UNIVERSITÀ DEGLI STUDI FIRENZE

Parallel Programming

Prof. Marco Bertini

Design models for parallel programs

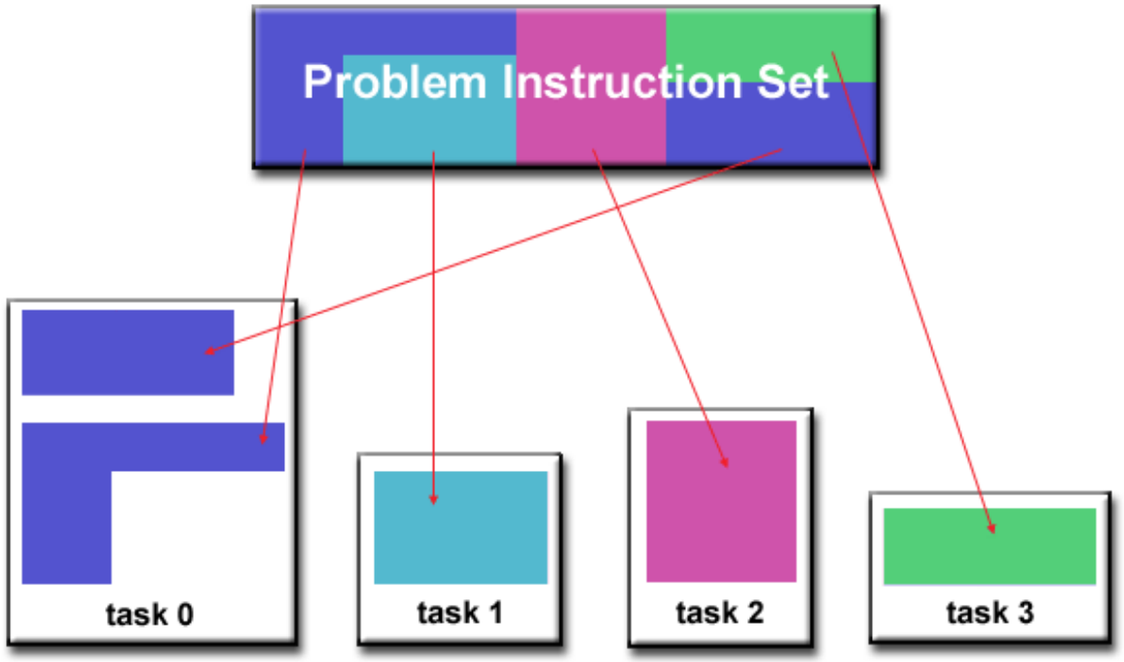
Reorganizing computations

- **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

- In this approach (also called functional decomposition), the focus is on the computation that is to be performed rather than on the data manipulated by the computation.

- The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.



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.

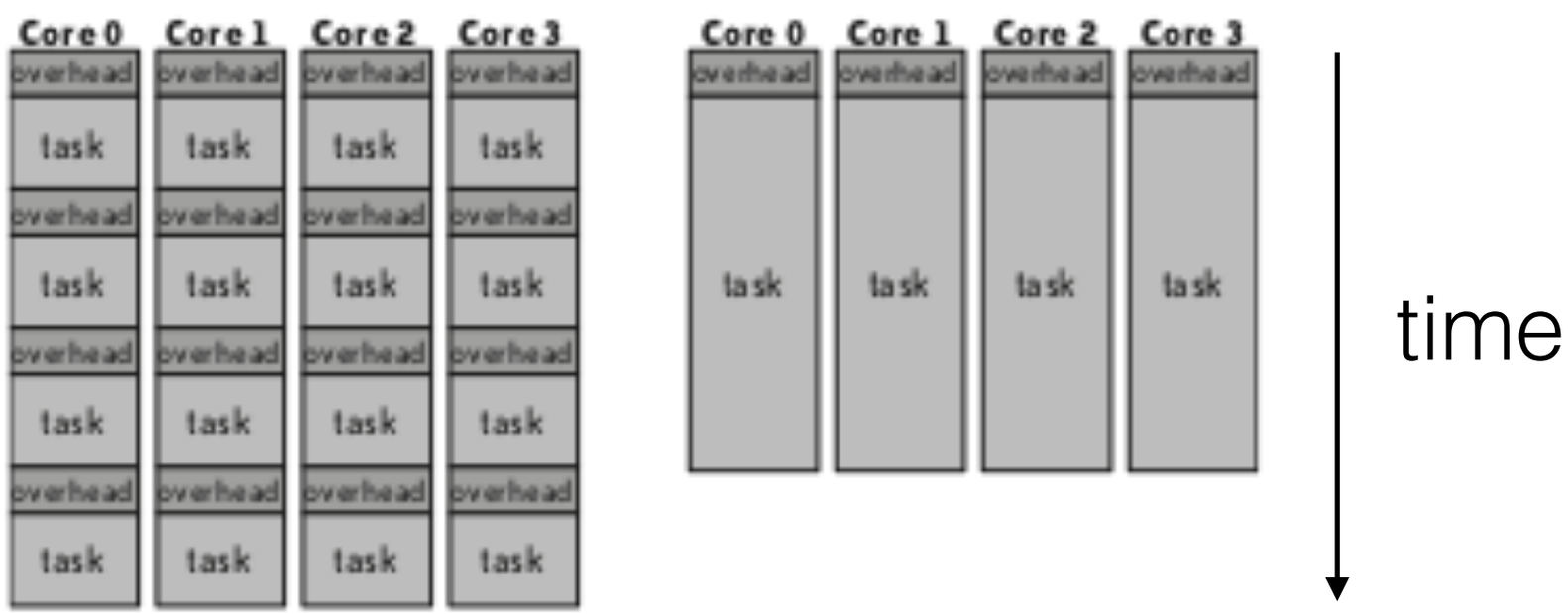
Decomposition criteria

- Programs often naturally decompose into tasks. Two common decompositions are
  - Function calls
  - Distinct loop iterations
- Easier to start with many tasks and later fuse them, rather than too few tasks and later try to split them

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

Goals

- Flexibility
  Program design should afford flexibility in the number and size of tasks generated
  - Tasks should not be tied to a specific architecture
  - Fixed tasks vs. Parameterized tasks
- Efficiency
  - Tasks should have enough work to amortize the cost of creating and managing them
  - Tasks should be sufficiently independent so that managing dependencies doesn't become the bottleneck
- Simplicity
  The code has to remain readable and easy to understand, and debug

Data decomposition

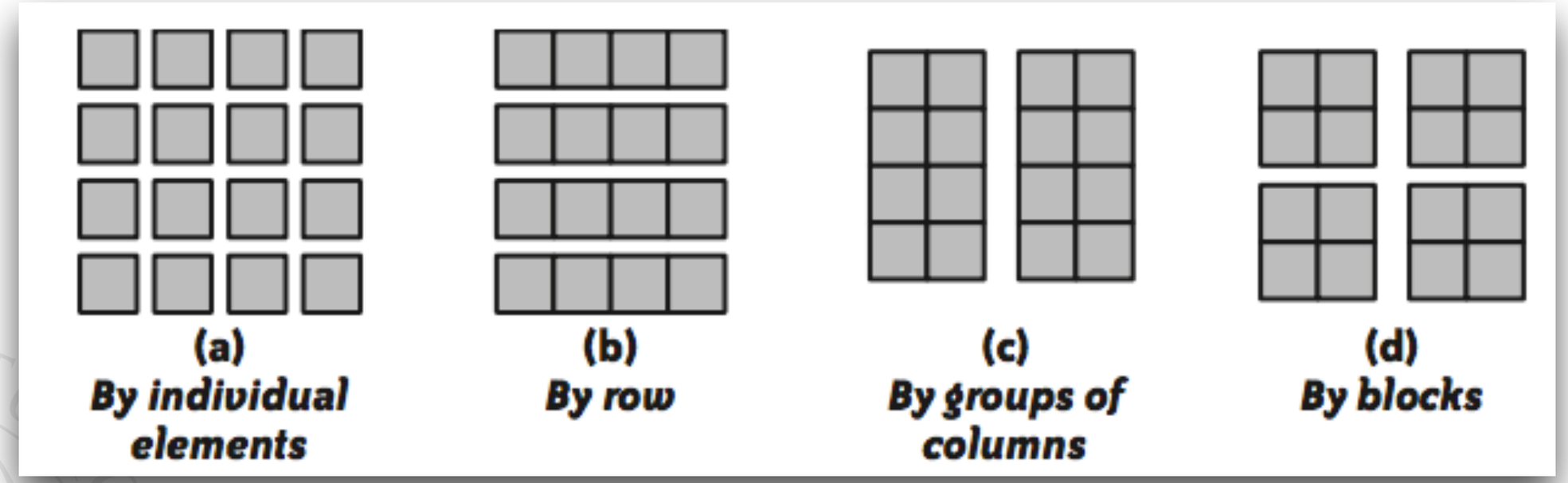
- We may identify that execution of a serial program is dominated by a sequence of operations on all elements of one or more large data structures.
  If these are independent we can divide the data assigning portions (*chunks*) to different tasks.

- 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 of 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 of 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 of 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.

Data decomposition

- Data decomposition is often implied by task decomposition
- Data decomposition is a good starting point when
  - Main computation is organized around manipulation of a large data structure
  - Similar operations are applied to different parts of the data structure

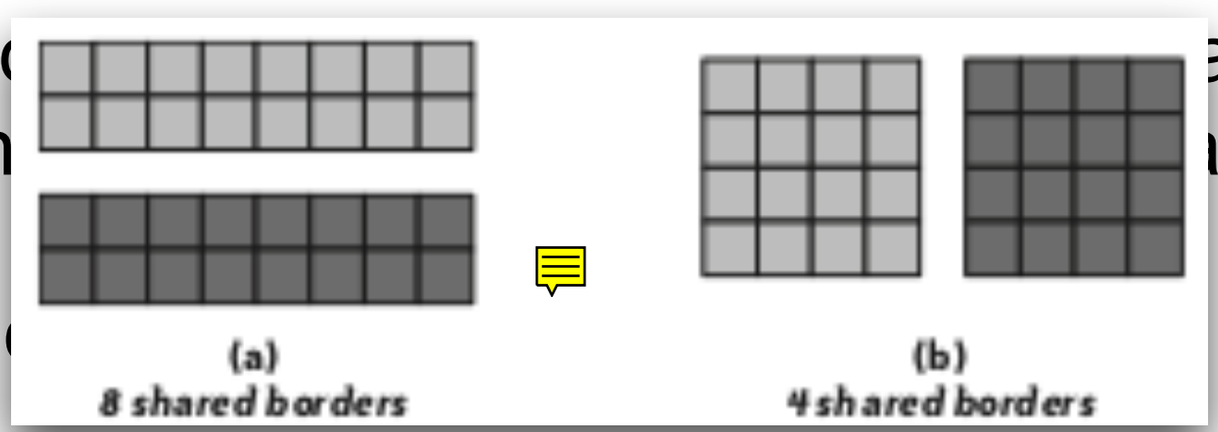
Chunk shape

- 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 d ata elements; reducing th re a border with a given operation less compli

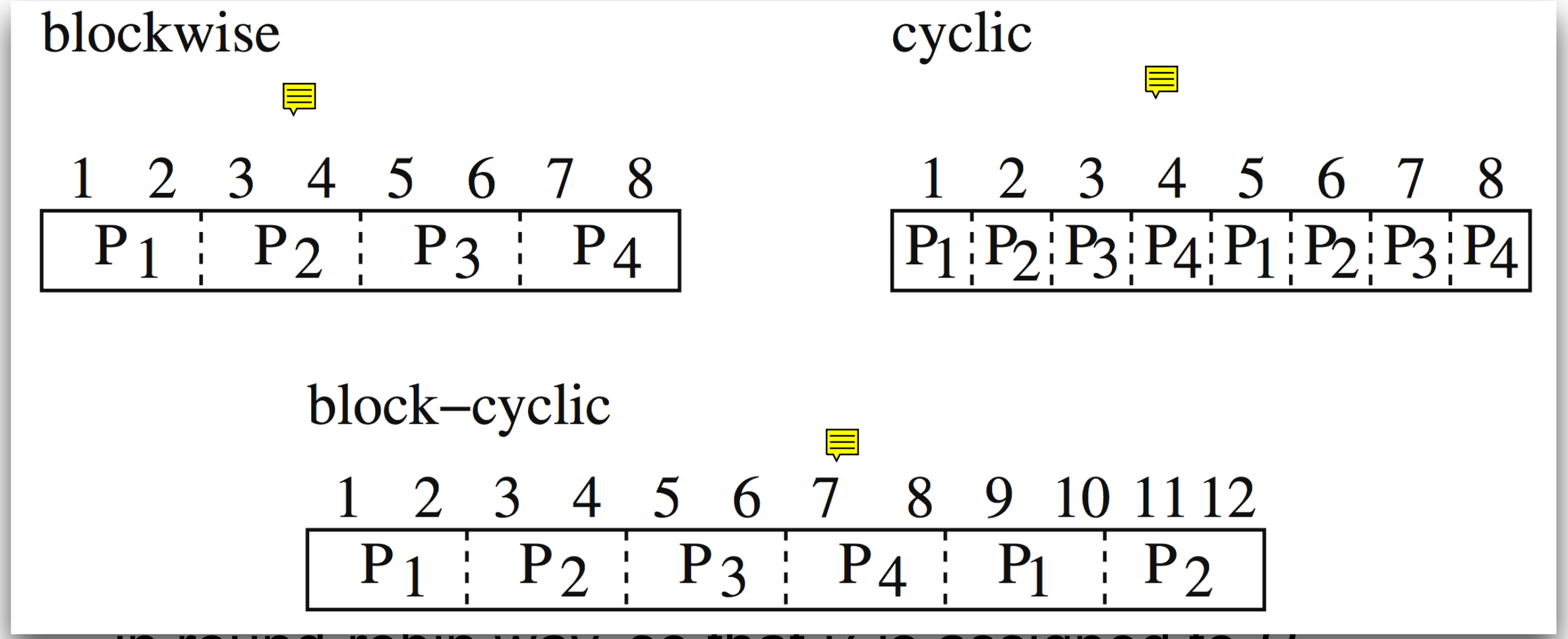


- A good rule of thumb is to try to maximize the volume-to-surface ratio. The volume defines the granularity of the computations, and the surface is the border of chunks that require an exchange of data.

Decomposition example: Data Distributions for Arrays

- Let us consider a set of processes $P = \{P_1, \ldots, P_p\}$
- 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



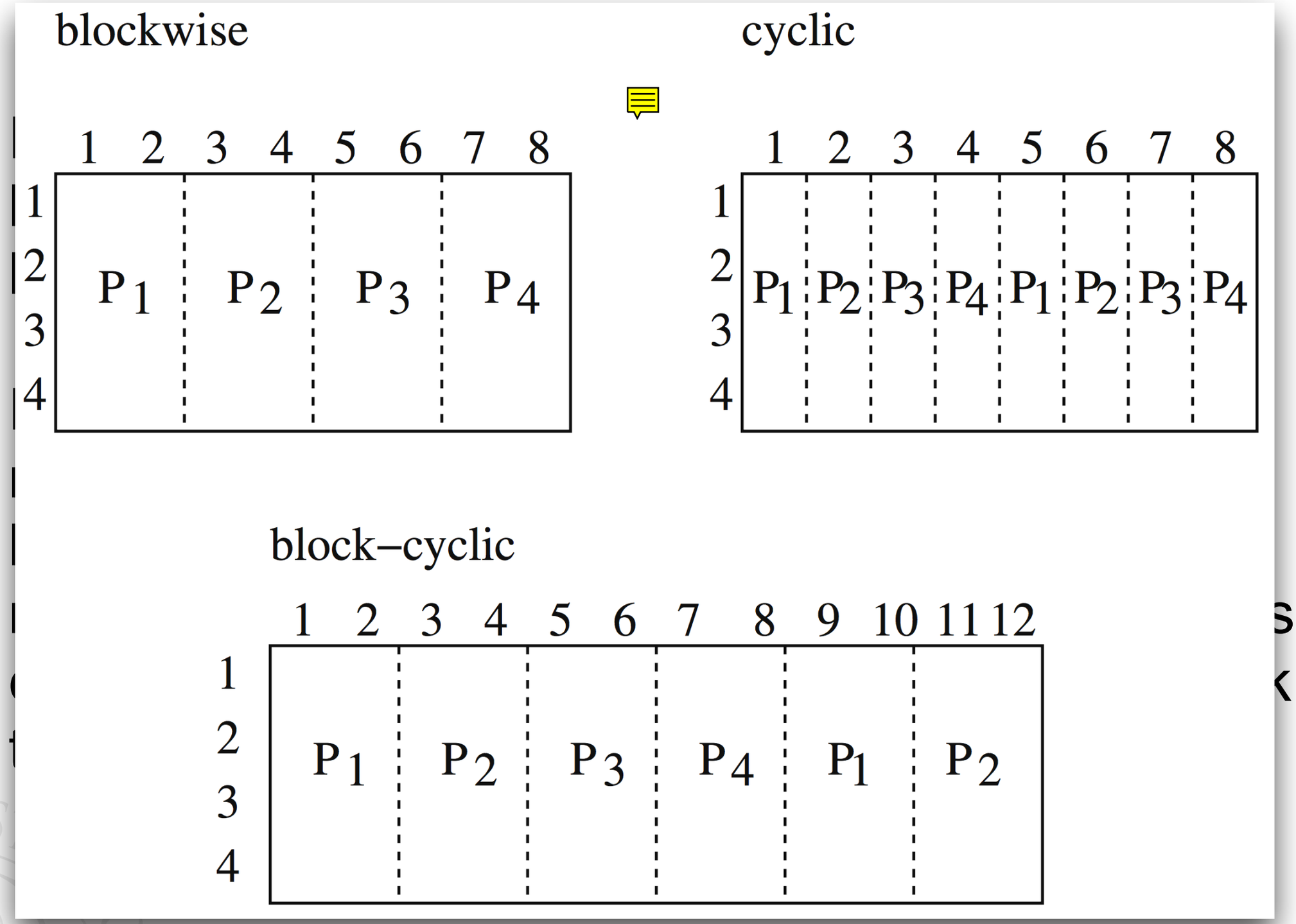
in round-robin way, so that v_i is assigned to $P_{(i-1) \bmod p+1}$

- **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.

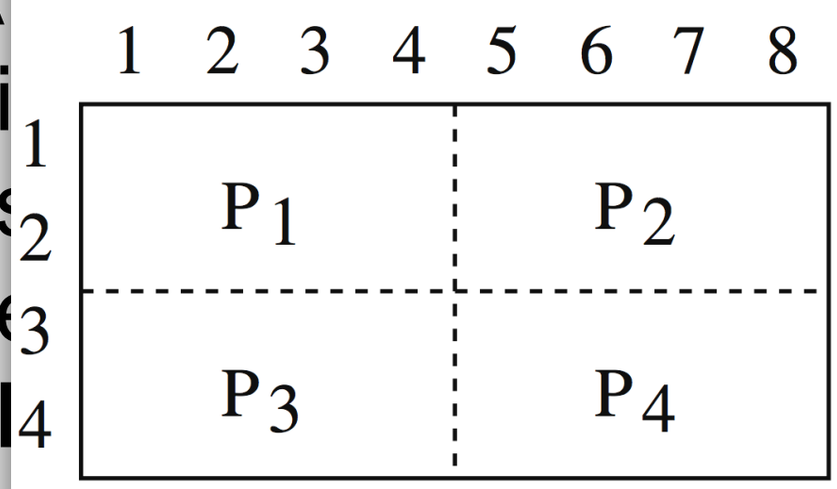
- The processors are arranged in a virtual mesh of size $p_1 \cdot p_2 = p$ where p_1 is the number of rows and p_2 is the number of columns in the mesh.
  Array elements (k,l) are mapped to processors $P_{i,j}$, $i = 1, \ldots, p_1$, $j = 1, \ldots, p_2$.

