length is collected at a specific root processor P_i

Communication operations

- **Gather**: for a gather operation, each processor provides a data block, and the data blocks of all processors are collected at a specific root processor P_i . No reduction operation is applied.

- **Scatter**: for a scatter operation, a specific root processor P_i provides a separate data block for every other processor.

Communication operations

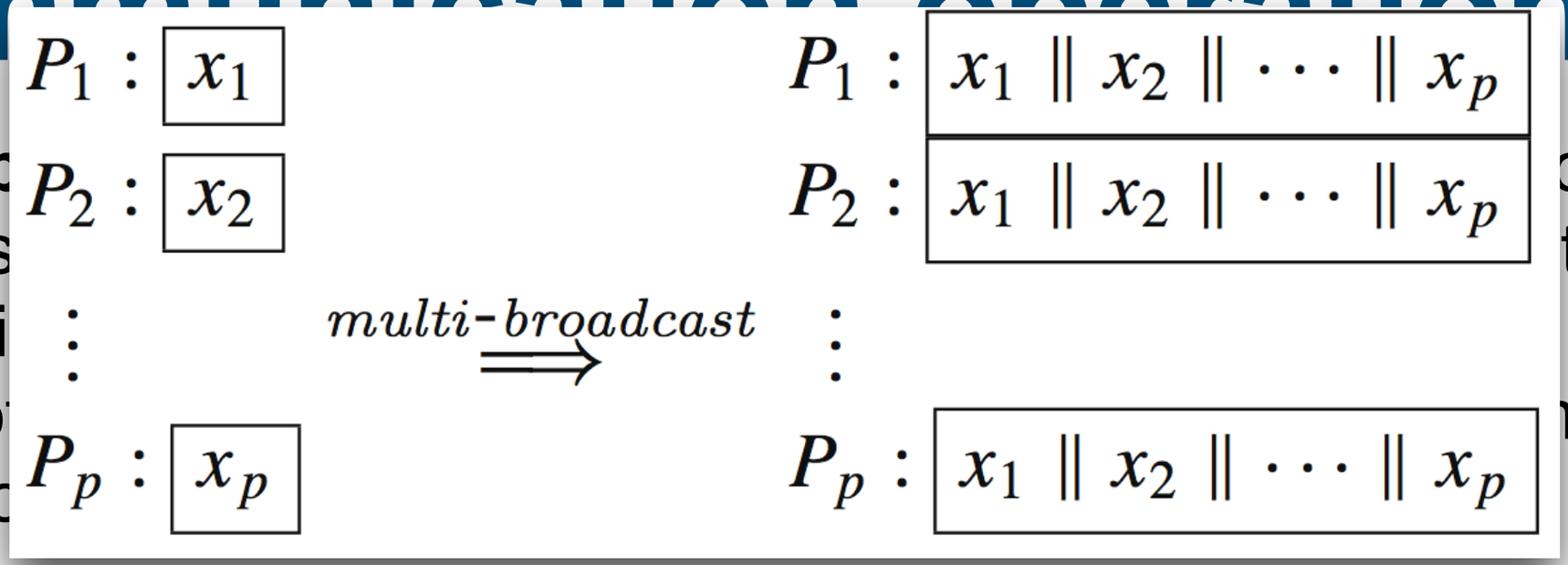
- **Multi-broadcast**: the effect of a multi-broadcast operation is the same as the execution of several single-broadcast operations, one for each processor. From the receiver's point of view, each processor receives a data block from every other processor; there is no root processor.

- **Multi-accumulation**: the effect of a multi-accumulation operation is that each processor executes a single-accumulation operation

- **Total exchange**: for a total exchange operation, each processor provides for each other processor a potentially different data block. These data blocks are sent to their intended receivers, i.e., each processor executes a **scatter** operation.

Communication operations

- **Multi-b** on is the s t operati point o n every c



- **Multi-accumulation**: the effect of a multi-accumulation operation is that each processor executes a single-accumulation operation

- **Total exchange**: for a total exchange operation, each processor provides for each other processor a potentially different data block. These data blocks are sent to their intended receivers, i.e., each processor executes a **scatter** operation.