Decomposition example: Data Distributions for Arrays

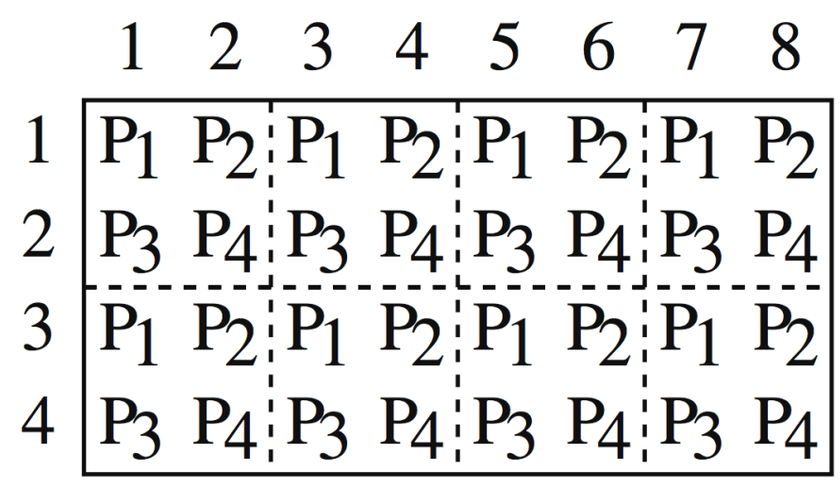
- A
di ons
us sh
be
cl

- Th of
si nd
p
A
P

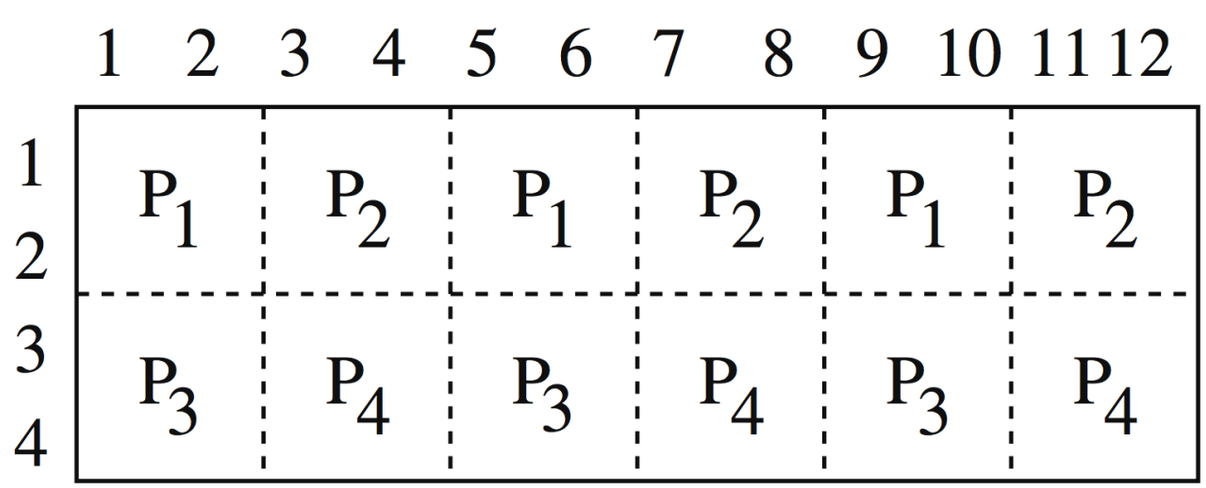
blockwise



cyclic



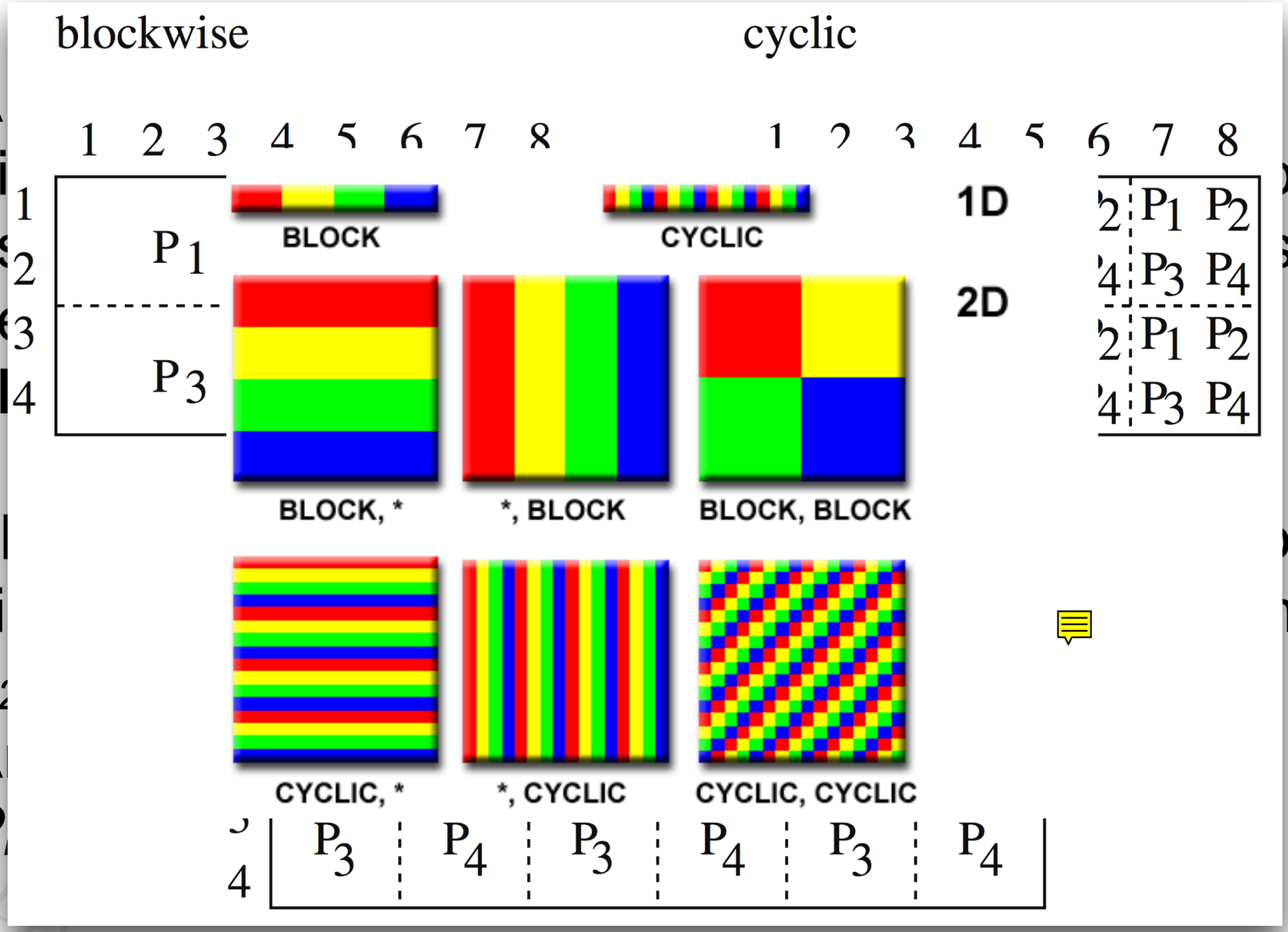
block–cyclic



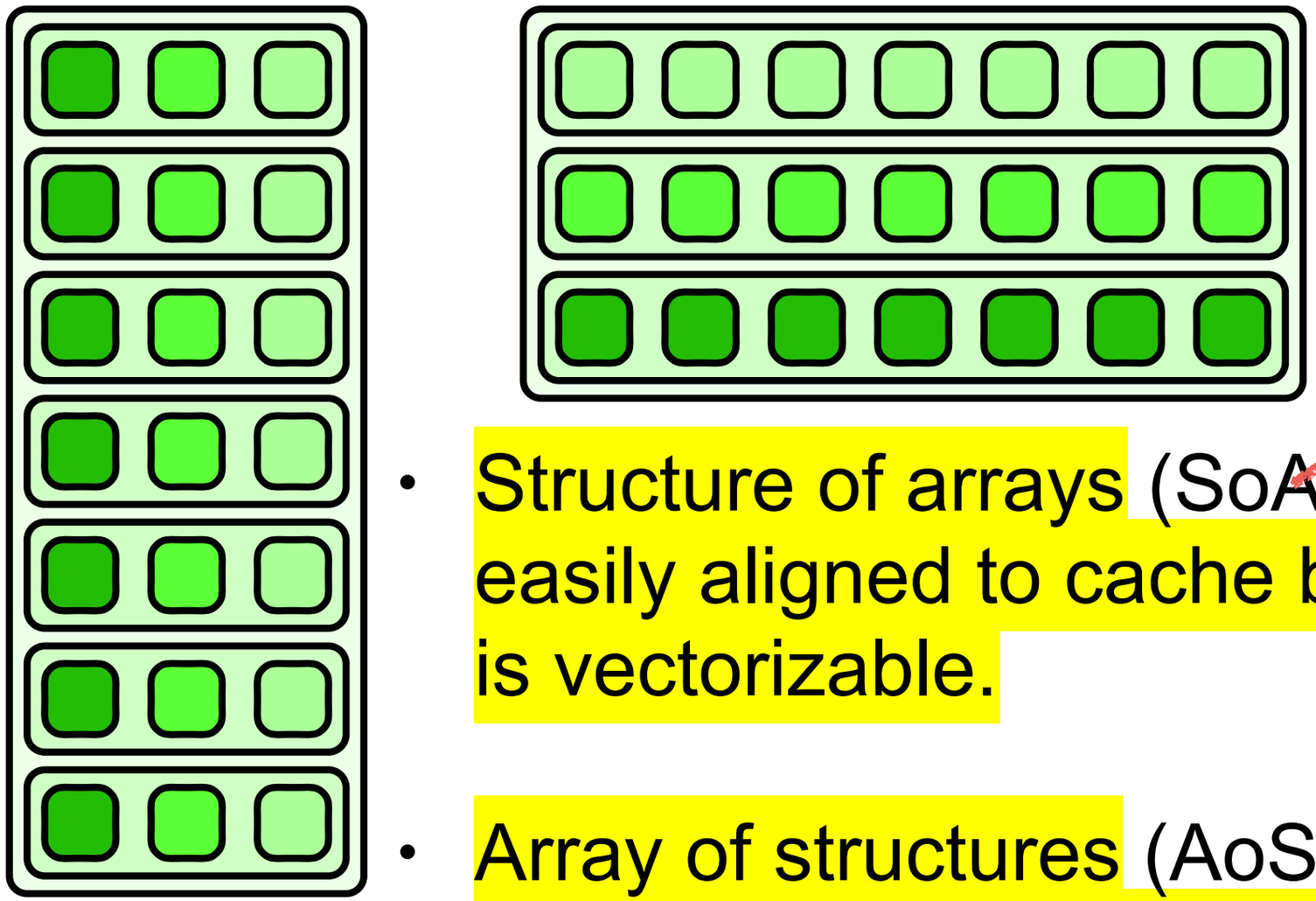
Decomposition example: Data Distributions for Arrays

- A
di ons
us sh
be
cl

- Th of
si nd
pa
A
P

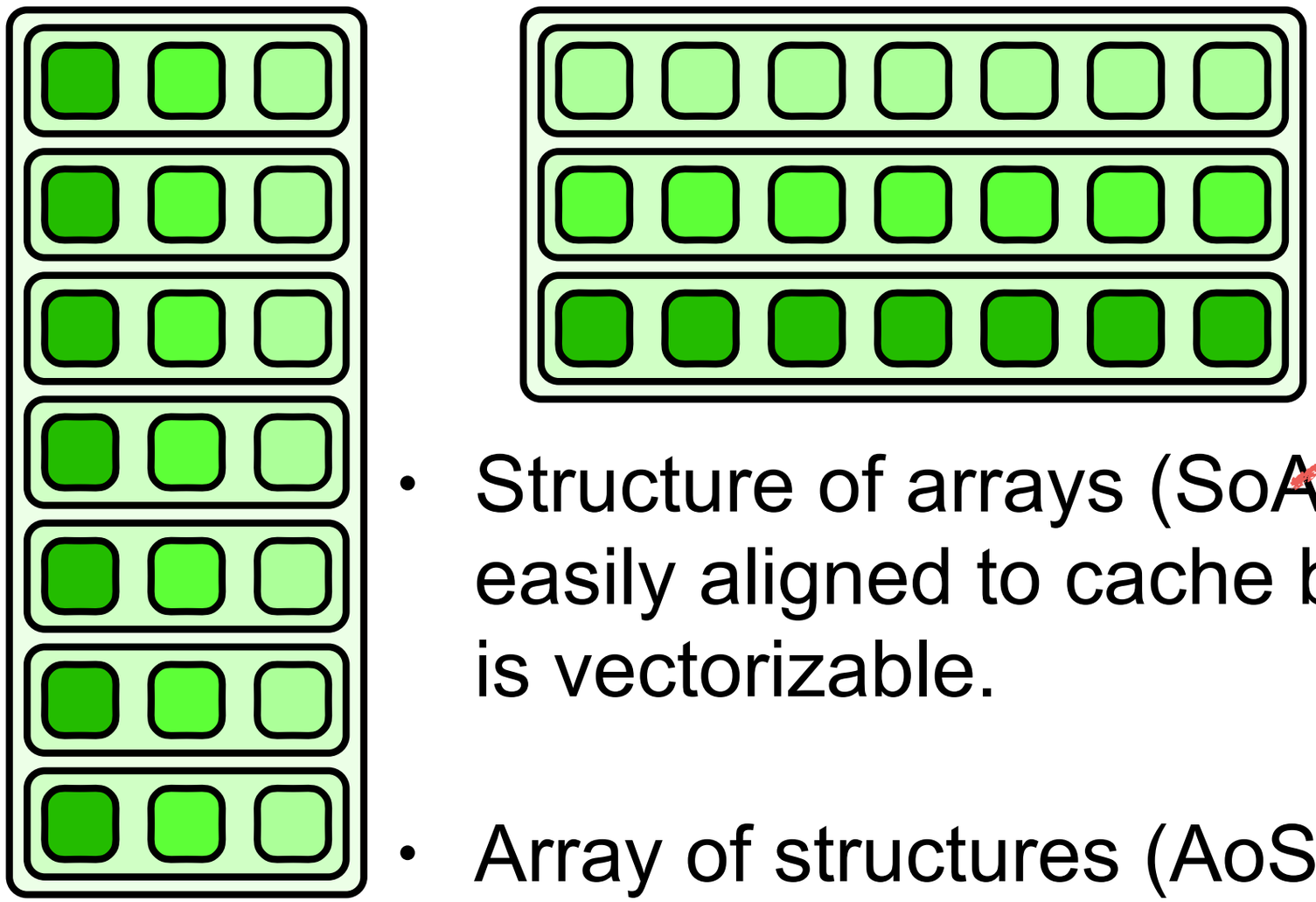


Data layout: AoS vs. SoA



- Structure of arrays (SoA) can be easily aligned to cache boundaries and is vectorizable.
- Array of structures (AoS) may cause cache alignment problems (but fields are near, so good for cache), and is harder to vectorize.

Data layout: AoS vs. SoA



- Structure of arrays (SoA) can be easily aligned to cache boundaries and is vectorizable.
- Array of structures (AoS) may cause cache alignment problems (but fields are near, so good for cache), and is harder to vectorize.

… but it's easier to branch or call a function on just an element.

Data layout: alignment

Array of Structures (AoS), padding at end.

Array of Structures (AoS), padding after each structure.

Structure of Arrays (SoA), padding at end.

Structure of Arrays (SoA), padding after each component.

Goals

- Flexibility
  - Size and number of data chunks should support a wide range of executions
- Efficiency
  - Data chunks should generate comparable amounts of work (for load balancing)
- Simplicity
  - Complex data compositions can get difficult to manage and debug

Data-oriented design

- In this programming style the focus is on the best data layout for the CPU and the cache. Some principles are:
  - Operate on arrays, not individual data items, avoiding the call overhead and the instruction and data cache misses
  - Prefer arrays rather than structures for better cache usage in more situations
    - E.g. use contiguous array-based linked lists to avoid the standard linked list implementations used in C and C++, which jump all over memory with poor data locality and cache usage
  - Inline subroutines rather than transversing a deep call hierarchy
    - Use short methods/function that can be inlined, use more a C-style design than C++
  - Control memory allocation, avoiding undirected reallocation behind the scenes

Communications

- The need for communications between tasks depends upon your problem:
- You DON'T need communications:
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. These types of problems are often called **embarrassingly parallel** - little or no communications are required.
- You DO need communications:
  - Most parallel applications are not quite so simple, and do require tasks to share data with each other.

Communications

- There are a number of important factors to consider when designing your program's inter-task communications:

- Communication overhead

  - Inter-task communication virtually always implies overhead.

  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.

  - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.

Communications

- Latency vs. Bandwidth
  - *latency* is the time it takes to send a minimal (0 byte) message from point A to point B.
  - *bandwidth* is the amount of data that can be communicated per unit of time.
  - Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

Communications

- Synchronous vs. asynchronous communications
  - **Synchronous** communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
  - Synchronous communications are often referred to as blocking communications since other work must wait until the communications have completed.
  - **Asynchronous** communications allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
  - Asynchronous communications are often referred to as non-blocking communications since other work can be done while the communications are taking place.
  - Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

Examples of data structures

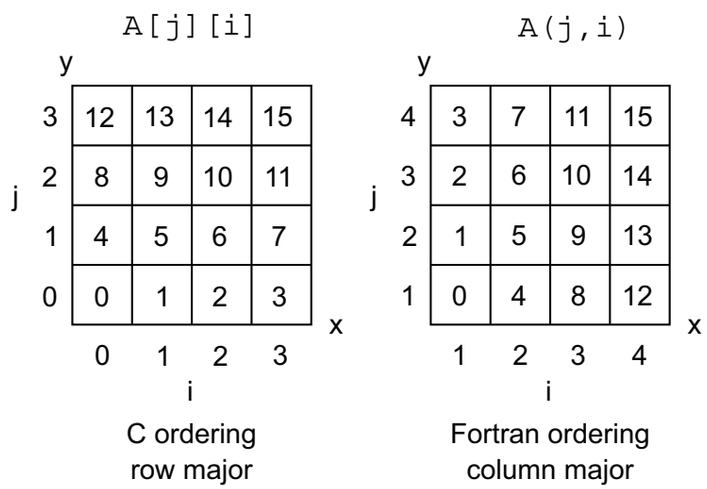
Why should we care ?

- Cache efficiency dominates the performance of intensive computations. As long as the data is cached, the computation proceeds quickly.
- When the data is not cached, a cache miss occurs. The cost of a cache miss is on the order of 100 to 400 cycles, needed to load data from memory
  - 100s of flops are lost during this miss
- We need to use data structures that are cache friendly: spatial and temporal locality are fundamental
- With regular, predictable memory accesses of large arrays, it is possible to prefetch data (i.e. preload data in cache before it is needed).

Cache misses

- There are 3 types of cache misses:
  - *Compulsory* — Cache misses that are necessary to bring in the data when it is first encountered.
  - *Capacity* — caused by a limited cache size, which evicts data from the cache to free up space for new cache line loads.
  - *Conflict* — When data is loaded at the same location in the cache. If two or more data items are needed at the same time but are mapped to the same cache line, both data items must be loaded repeatedly for each data element access.
- Cache misses due to *capacity* or *conflict* evictions followed by reloads of the cache lines, are referred to as *cache thrashing*, which can lead to poor performance.

Multidimensional arrays



- Matrices in C are stored in data order referred to as row major, while in Fortran the data layout is column major (and indexes start at 1 instead of 0)
- As programmers, we must remember which index should be in the inner loop to leverage the contiguous memory in each situation
  - In C the inner loop should use the column index to leverage the contiguous memory

```
for (int j=0; j<jmax; j++){
for (int i=0; i<imax; i++){
    A[j][i] = 0.0;
  }
}
```

In C, the last index is fastest and should be the inner loop.

Contiguous allocation

- In Fortran it is possible to specify a contiguous allocation for the multidimensional arrays

- In C we must be careful on how we allocate it:

```
double **x =
   (double **)malloc(jmax*sizeof(double *));

for (j=0; j<jmax; j++){
   x[j] =
      (double *)malloc(imax*sizeof(double));
}

// computation

for (j=0; j<jmax; j++){
   free(x[j]);
}
free(x);
```

Allocates a column of pointers of type pointer to double

Allocates each row of data

Deallocates memory

- Each allocation can come from a different place in the heap.

- No spatial locality

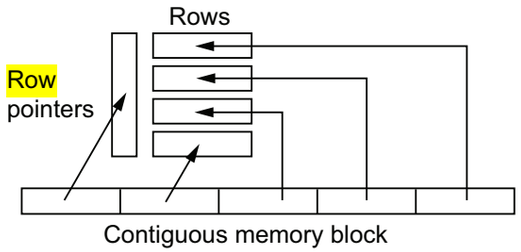
```
double **x = (double **)malloc(jmax*sizeof(double *) +
             jmax*imax*sizeof(double));

x[0] = (double *)x + jmax;

for (int j = 1; j < jmax; j++) {
   x[j] = x[j-1] + imax;
}
```

Allocates a block of memory for the row pointers and the 2D array

Assigns the start of the memory block for the 2D array after the row pointers

Assigns the memory location to point to the data block for each row pointer

- Contiguous allocation, pointers to rows



- Can be accessed using `x[j][i]` or using `*x1d = x[0]`, `x1d[i + imax*j]`

Contiguous allocation

- In Fortran it is possible to specify a contiguous allocation for the multidimensional arrays
- In C we must be careful on how

Or simply linearize 2D coordinates to 1D and skip the row pointer allocation

```
double **x =
   (double **)malloc(jmax*sizeof(double *));

for (j=0; j<jmax; j++){
   x[j] =
      (double *)malloc(imax*sizeof(double));
}

// computation

for (j=0; j<jmax; j++){
   free(x[j]);
}
free(x);
```

Allocates a column of pointers of type pointer to double

Allocates each row of data

Deallocates memory

- Each allocation can come from a different place in the heap.
- No spatial locality

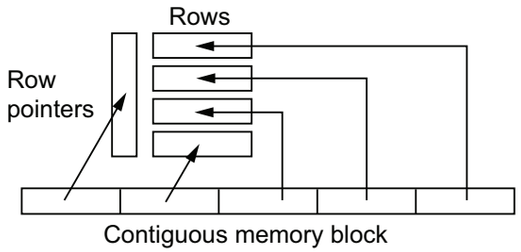
```
double **x = (double **)malloc(jmax*sizeof(double *) +
             jmax*imax*sizeof(double));

x[0] = (double *)x + jmax;

for (int j = 1; j < jmax; j++) {
   x[j] = x[j-1] + imax;
}
```

Allocates a block of memory for the row pointers and the 2D array

Assigns the start of the memory block for the 2D array after the row pointers

Assigns the memory location to point to the data block for each row pointer

- Contiguous allocation, pointers to rows
- Can be accessed using `x[j][i]` or using `*x1d = x[0]`, `x1d[i + imax*j]`



Array of Structures (AoS)

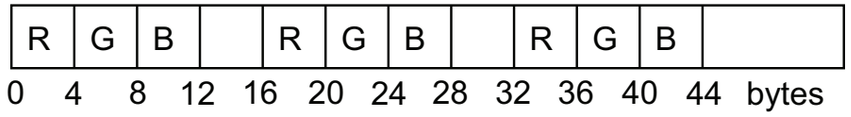
- Let us consider a graphics application that operates on RGB values. It has to read the three color channels at once.

```
struct RGB {
   int R;
   int G;
   int B;
};
struct RGB polygon_color[1000];
```

Defines a scalar color value

Defines an Array of Structures (AoS)

- The AoS works well on a CPU, but if we had to read only one of the values we'd reduce cache usage and code wouldn't be vectorized
  - We may suffer from padding added by the compiler for memory alignment of the structure



Structure of Arrays (SoA)

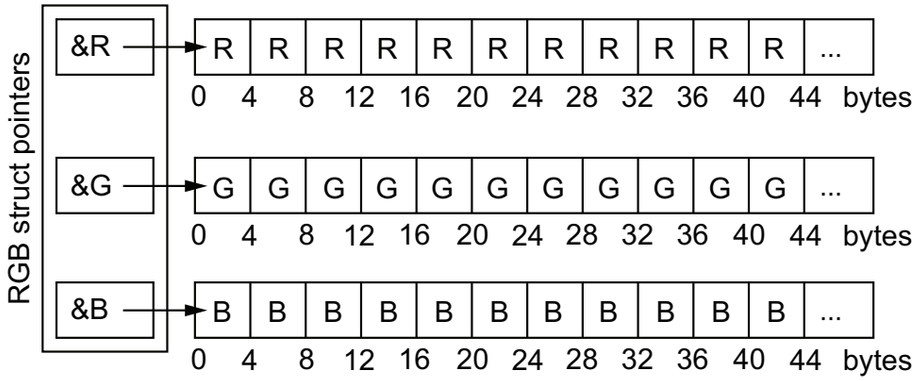
- Considering RGB values now Rs are contiguous, like G and B but all these may be in different parts of the heap.

```
struct RGB {
   int *R;
   int *G;
   int *B;
};
struct RGB polygon_color;

polygon_color.R = (int *)malloc(1000*sizeof(int));
polygon_color.G = (int *)malloc(1000*sizeof(int));
polygon_color.B = (int *)malloc(1000*sizeof(int));
```

Defines an integer array of a color value

Defines a Structure of Arrays (SoA)

- If we need to operate on all the three channels at the same time and rows are long then it may be a problem for CPU caches (but not for GPUs)



SoA vs. AoS: other cases

- Let's consider a structure that represents 3D points.
  - If we use all the coordinates for a computation (e.g. compute a distance) AoS makes sense.
  - If we need only some coordinates (e.g. compute a gradient along a specific axis) then SoA is better.
  - In AoS, fields are adjacent (ie. cache-friendly) when always touching all of them, but bad for SIMD if only touching one field.
- The optimal data layout depends entirely on usage and the particular data access patterns.
- In mixed use cases, which are likely to appear in real applications, sometimes the structure variables are used together and sometimes not.
- Generally, the AoS layout performs better overall on CPUs, while the SoA layout performs better on GPUs. If there is enough variability test for a particular usage pattern.

SoA vs. AoS: other cases

- Let's consider a structure that represents a simple/naïve hash table.
  - If we use a AoS we pass through the keys until we find the required one and then select the value. Values wast space in the cache lines.
  - If we use a SoA we have a faster search.

```
struct hash_type {
   int key;
   int value;
};
struct hash_type hash[1000];
```

```
struct hash_type {
   int *key;
   int *value;
} hash;
hash.key   = (int *)malloc(1000*sizeof(int));
hash.value = (int *)malloc(1000*sizeof(int));
```

Array of Structures of Arrays (AoSoA)

- Array of Structures (AoS):
  Each element is a full particle (all fields together).
  Memory: [x y z id][x y z id][x y z id]...

- Structure of Arrays (SoA):
  Each attribute has its own contiguous array.
  Memory: [xxx...][yyy...][zzz...][ididid...]

- Array of Structures of Arrays (AoSoA):
  Data is grouped into blocks (tiles), and inside each block, data is stored as SoA.
  Memory: [(xxx...)(yyy...)(zzz...)] [(xxx...)(yyy...)(zzz...)] ...

- SoA inside each tile, AoS across tiles.

- Tries to combine the cache efficiency of AoS (near fields) with the vectorization advantages of SoA

Array of Structures of Arrays (AoSoA)

- The Array of Structures of Arrays (AoSoA) can be used to “tile” the data into vector lengths. Let’s consider the RGB example.

- Let’s introduce the notation `A[len/4]S[3]A[4]` to represent this layout. `A[4]` is an array of four data elements and is the inner, contiguous block of data. `S[3]` represents the next level of the data structure of three fields (RGB).

- `S[3]A[4]` =

R	R	R	R	G	G	G	G	B	B	B	B	...

0 4 8 12 16 20 24 28 32 36 40 44 bytes

- Repeat the block of 12 data values A[len/4] times to get all the data.

- More in general: A[len/V]S[3]A[V] where V is the length of data elements (4 in the example)

Example: RGB Array of Structures of Arrays (AoSoA)

```
const int V=4;
struct SoA_type{
   int R[V], G[V], B[V];
};

int main(int argc, char *argv[])
{
   int len=1000;
   struct SoA_type AoSoA[len/V];

   for (int j=0; j<len/V; j++){
      for (int i=0; i<V; i++){

         AoSoA[j].R[i] = 0;
         AoSoA[j].G[i] = 0;
         AoSoA[j].B[i] = 0;
      }
   }
}
```

Sets vector length

Divides the array length by vector length

Loops over array length

Loops over vector length, which should vectorize

- If V=1 or V=len, we recover the AoS and SoA data structures, respectively.
- Varying V to match the hardware vector length or the GPU work group size, we create a portable data abstraction
- This data layout then becomes a way to adapt for the hardware and the program's data use patterns.

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. (synchronous, like speaking)
  - **Persistent communication** allows the sender and receiver to participate at different times. (asynchronous, 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. (synchronous, like speaking)
  - **Persistent communication** allows the sender and receiver to participate at different times. (asynchronous, 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.

Parallel algorithms

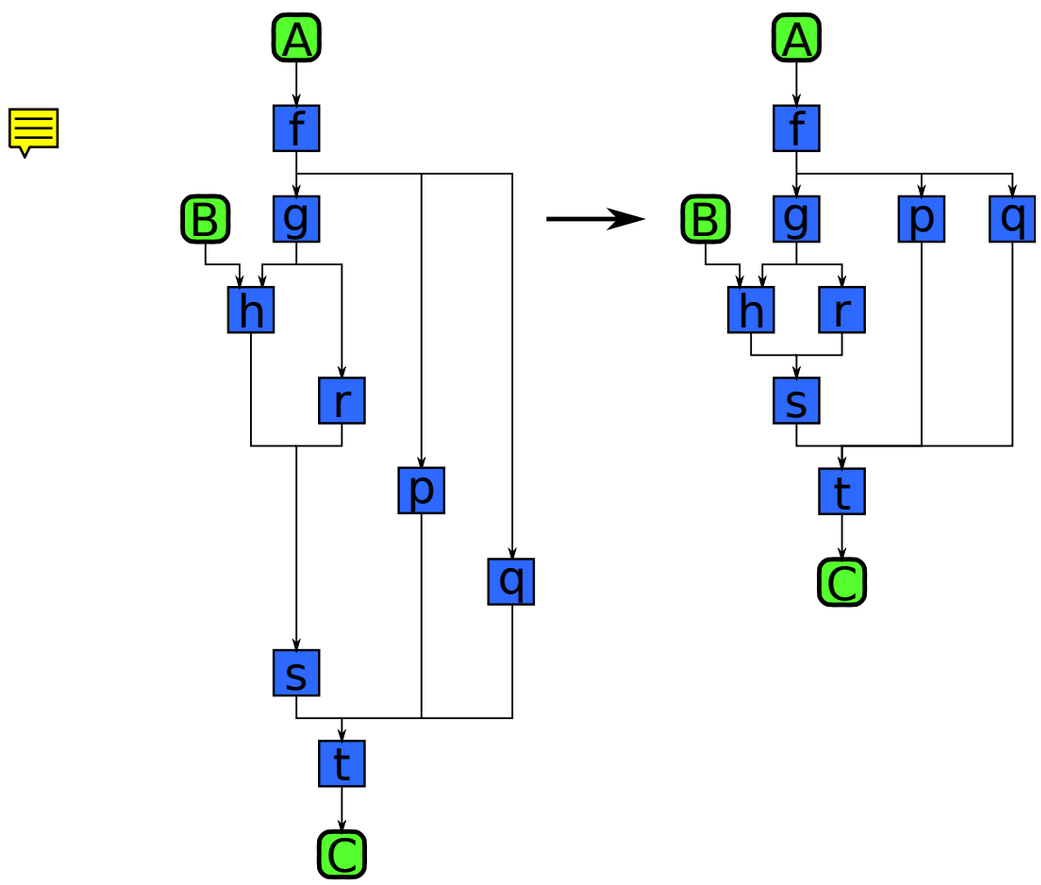
- Important properties for parallel algorithms are:
  - Locality — Often-used term in describing good algorithms but without any definition. It can have multiple meanings. Here are a couple:
    - Locality for cache — Keeps the values that will be used together close together so that cache utilization is improved.
    - Locality for operations — Avoids operating on all the data when not all is needed.
  - Asynchronous — Avoids coordination between threads that can cause synchronization.
  - Fewer conditionals — Besides the additional performance hit from conditional logic, thread divergence can be a problem on some architectures (i.e. GPUs).
  - Reproducibility — Often a highly parallel technique violates the lack of associativity of finite-precision arithmetic. Enhanced-precision techniques can help counter this issue.
  - Higher arithmetic intensity — Current architectures have added floating-point capability faster than memory bandwidth. Algorithms that increase arithmetic intensity can make good use of parallelism such as the vector operations.

UNIVERSITÀ DEGLI STUDI FIRENZE

Design patterns for parallel programming

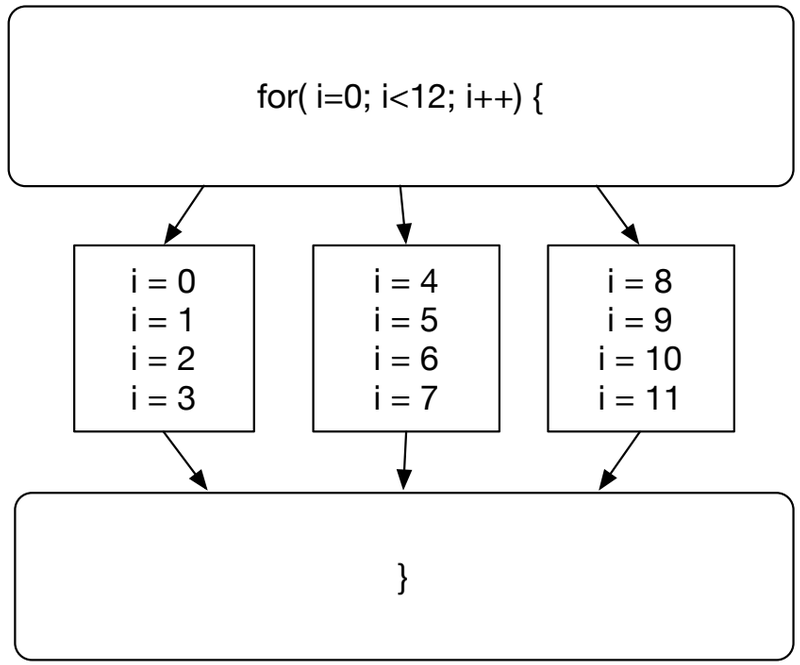
Superscalar sequence

- Tasks ordered only by data dependencies
- Tasks can run whenever input data is ready



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

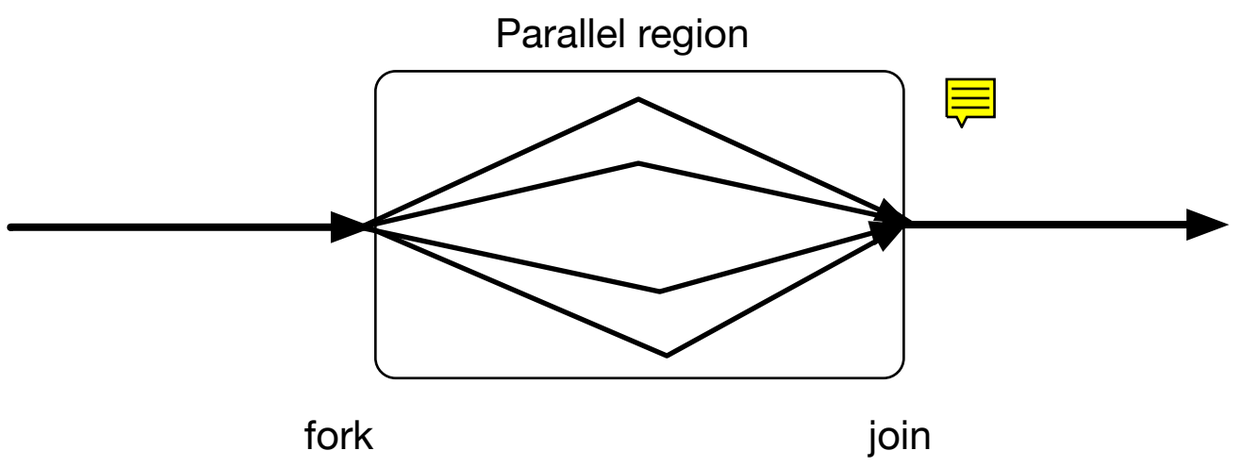


Task parallelism

- 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

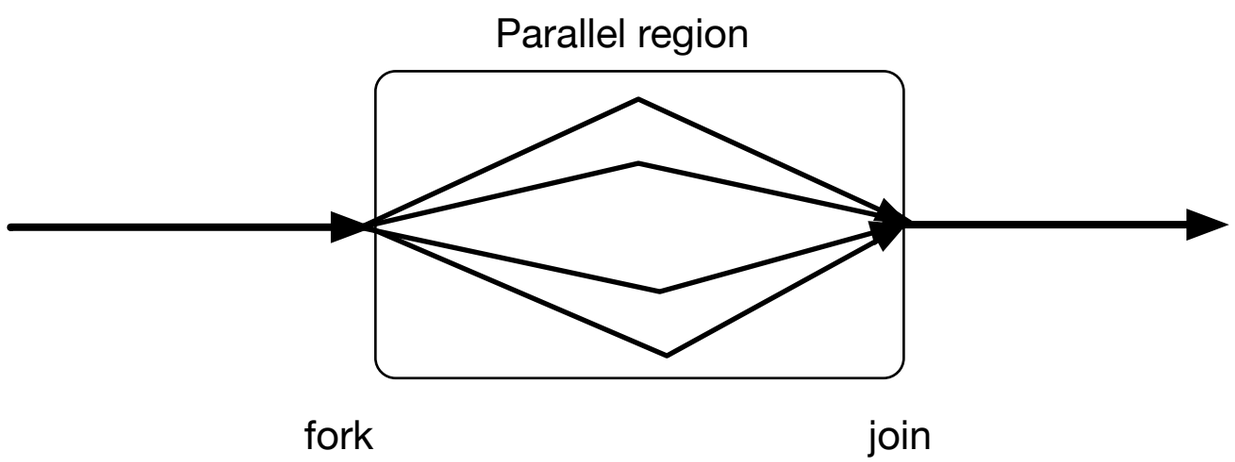
- A main process/thread forks some number of processes/threads that then continue in parallel to accomplish some portion of the overall work.
- Parent tasks creates new task (fork) then waits until all they complete (join) before continuing on with the computation



Fork-Join parallelism

Fork-Join comes from basic forms of creating processes and threads in operating system

- 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



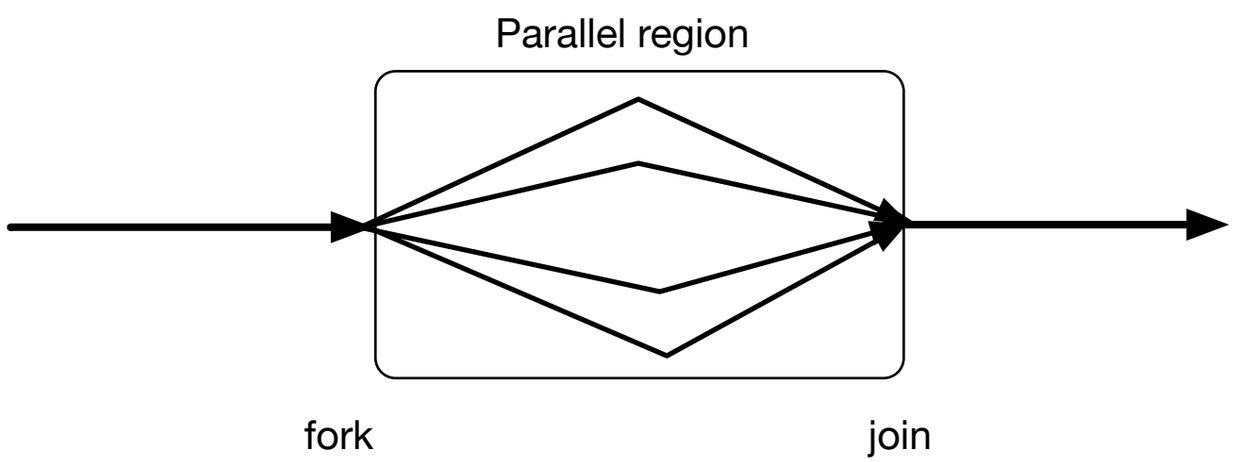
Fork-Join parallelism

Fork-Join comes from basic forms of creating processes and threads in operating system

During a fork, one flow of control becomes two.