Communication operations

- **Multi-b**
  is the s
  operati
  point o
  every c

$$\begin{array}{|l|l|}\hline P_1 : & x_1 \\ \hline P_2 : & x_2 \\ \hline \vdots & \\ \hline P_p : & x_p \\ \hline \end{array} \quad \overset{multi\text{-}broadcast}{\Longrightarrow} \quad \begin{array}{|l|l|}\hline P_1 : & x_1 \parallel x_2 \parallel \cdots \parallel x_p \\ \hline P_2 : & x_1 \parallel x_2 \parallel \cdots \parallel x_p \\ \hline \vdots & \\ \hline P_p : & x_1 \parallel x_2 \parallel \cdots \parallel x_p \\ \hline \end{array}$$

- **Multi-acc**
  operation
  accumula

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

- **Total exchange**: for a total exchange operation, each processor provides for each other processor a potentially different data block. These data blocks are sent to their intended receivers, i.e., each processor executes a **scatter** operation.

Communication operations

- **Multi-b**
  is the s
  operati
  point o
  every c

$$\begin{array}{|l} P_1 : \boxed{x_1} \\ P_2 : \boxed{x_2} \\ \vdots \\ P_p : \boxed{x_p} \end{array} \xrightarrow{multi\text{-}broadcast} \begin{array}{|l} P_1 : \boxed{x_1 \parallel x_2 \parallel \cdots \parallel x_p} \\ P_2 : \boxed{x_1 \parallel x_2 \parallel \cdots \parallel x_p} \\ \vdots \\ P_p : \boxed{x_1 \parallel x_2 \parallel \cdots \parallel x_p} \end{array}$$

- **Multi-acc**
  operation
  accumula

$$\begin{array}{|l} P_1 : \boxed{x_{11} \parallel x_{12} \parallel \cdots \parallel x_{1p}} \\ P_2 : \boxed{x_{21} \parallel x_{22} \parallel \cdots \parallel x_{2p}} \\ \vdots \\ P_p : \boxed{x_{p1} \parallel x_{p2} \parallel \cdots \parallel x_{pp}} \end{array} \xrightarrow{multi\text{-}accumulation} \begin{array}{|l} P_1 : \boxed{x_{11} + x_{21} + \cdots + x_{p1}} \\ P_2 : \boxed{x_{12} + x_{22} + \cdots + x_{p2}} \\ \vdots \\ P_p : \boxed{x_{1p} + x_{2p} + \cdots + x_{pp}} \end{array}$$

- **Total**
  proces
  differe
  intend
  opera

$$\begin{array}{|l} P_1 : \boxed{x_{11} \parallel x_{12} \parallel \cdots \parallel x_{1p}} \\ P_2 : \boxed{x_{21} \parallel x_{22} \parallel \cdots \parallel x_{2p}} \\ \vdots \\ P_p : \boxed{x_{p1} \parallel x_{p2} \parallel \cdots \parallel x_{pp}} \end{array} \xrightarrow{total\ exchange} \begin{array}{|l} P_1 : \boxed{x_{11} \parallel x_{21} \parallel \cdots \parallel x_{p1}} \\ P_2 : \boxed{x_{12} \parallel x_{22} \parallel \cdots \parallel x_{p2}} \\ \vdots \\ P_p : \boxed{x_{1p} \parallel x_{2p} \parallel \cdots \parallel x_{pp}} \end{array}$$

Credits

- These slides report material from:
  - Prof. Robert van Engelen (Florida State University)
  - Prof. Jan Lemeire (Vrjie Universiteit Brussel)

Books

- The Art of Concurrency, Clay Breshears, O'Reilly - Chapt. 2, 4

- Parallel Programming for Multicore and Cluster Systems, Thomas Rauber and Gudula Rünger, Springer - Chapt. 3

- The Art of Multiprocessor Programming, Maurice Herlihy and Nir Shavit, Morgan Kaufmann - Chapt. 1