Separate flows are “independent”
It means that the 2 flows of control “are not constrained to do similar computation”

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



Fork-Join

- Fork-Join dependency rules
  - A parent must join with its forked children
  - Forked children with the same parent can join with the parent in any order
  - A child can not join with its parent until it has joined with all of its children
- Typically, fork operates by assigning the child thread with some piece of “work”
  - Child work is usually specified by providing the child with a function to call on startup
  - Nature of the child work relative to the parent is not specified
- Join informs the parent that the child has finished
  - Child thread notifies the parent and then exits
  - Might provide some status back to the parent

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

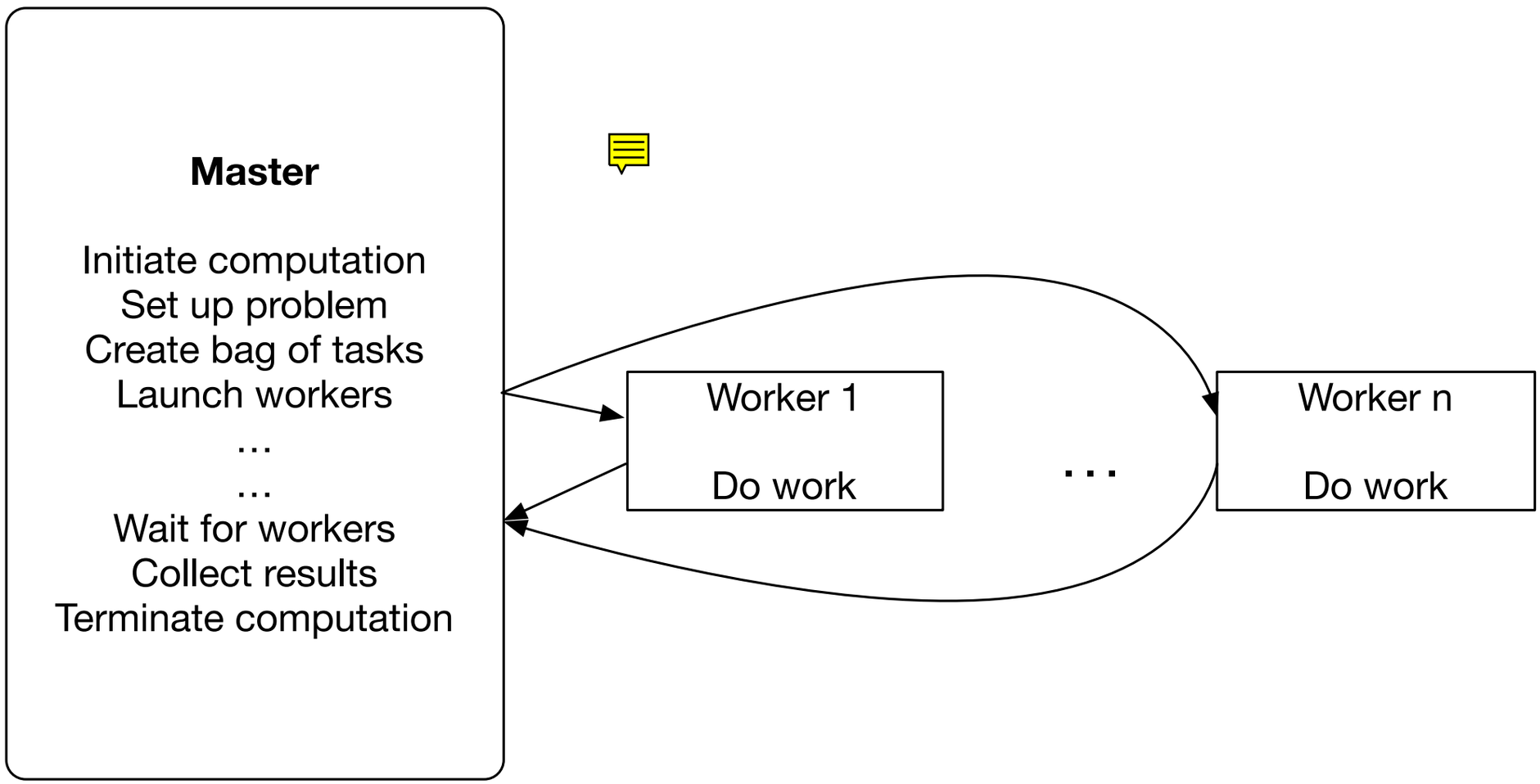
- A **master** process or thread set up a pool of **worker** processes of threads and a bag of tasks (often managed with a queue).
- The workers execute concurrently, with each worker repeatedly removing a tasks from the bag of the tasks.
- Workers request a new task as they finish their assigned work: load is automatically balanced between a collection of workers.
- Appropriate for “embarrassingly parallel problems”
- Other name: master-slave
- Variation: producer-consumer

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.

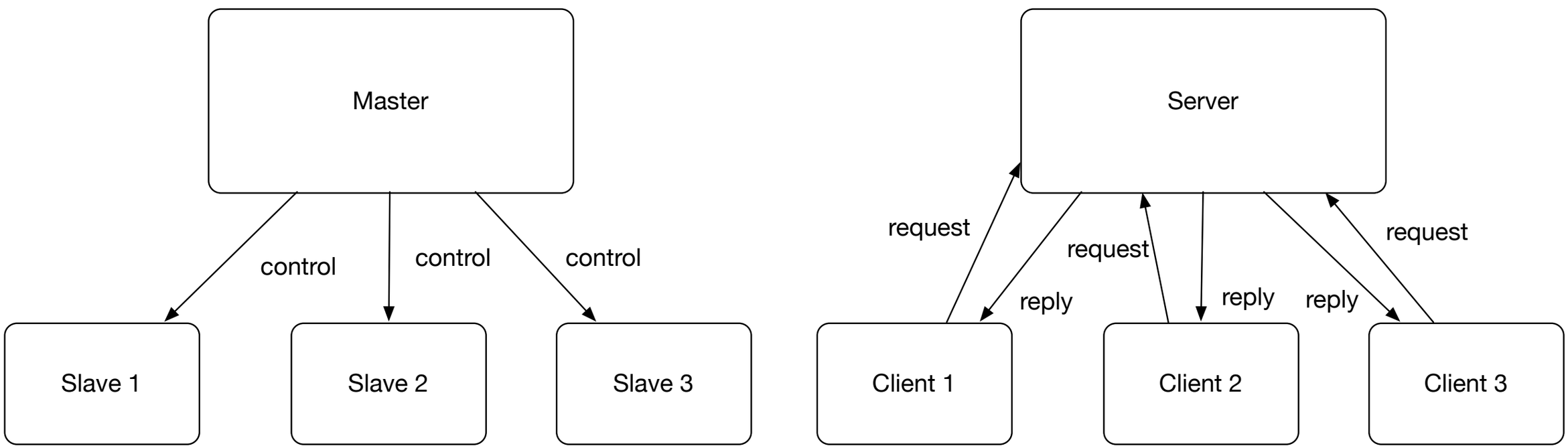
Master-worker



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-worker



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.

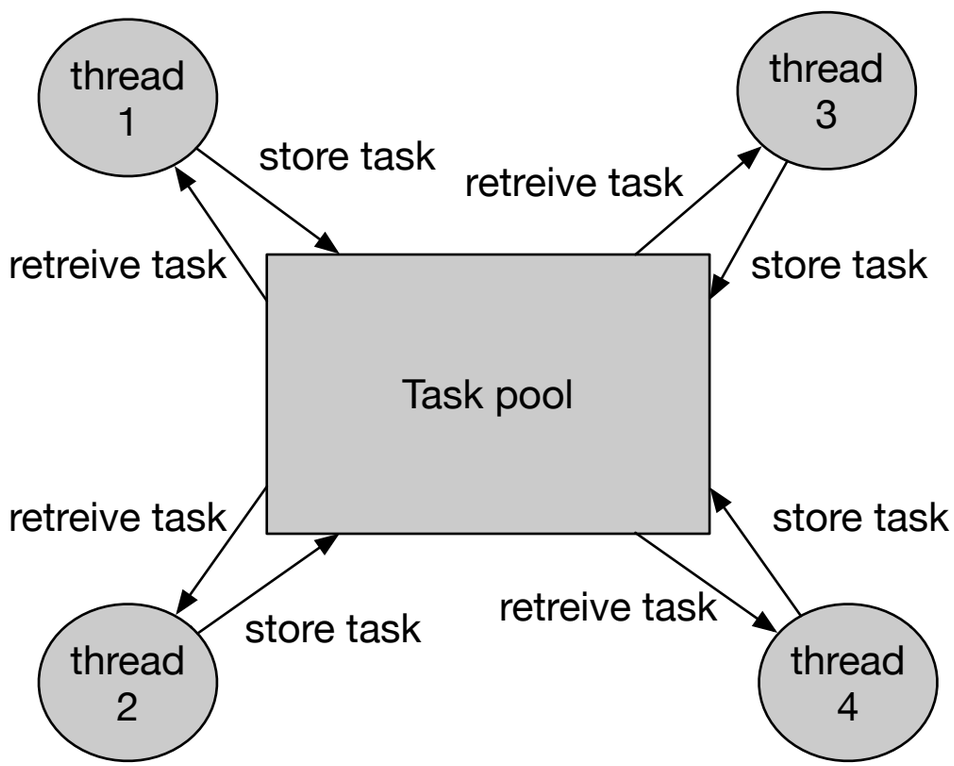
Producer-Consumer

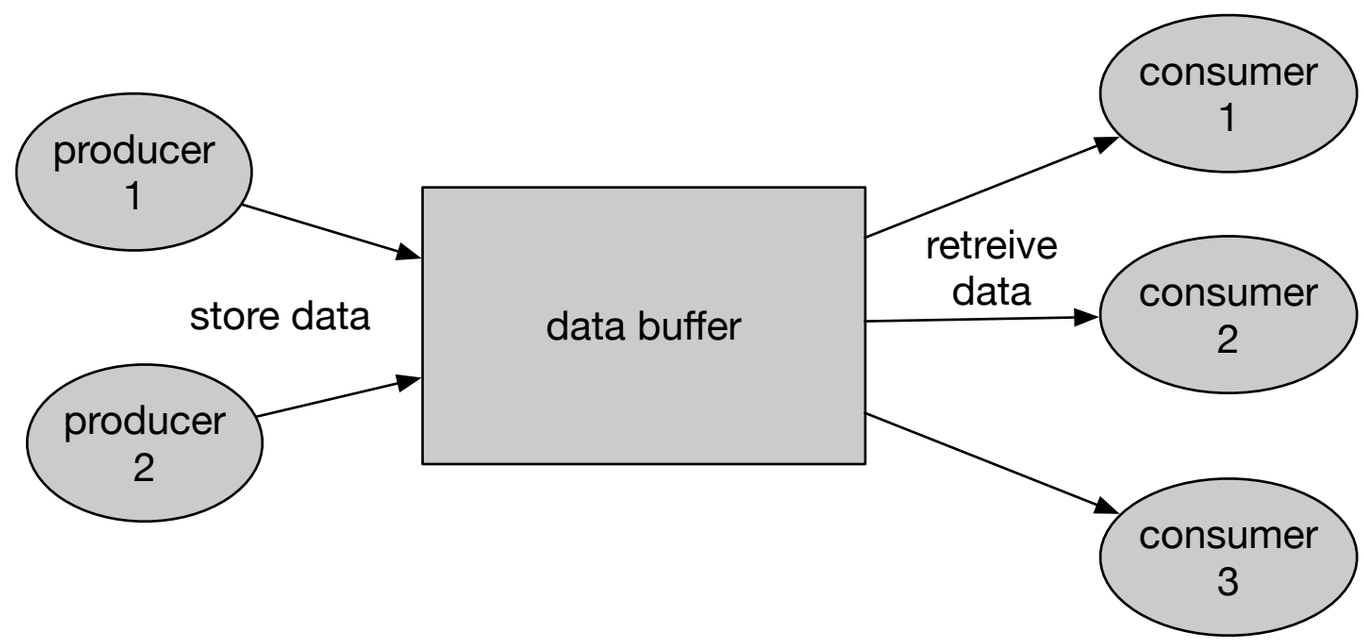
- **Producer** threads produce data which are used as input by **consumer** threads.

- 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



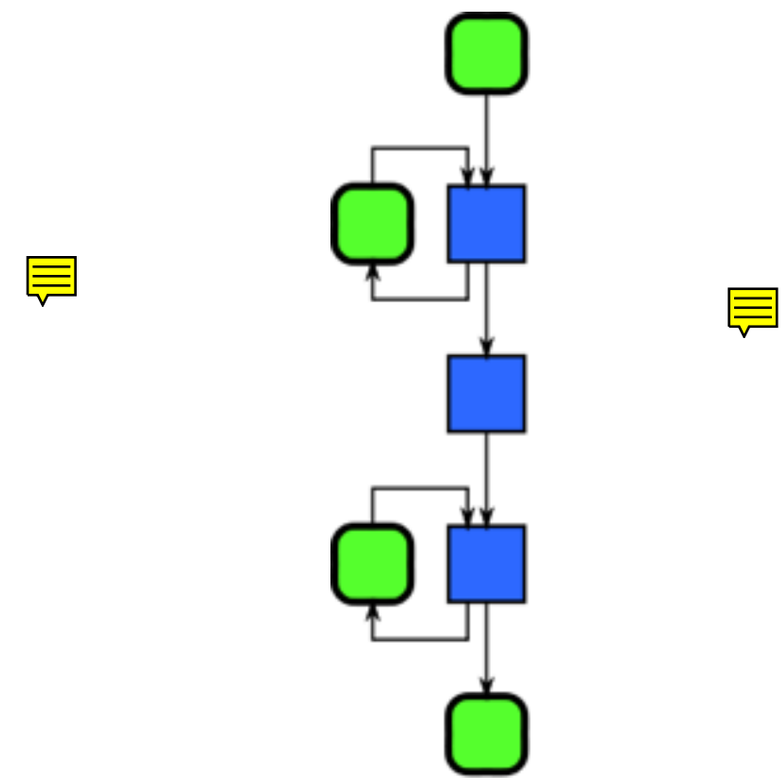


Work queue

- 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.

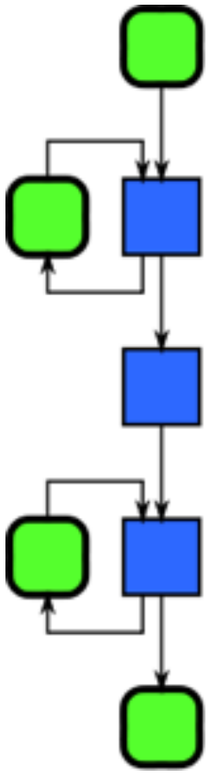
Pipeline

- Pipeline uses a sequence of stages that transform a flow of data
- Some stages may retain state
- Data can be consumed and produced incrementally: “online”



Pipeline

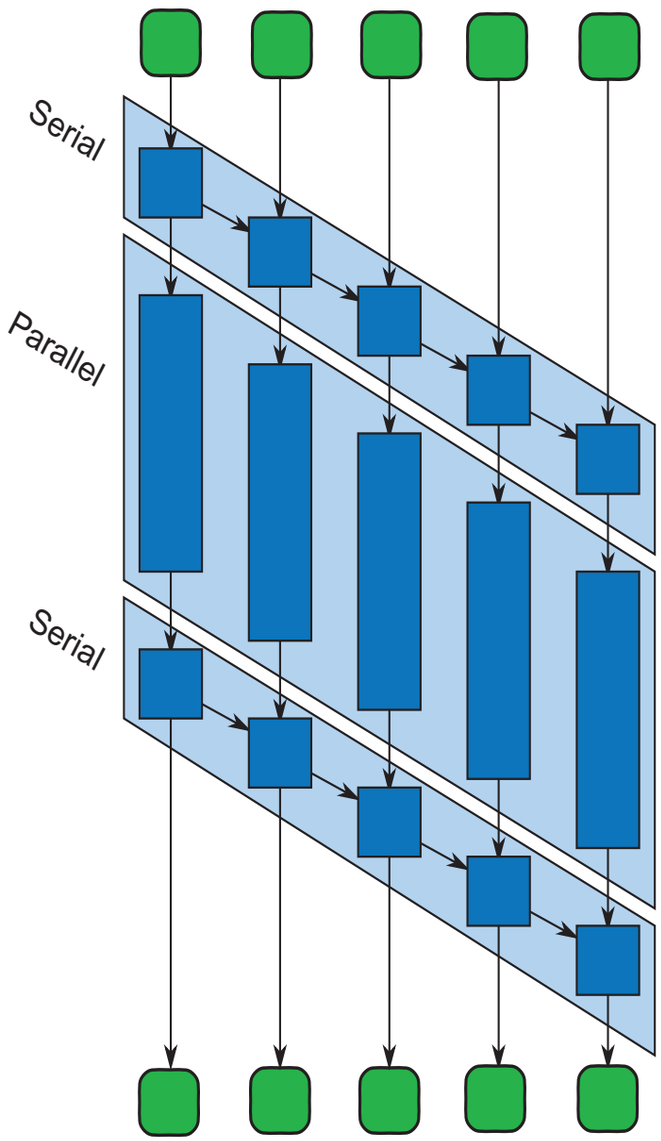
- Pipeline uses a sequence of stages that transform a flow of data
- Some stages may retain state
- Data can be consumed and produced incrementally: “online”



Examples: image filtering, data compression and decompression, signal processing

Pipeline

- Special form of coordination of different threads in which data elements are forwarded from thread to thread to perform different processing steps.

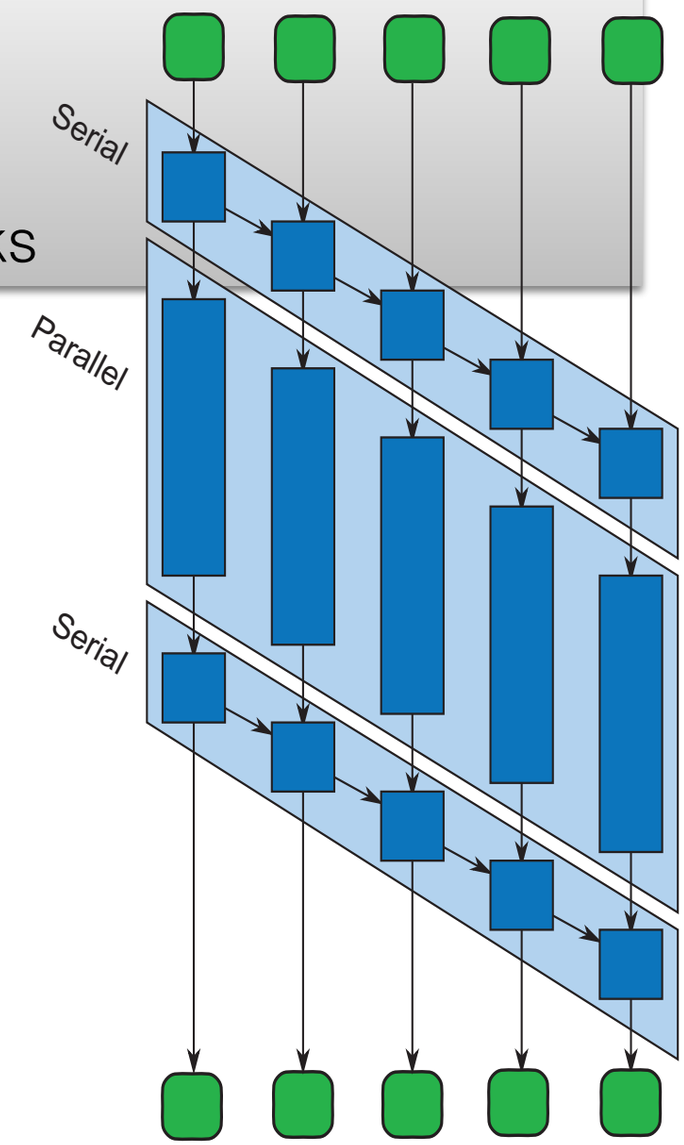


- can be considered as a special form of functional decomposition where the pipeline threads process the computations of an application algorithm one after another.
  A parallel execution is obtained by partitioning the data into a stream of data elements which flow through the pipeline stages one after another.

Pipeline

Pipeline computation is a special form of producer-consumer parallelism:

Producer tasks output data …
… used as input by consumer tasks



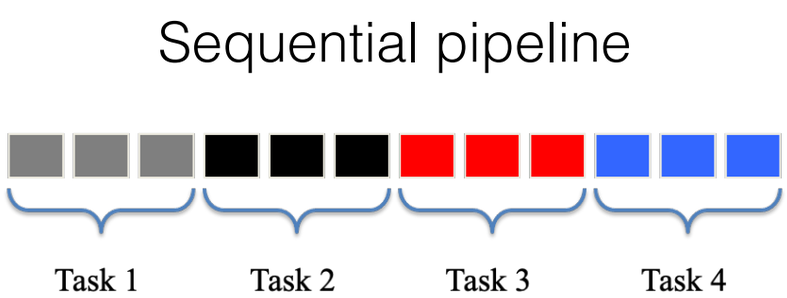
- can be considered as a special form of functional decomposition where the pipeline threads process the computations of an application algorithm one after another.
  A parallel execution is obtained by partitioning the data into a stream of data elements which flow through the pipeline stages one after another.

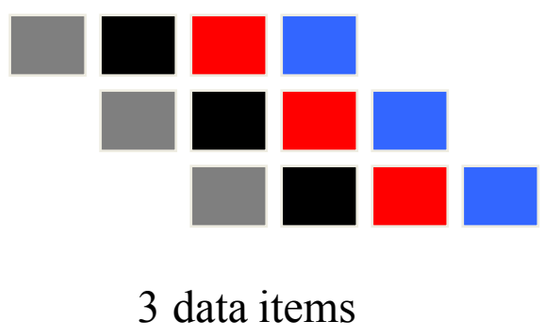
Pipeline

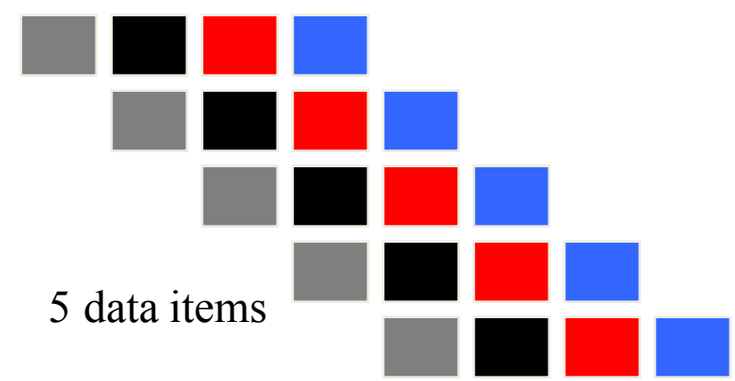
- Two parallel execution choices:

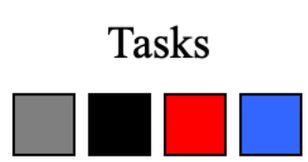
  1. processor executes the entire pipeline

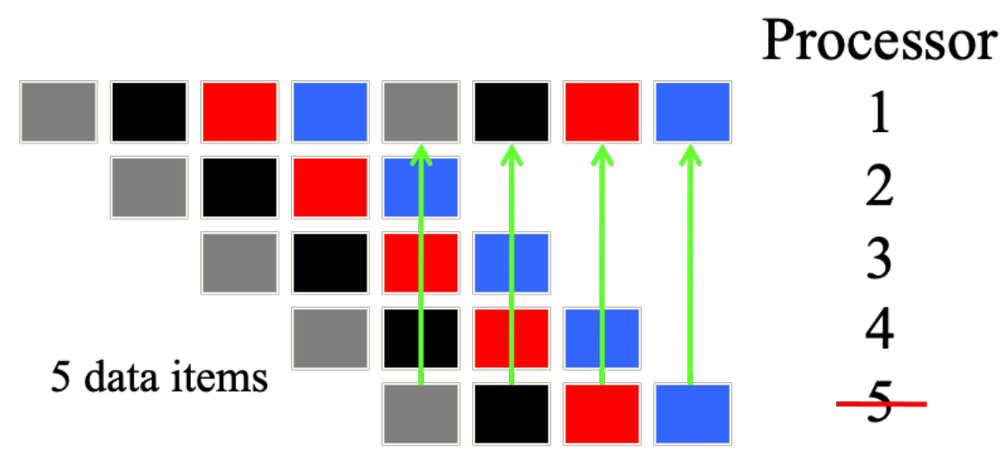
  2. processor assigned to execute a single task









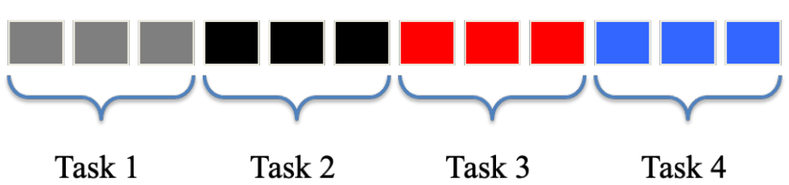


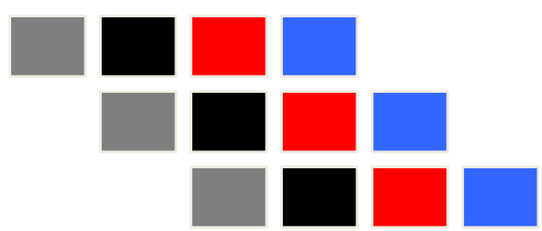
Pipeline

Suggestions:

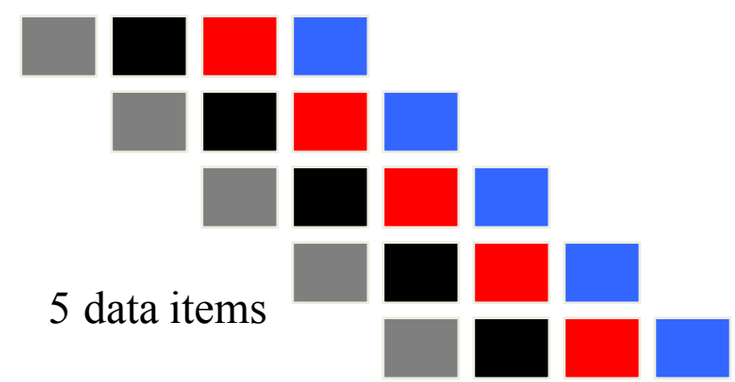
- Try to find a lot of data to pipeline
- Try to divide computation in a lot of pipeline tasks
  - More tasks to do (longer pipelines)
  - Break a larger task into more (shorter) tasks to do

Sequential pipeline



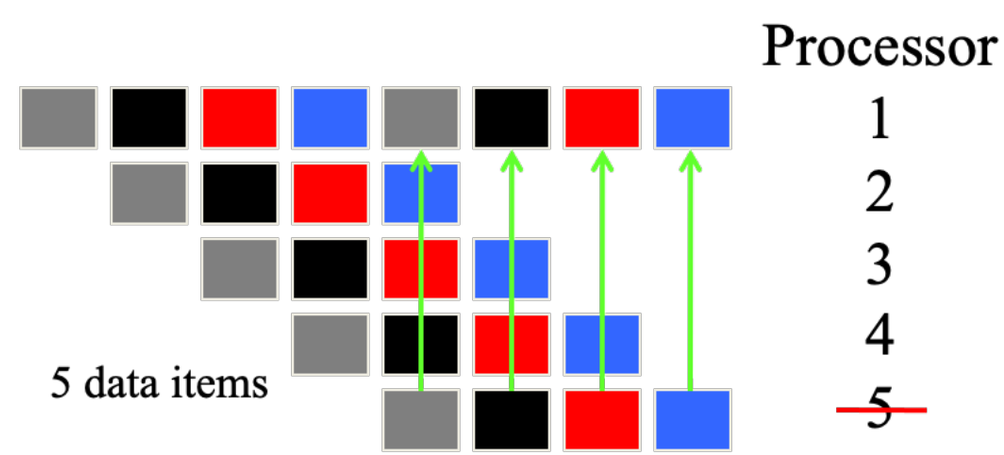


3 data items



5 data items

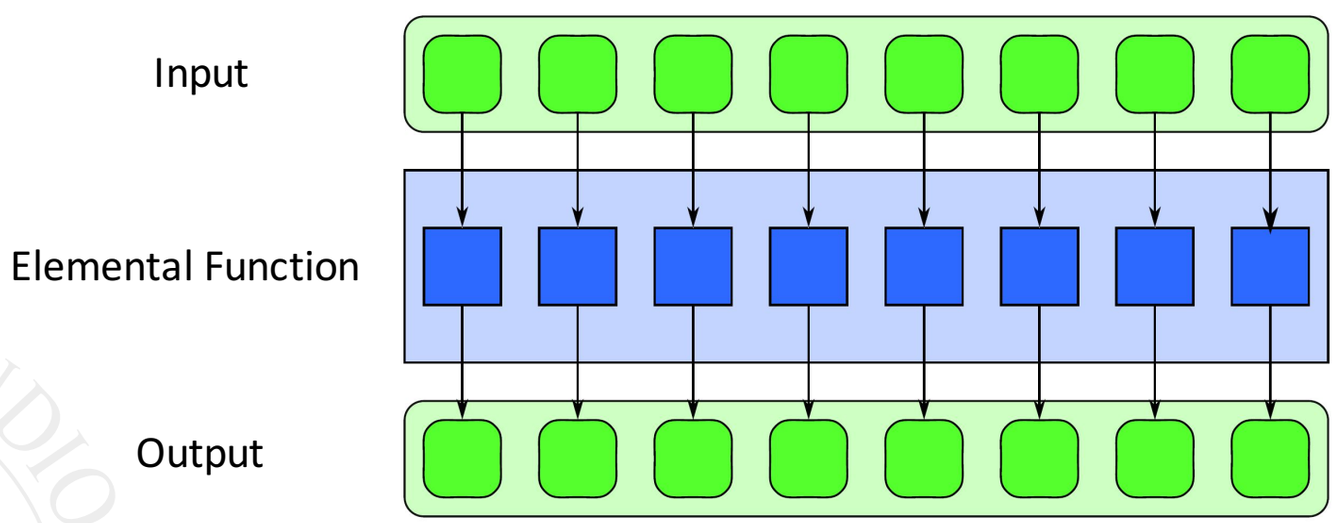
Tasks



5 data items

Map

- Map: performs a function over every element of a collection
- Map replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection
- The replicated function is referred to as an “elemental function”

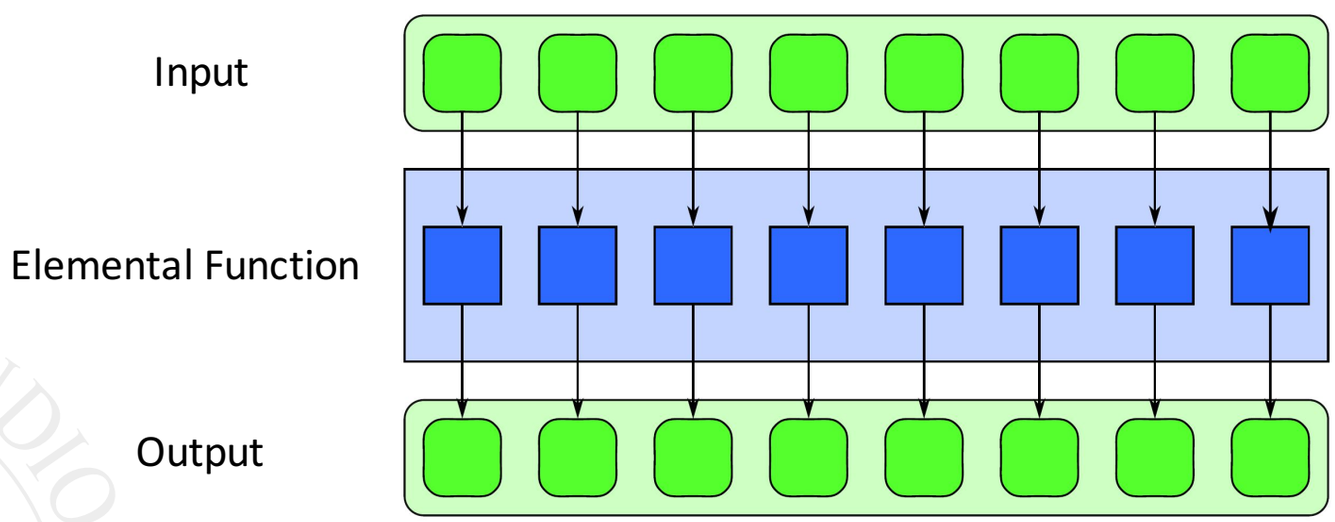


Map

- Map: performs a function over every element of a collection

Map is a “foreach loop” where each iteration is independent
We can run map completely in parallel…
Example: SAXPY (i.e. scaled vector addition):
z = ax + y
Scales vector x by a and adds it to vector y - since every element in vector x and vector y are independent we can compute z[i] = a * x[i] + y[i] parallelizing the for loop on i

- The replicated function is referred to as an “elemental function”

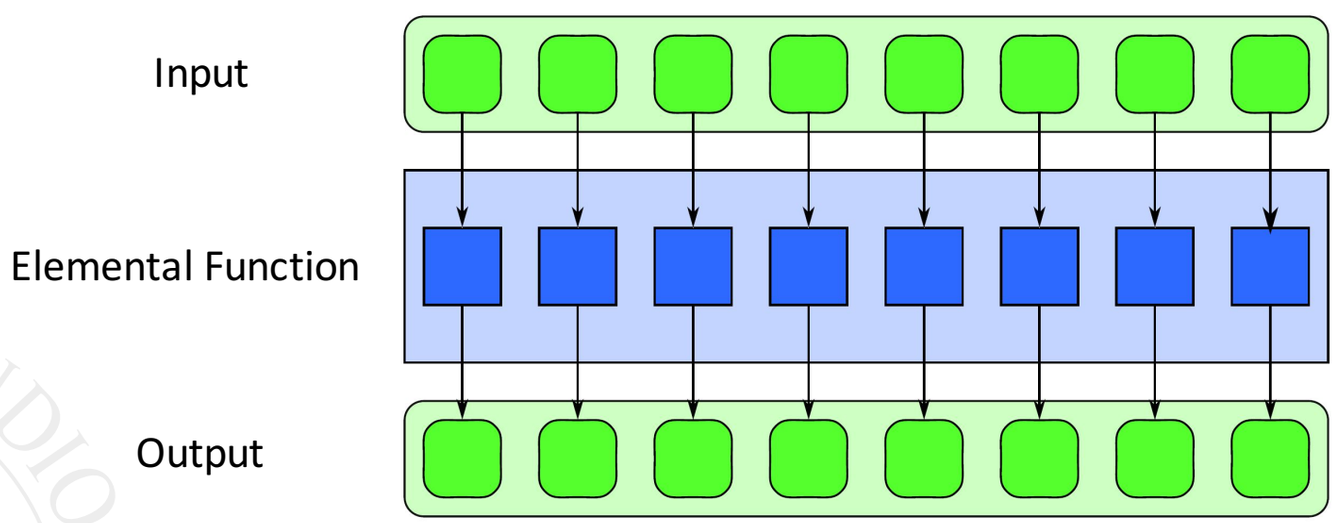


Map

- Map: performs a function over every element of a collection

Map is a “foreach loop” where each iteration is independent
We can run map completely in parallel…
Example: SAXPY (i.e. scaled vector addition):
$z = ax + y$
Scales vector x by a and adds it to vector y - since every element in vector x and vector y are independent we can compute z[i] = a * x[i] + y[i] parallelizing the for loop on i

Examples: gamma correction and thresholding in images; color space conversions; Monte Carlo sampling; ray tracing.

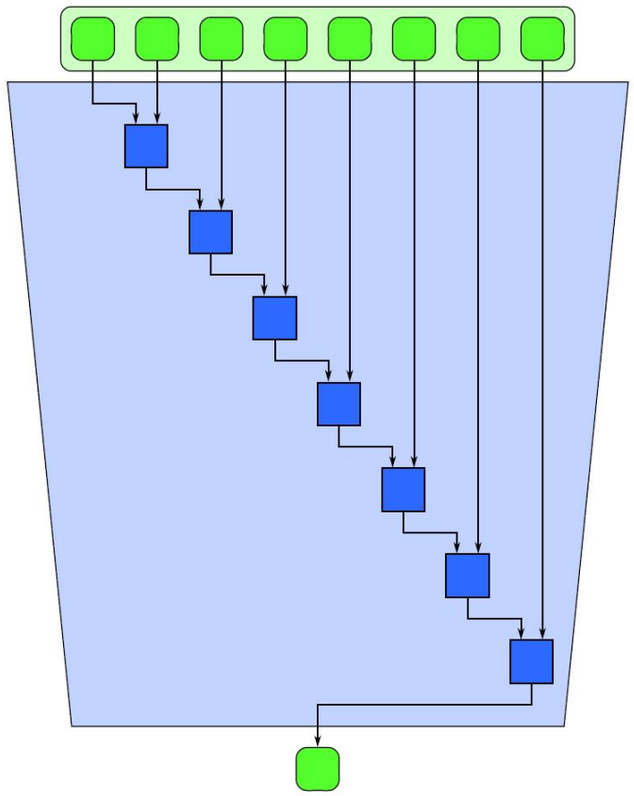


Reduction

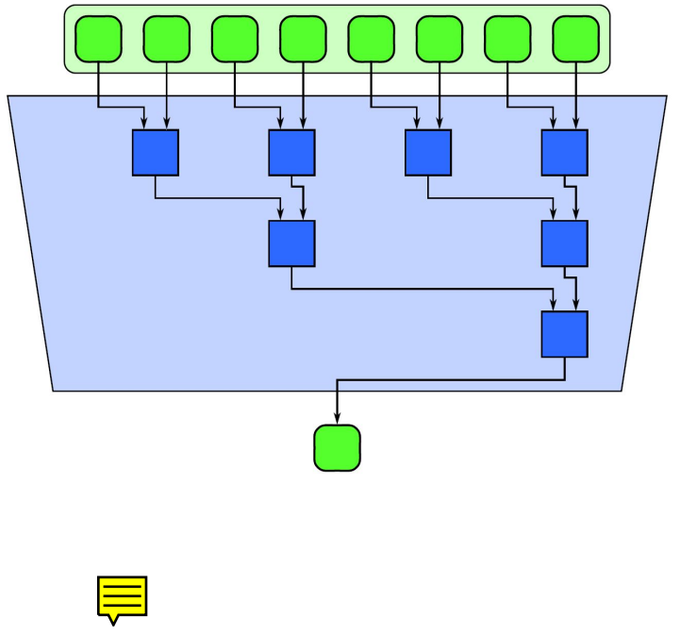
- Reduction: Combines every element in a collection using an associative "combiner function"
- Because of the associativity of the combiner function, different orderings of the reduction are possible
- Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR, and XOR

Reduction

Serial Reduction

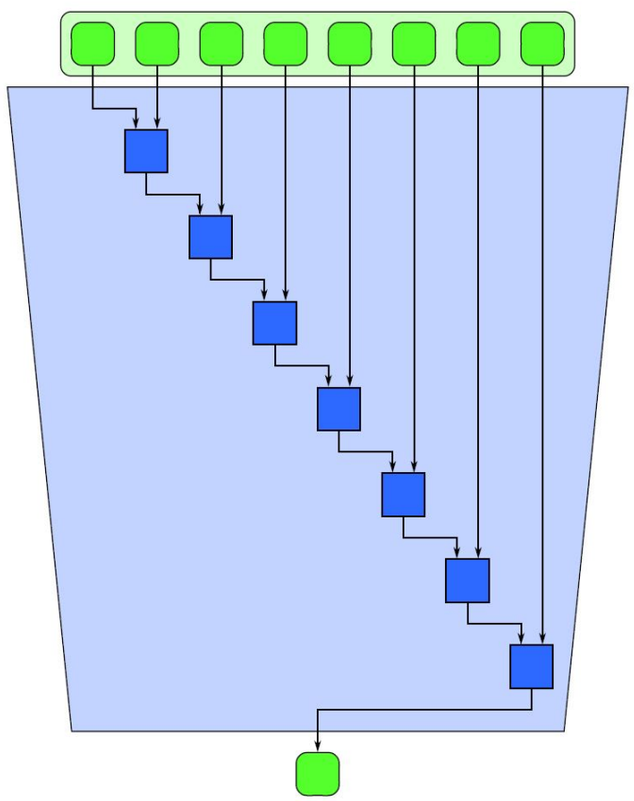


Parallel Reduction

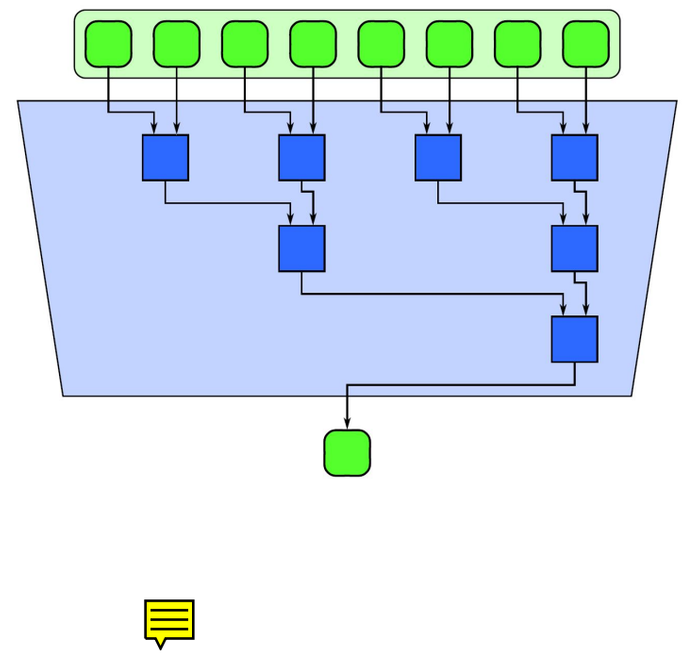


Reduction

Serial Reduction



Parallel Reduction



Examples: averaging of Monte Carlo samples; convergence testing; image comparison metrics; matrix operations.

Combining map and reduce

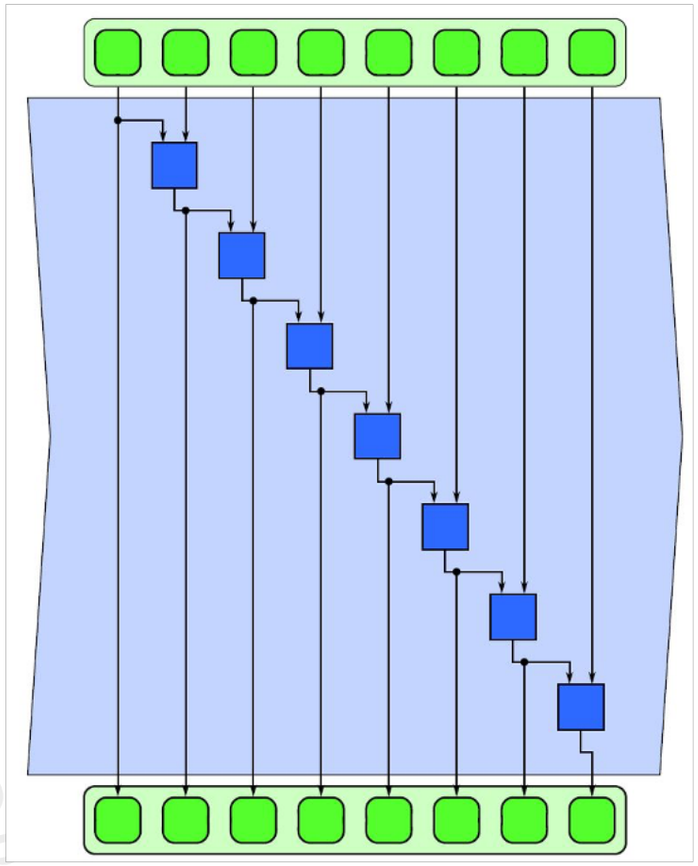
- We can “fuse” the map and reduce patterns
- E.g.: dot product of two vectors:
  - Map (*) to multiply the components
  - Then reduce with (+) to get the final answer

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i.$$

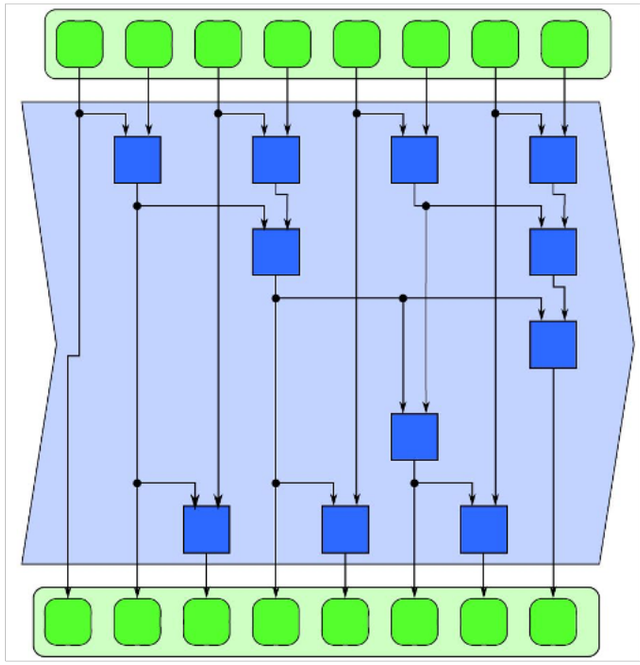
Scan

- Scan computes all partial reductions of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- Operator must be (at least) associative to parallelize
- A parallel scan will require more operations than a serial version

Serial Scan



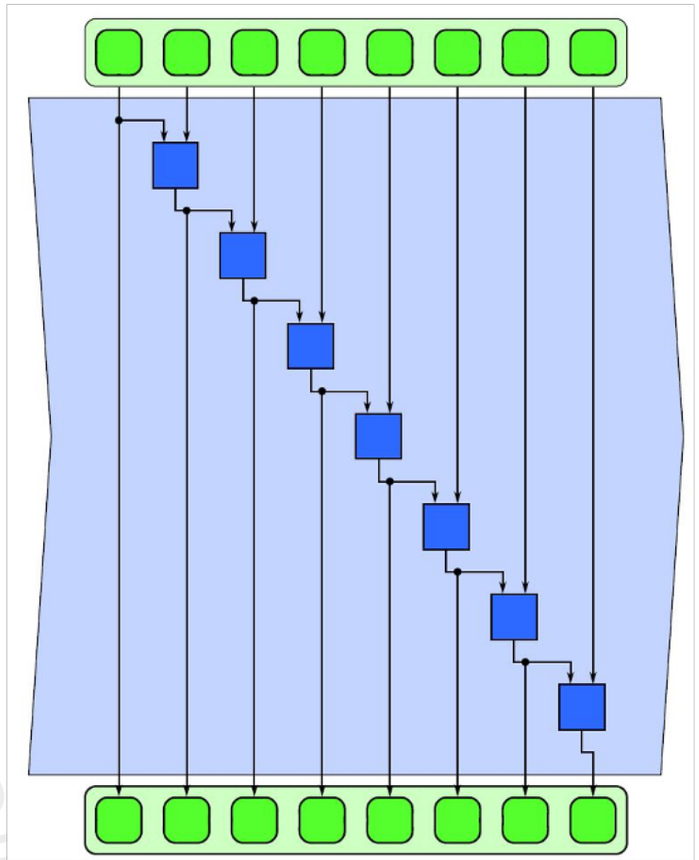
Parallel Scan



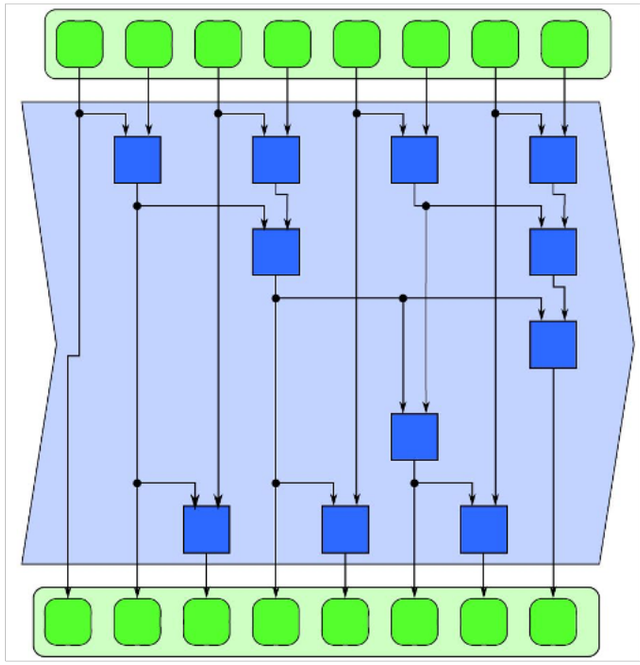
Scan

- Scan computes all partial reductions of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- Operator must be (at least) associative to parallelize
- A parallel scan will require more operations than a serial version

Serial Scan



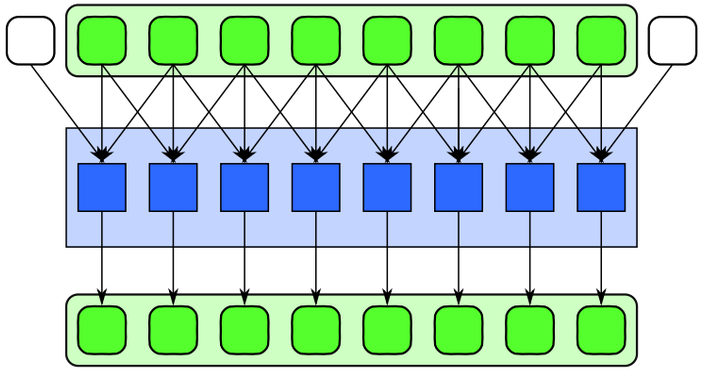
Parallel Scan

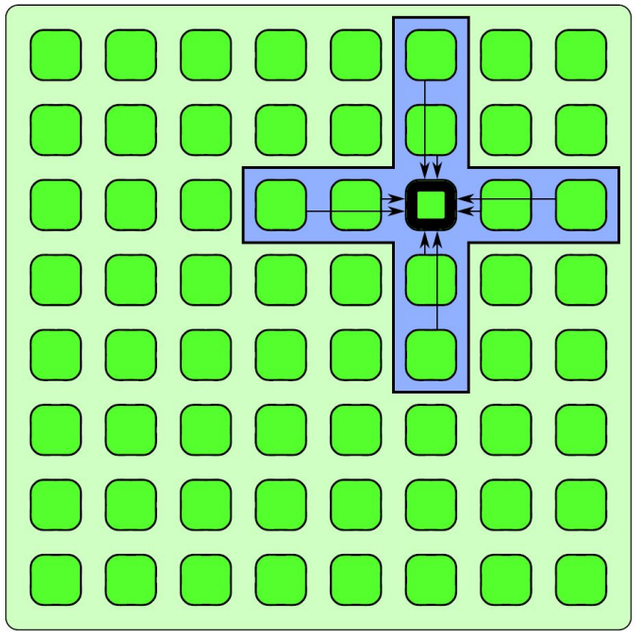


Used in time series analysis

Stencil

- Stencil: Elemental function accesses a set of “neighbors”, stencil is a generalization of map
  - Each instance of the map function accesses neighbors of its input, offset from its usual input. It is a map where each output depends on a “neighborhood” of inputs
- Often combined with iteration – used with iterative solvers or to evolve a system through time
- Boundary conditions must be handled carefully in the stencil pattern

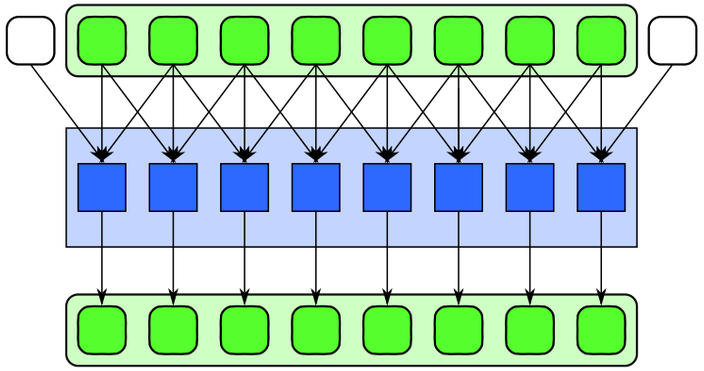


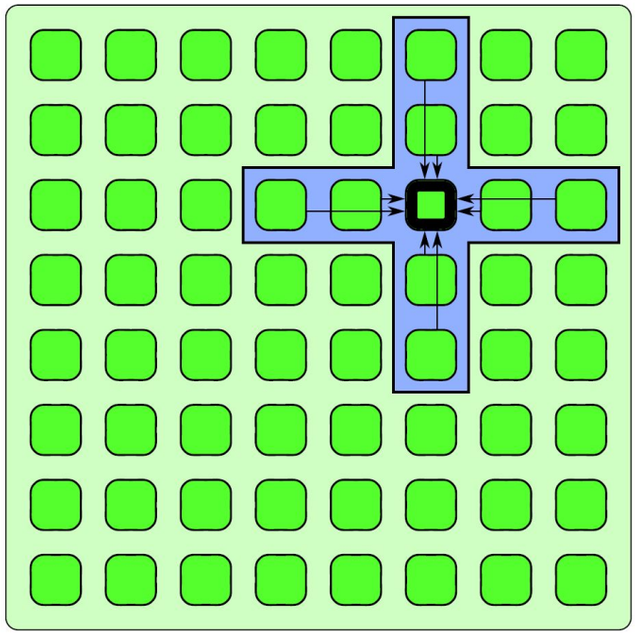


Stencil

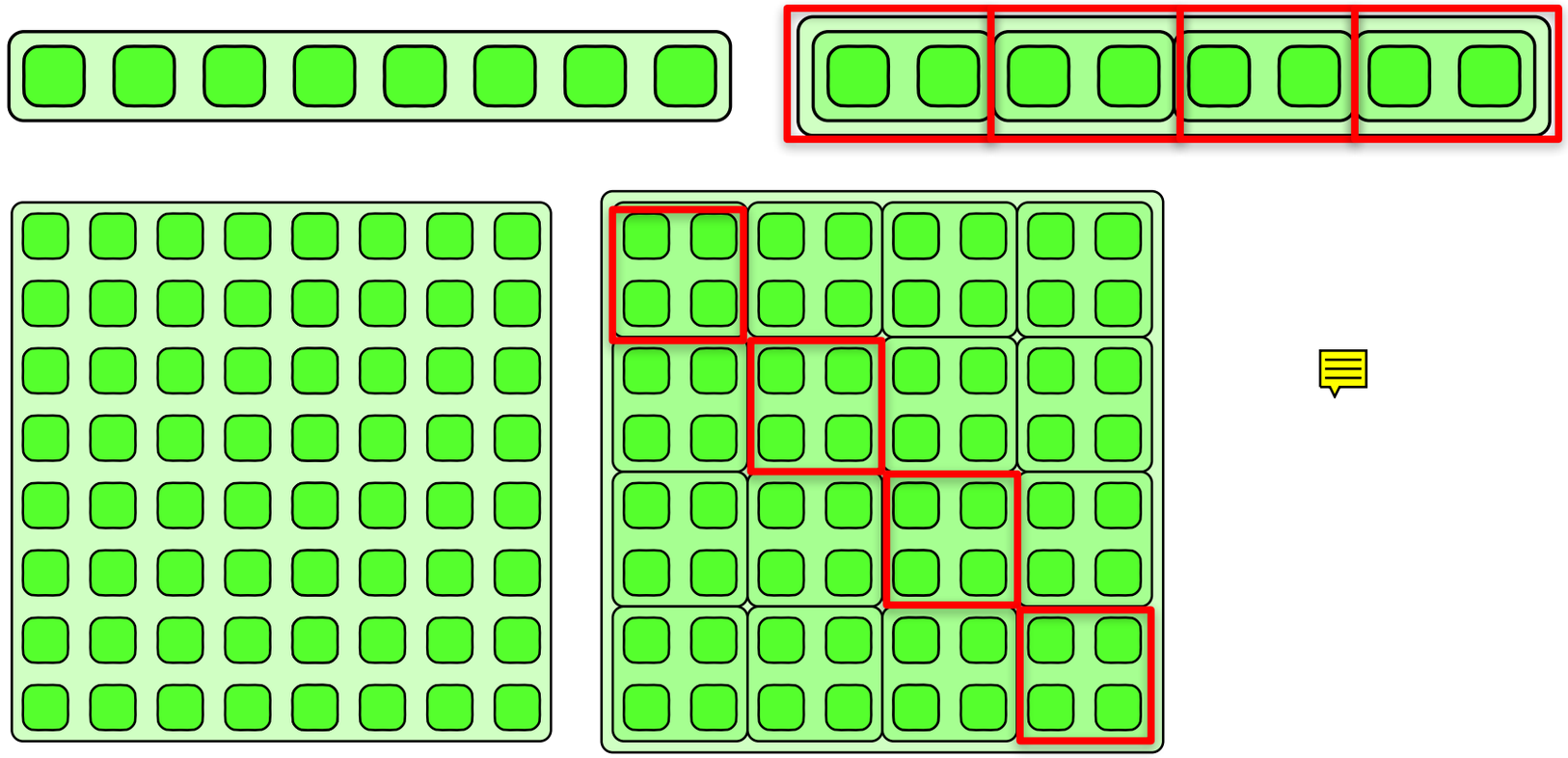
Examples: signal filtering including convolution, median, anisotropic diffusion

- Stencil: Elemental function accesses a set of "neighbors", stencil is a generalization of map
  - Each instance of the map function accesses neighbors of its input, offset from its usual input. It is a map where each output depends on a "neighborhood" of inputs
  - Often combined with iteration – used with iterative solvers or to evolve a system through time
  - Boundary conditions must be handled carefully in the stencil pattern





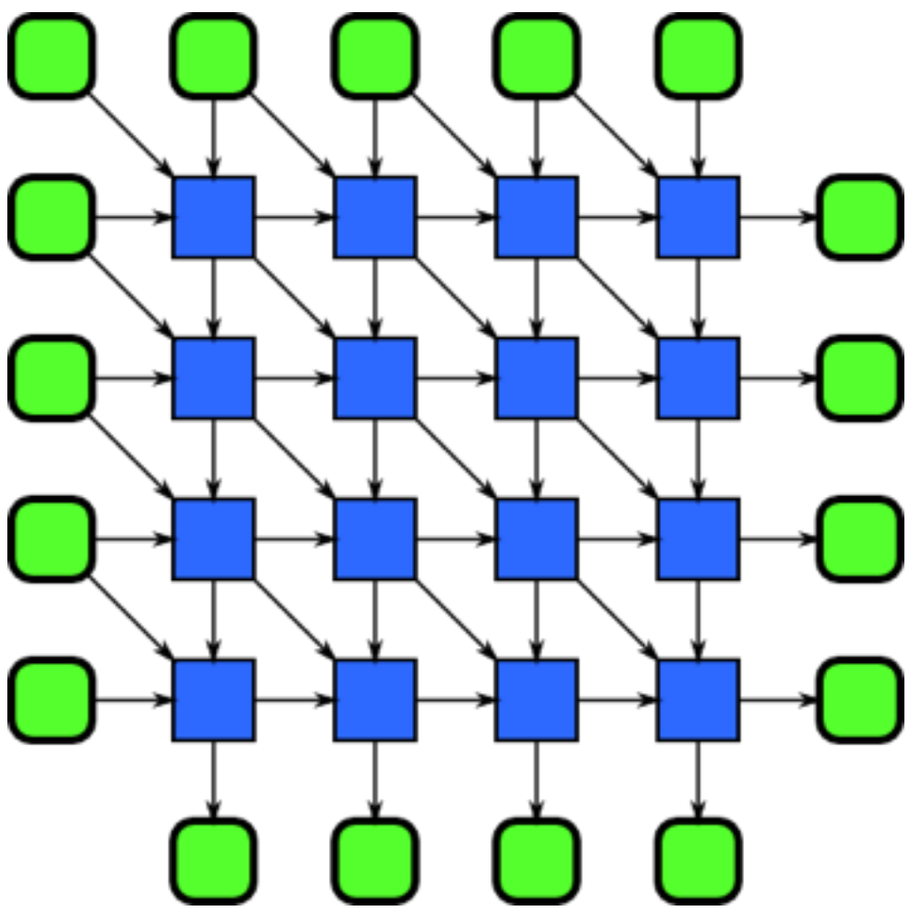
Stencil



- Data is divided into
  - non-overlapping regions (avoid write conflicts, race conditions)
  - equal-sized regions (improve load balancing)
- It is important to divide data to be cache friendly, to avoid false sharing and redundant reads

Recurrence

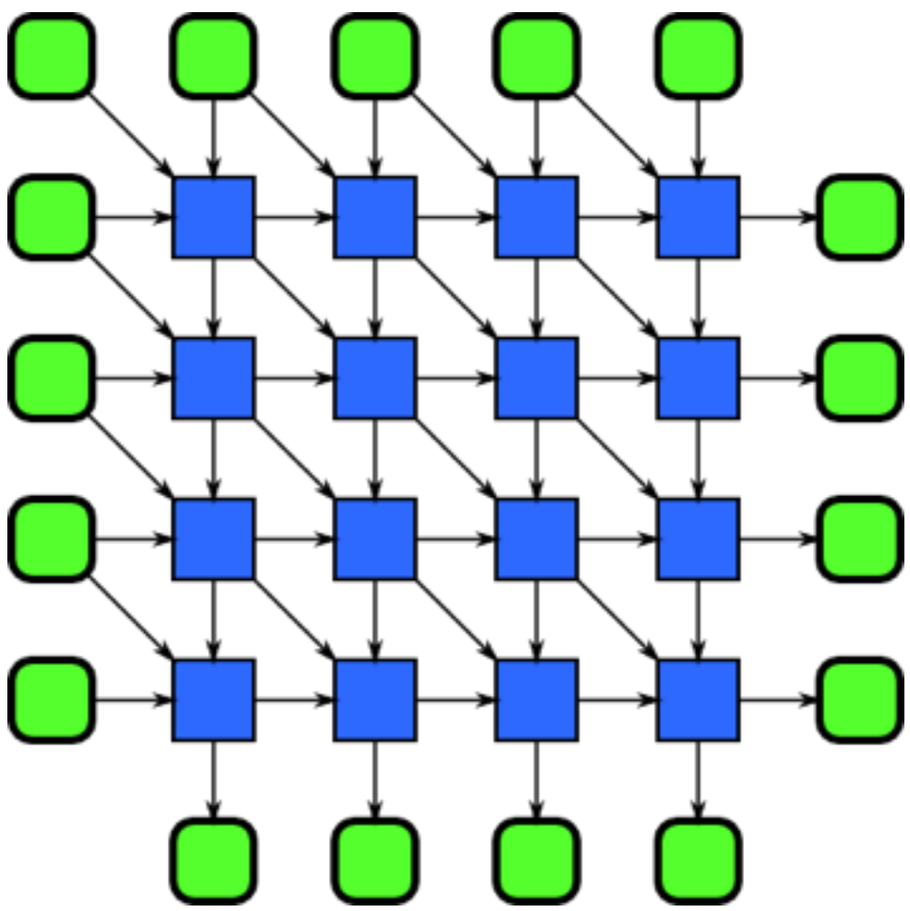
- Recurrence results from loop nests with both input and output dependencies between iterations•
- Can also result from iterated stencils



Recurrence

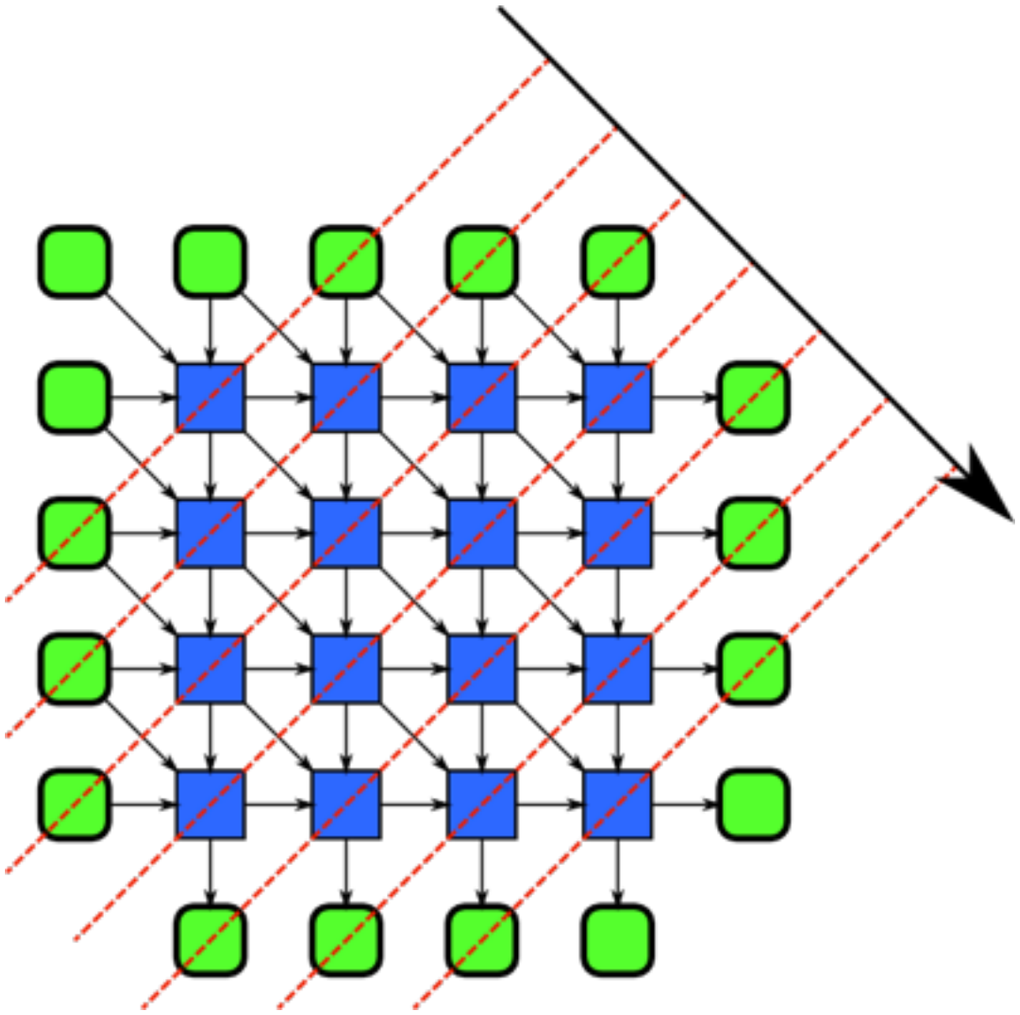
- Recurrence results from loop nests with both input and output dependencies between iterations•

Examples: Simulation including fluid flow, electromagnetic, and financial Partial Differential Equation solvers, sequence alignment and pattern matching



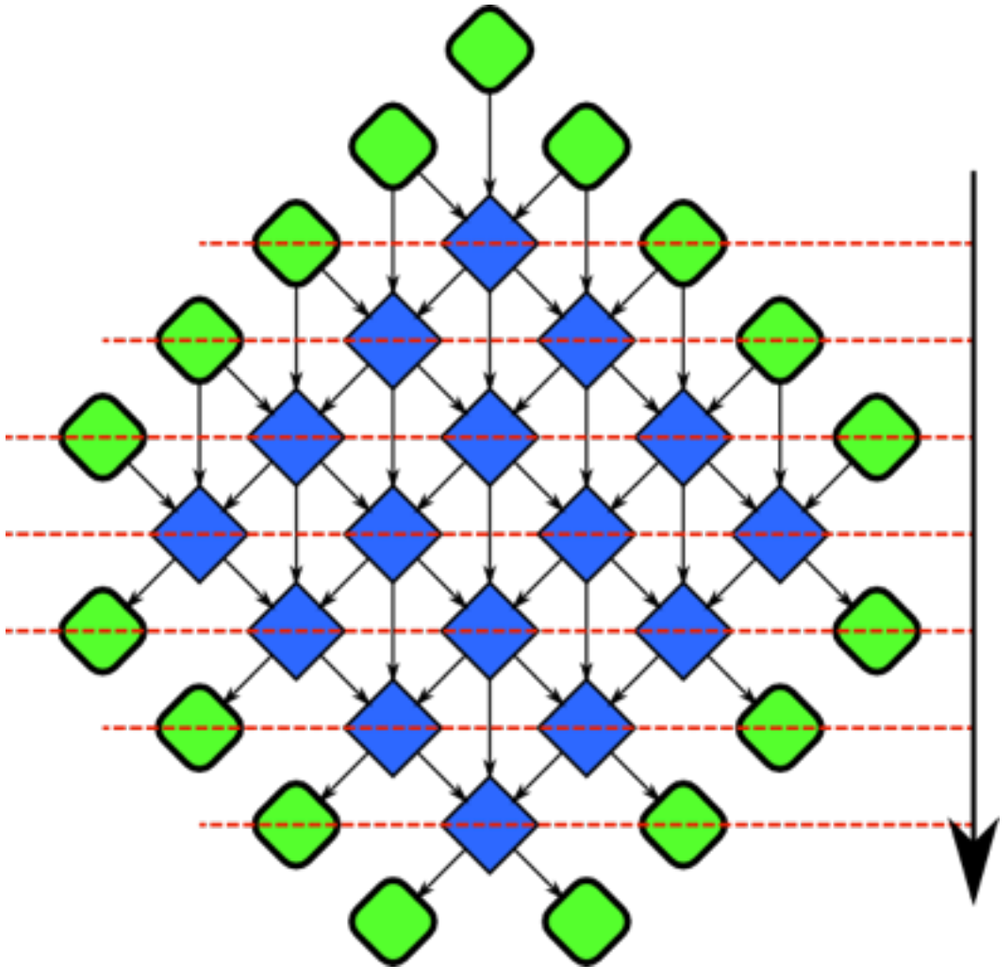
Recurrence Hyperplane Sweep

- Multidimensional recurrences can always be parallelized
- Leslie Lamport's hyperplane separation theorem:
  - Choose hyperplane with inputs and outputs on opposite sides
  - Sweep through data perpendicular to hyperplane



Recurrence Hyperplane Sweep

- Multidimensional recurrences can always be parallelized
- Leslie Lamport's hyperplane separation theorem:
  - Choose hyperplane with inputs and outputs on opposite sides
  - Sweep through data perpendicular to hyperplane



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.

Note: gather is also supported in AVX2 (since 2013)
Scatter has been added in AVX512, in AVX2 it must be implemented as scalar loop

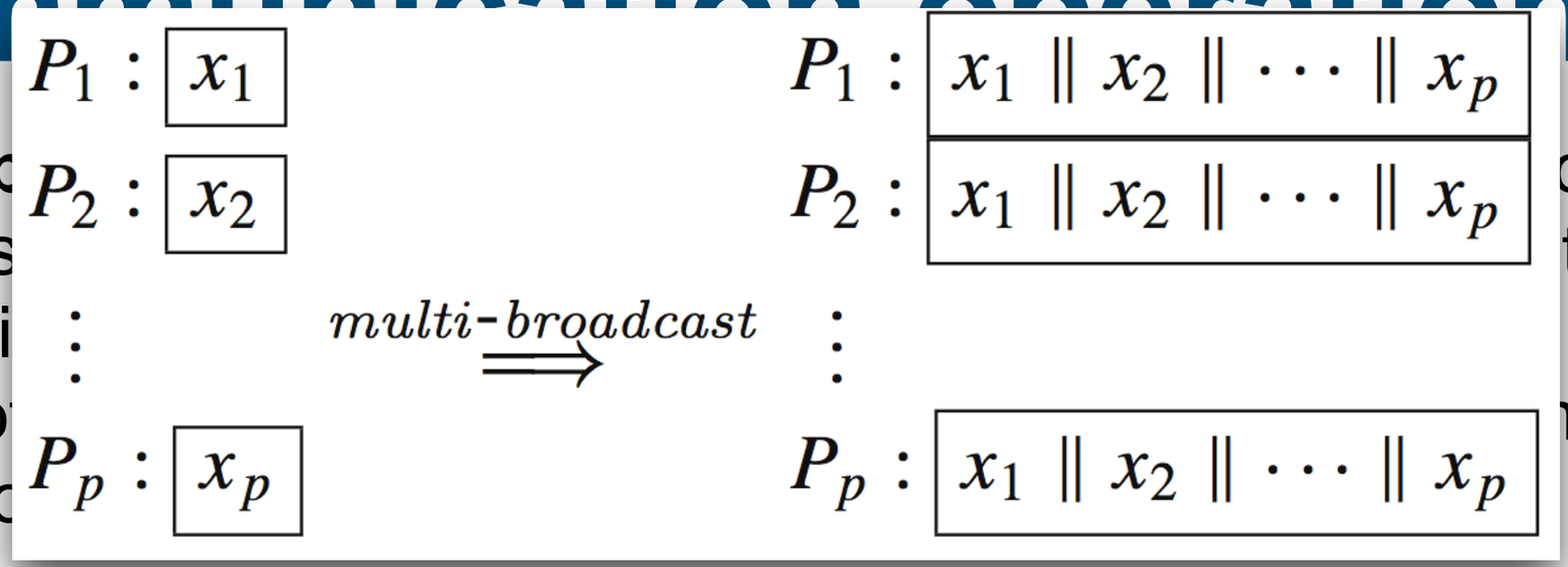
Communication operations

- **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



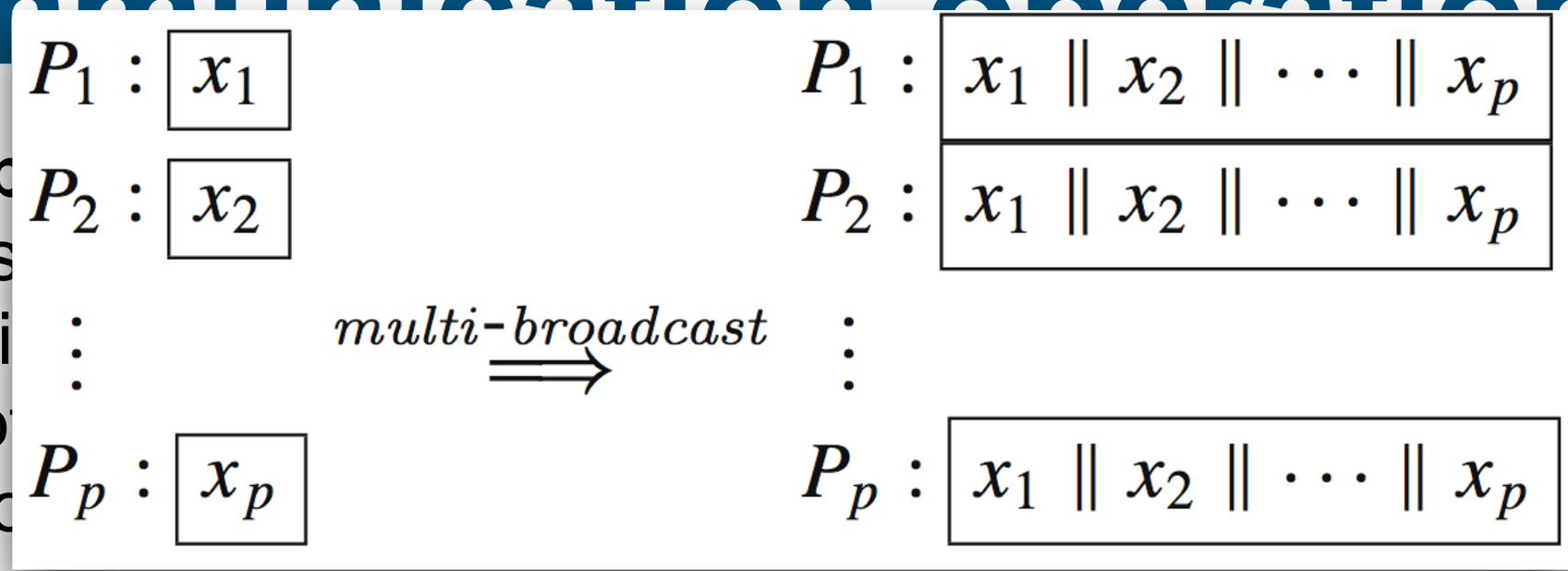
- **Multi-b** on is the s t operati point o n every c

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

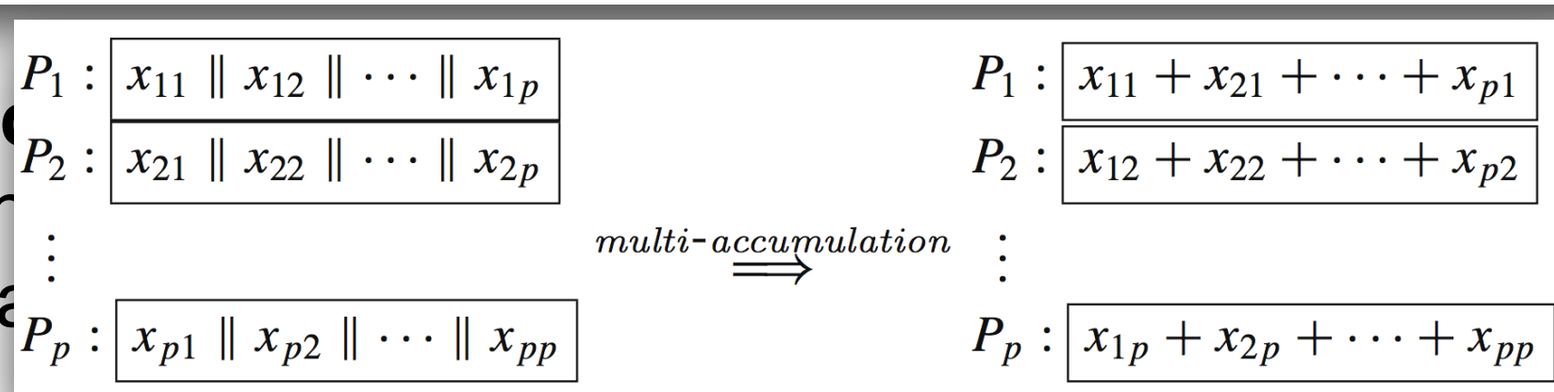
- **Total exchange**: for a total exchange operation, each processor provides for each other processor a potentially different data block. These data blocks are sent to their intended receivers, i.e., each processor executes a **scatter** operation.

Communication operations

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



- **Multi-acc** n operation accumula



- **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

$$\begin{array}{ll} P_1 : \boxed{x_1} & P_1 : \boxed{x_1 \parallel x_2 \parallel \cdots \parallel x_p} \\ P_2 : \boxed{x_2} & P_2 : \boxed{x_1 \parallel x_2 \parallel \cdots \parallel x_p} \\ \vdots & \vdots \\ P_p : \boxed{x_p} & P_p : \boxed{x_1 \parallel x_2 \parallel \cdots \parallel x_p} \end{array} \xrightarrow{\textit{multi-broadcast}}$$

- **Multi-acc** n
  operation
  accumula

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

- **Total** /
  proces
  differe
  intend er
  opera

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

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.
  - algorithms, functions, or procedures that contain a state cannot be executed concurrently.
- 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, typically 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, typically 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

- 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.

Sign of loop-carried dependences: typically, this situation will be evidenced by references to the same array element on both the left- and righthand sides of assignments and a backward reference in some righthand side use of the array element.

```
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

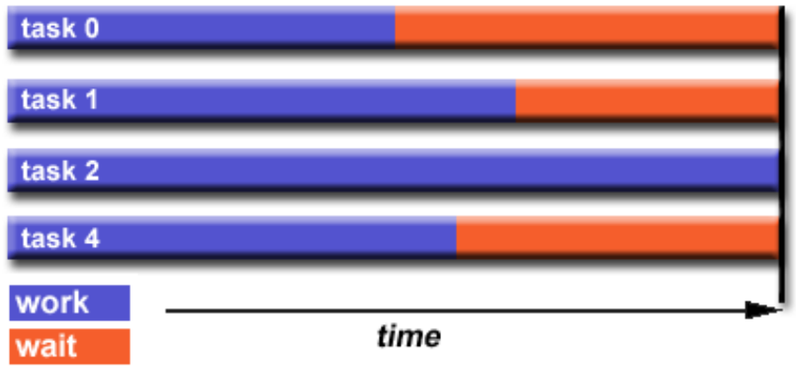
- Check that the library you use is thread-safe. It may contain some shared variable that causes data races.
- A library function is thread-safe if it can be called by different threads concurrently, without performing additional operations to avoid race conditions.

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++

Achieve load balance

- Load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.
- Depending on tasks it is possible to equally partition the work each task receives or use dynamic work assignment



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.

Design methodology

Top-Down

- We know that a system is composed of more than one sub-system and it contains a number of components. Further, these sub-systems and components may have their on set of sub-system and components and creates hierarchical structure in the system.

- **Top-down** design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top- down hierarchy is achieved.

- Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

- Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

Bottom-up

- The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.
- Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
- Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

Design spaces

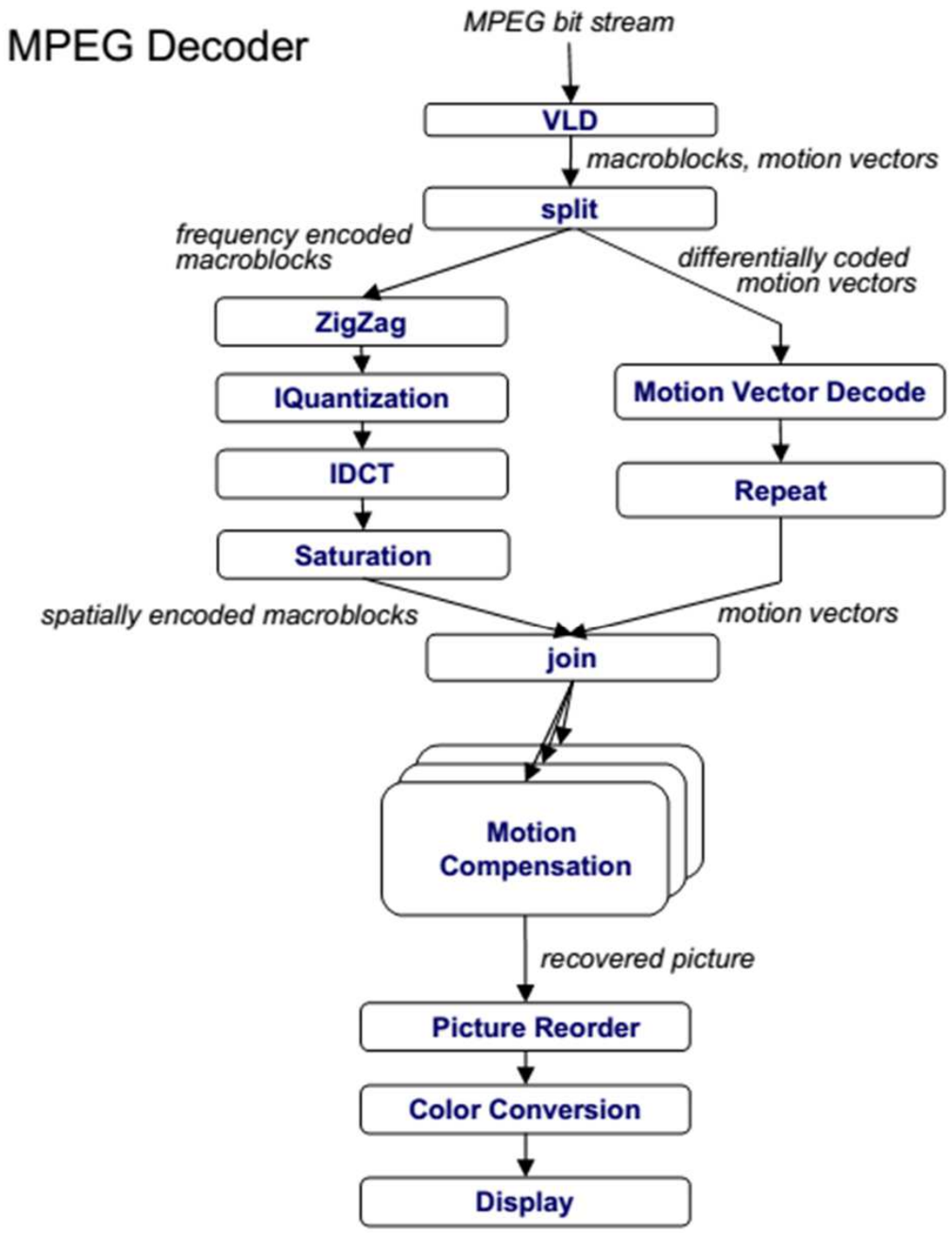
Algorithm Expression

- Finding Concurrency
  - Expose concurrent tasks
- Algorithm structure
  - Map tasks to processes to exploit parallel architecture

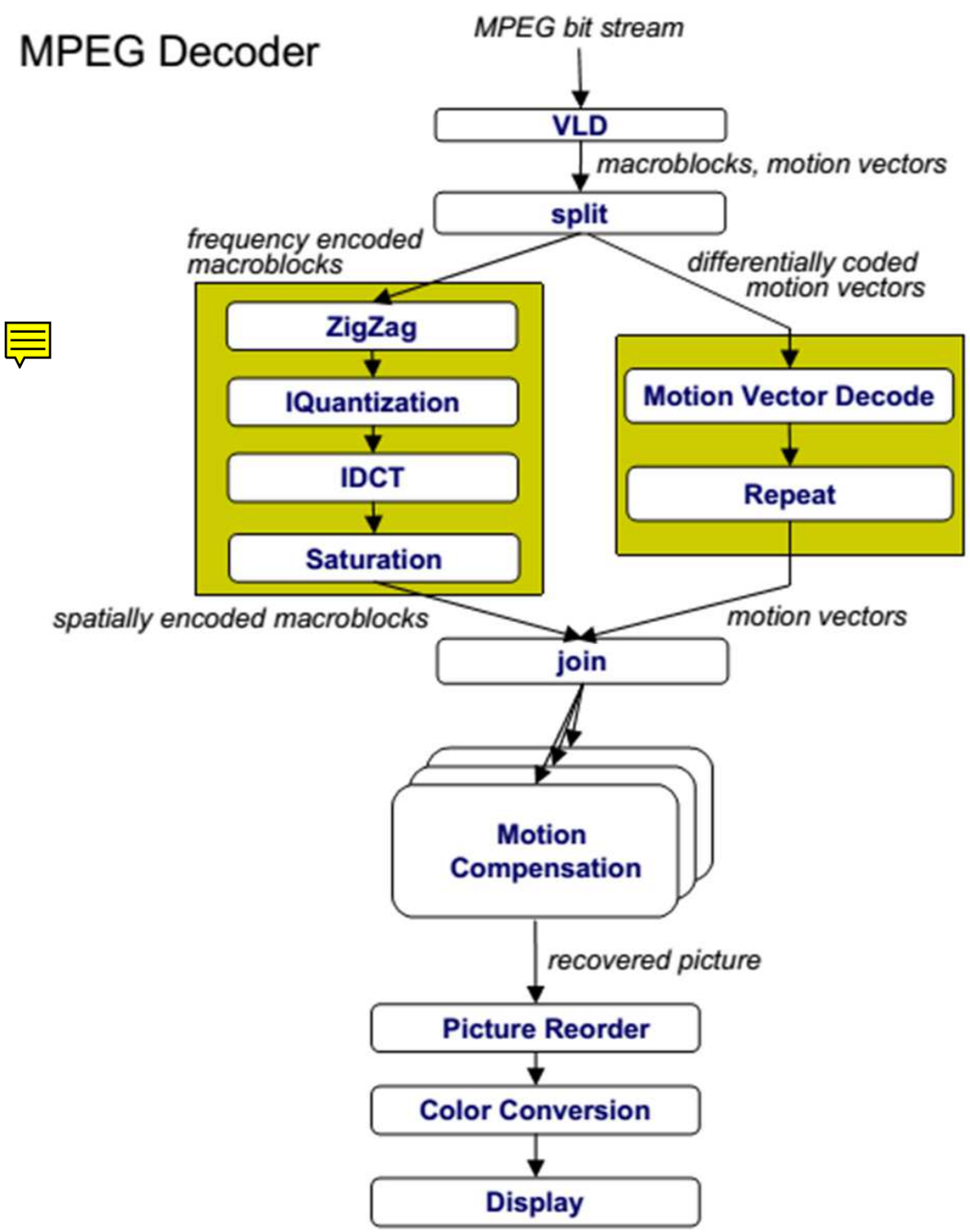
Software Construction

- Supporting Structures
  - Code and data structuring patterns
- Implementation Mechanisms
  - Low level mechanisms used to write parallel programs

Example

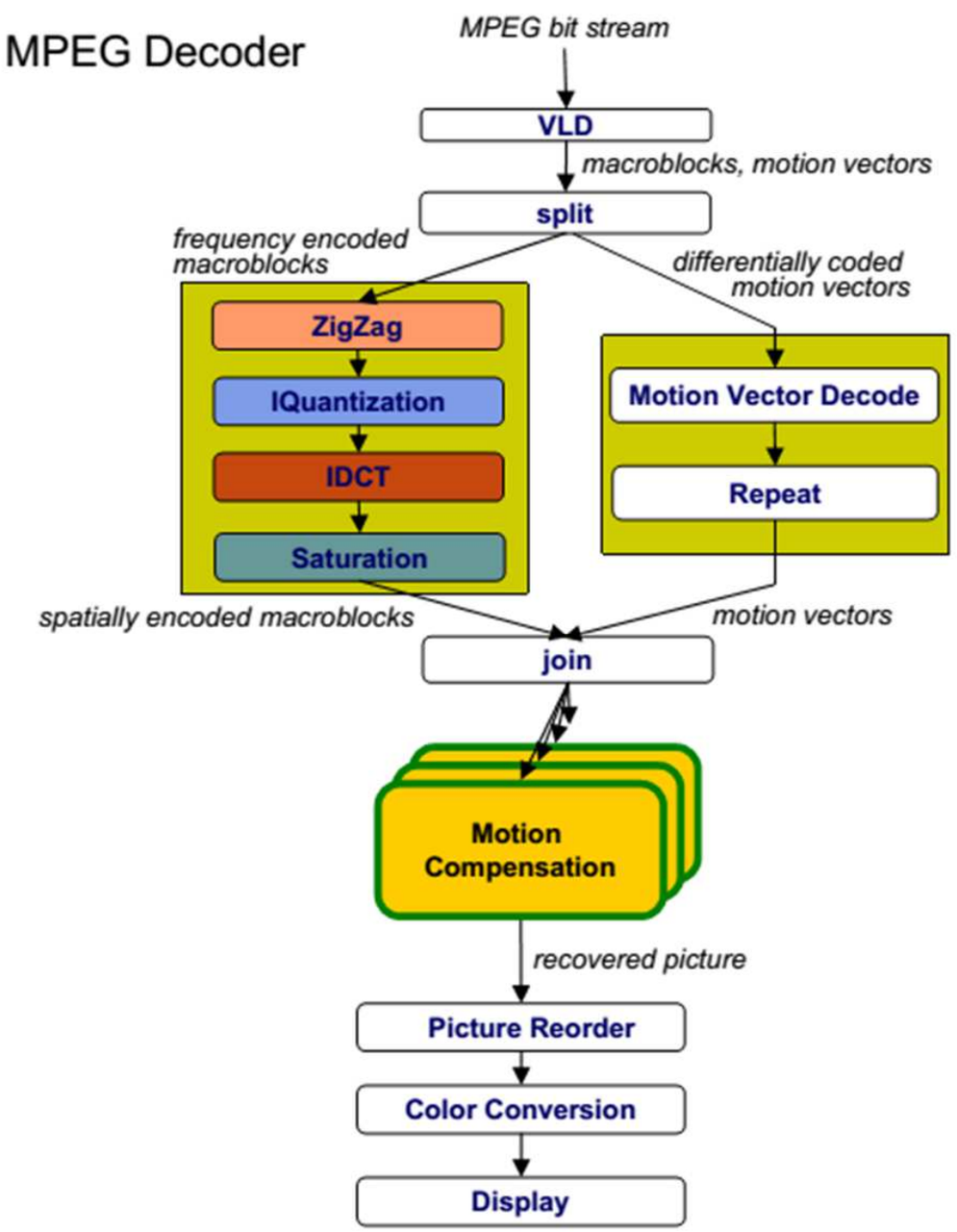

MPEG Decoder
MPEG bit stream
VLD
macroblocks, motion vectors
split
frequency encoded macroblocks
differentially coded motion vectors
ZigZag
IQuantization
IDCT
Saturation
Motion Vector Decode
Repeat
spatially encoded macroblocks
motion vectors
join
Motion Compensation
recovered picture
Picture Reorder
Color Conversion
Display

Example

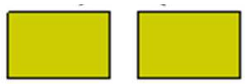


- Task decomposition
  - Independent coarse-grained computation
  - Inherent to algorithm
- Sequence of statements (instructions) that operate together as a group
  - Corresponds to some logical part of program
  - Usually follows from the way programmer thinks about a problem

Example

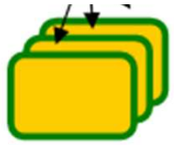


- Task decomposition



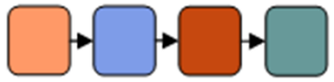
  - Parallelism in the application

- Data decomposition



  - Same computation many data

- Pipeline decomposition



  - Data assembly lines
  - Producer-consumer chains

Credits

- These slides report material from:
  - Prof. Robert van Engelen (Florida State University)
  - Prof. Jan Lemeire (Vrjie Universiteit Brussel)
  - Prof. Allen D. Malony (Univ. of Oregon)
  - Roberto Cavicchioli (Univ. Modena e Reggio Emilia)

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
- Parallel and High Performance Computing, R. Robey, Y. Zamora, Manning - Chapt. 4