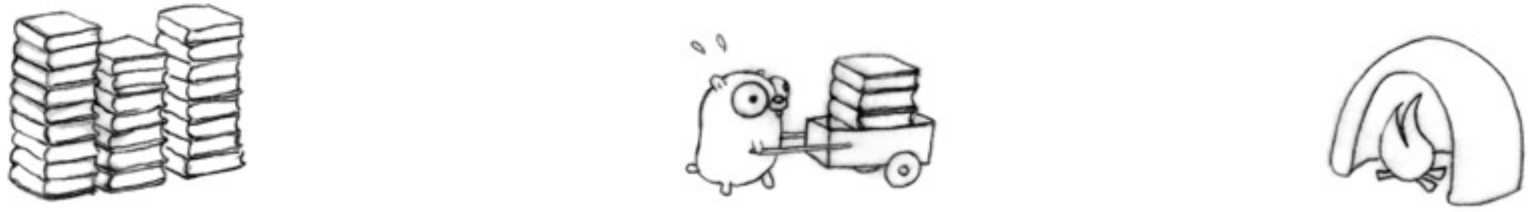
Concurrency & Parallel programming patterns

Evgeny Gavrin

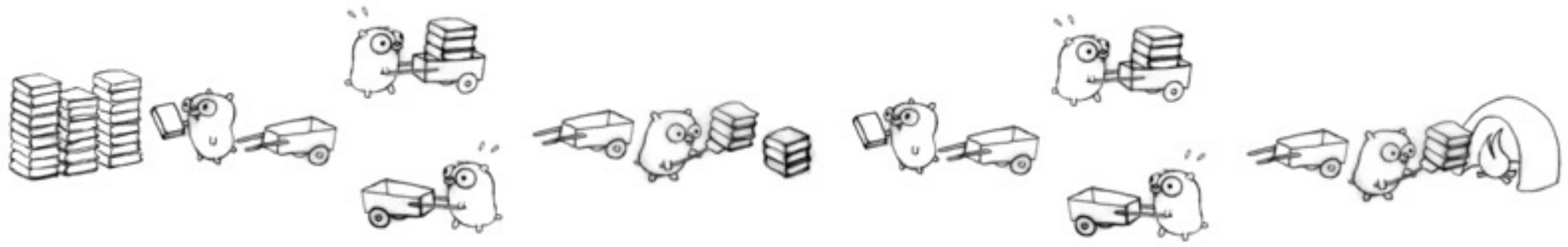
Outline

Concurrency vs Parallelism

- **Parallelism** is the simultaneous execution of computations
  “*doing* lots of things at once”



- **Concurrency** is the composition of independently execution processes
  “*dealing with* lots of thing at once”



Patterns by groups

Architectural Patterns

These patterns define the overall architecture for a program:

- **Pipe-and-filter**: view the program as filters (pipeline stages) connected by pipes (channels). Data flows through the filters to take input and transform into output.
- **Agent and Repository**: a collection of autonomous agents update state managed on their behalf in a central repository.
- **Process control**: the program is structured analogously to a process control pipeline with monitors and actuators moderating feedback loops and a pipeline of processing stages.
- **Event based implicit invocation**: The program is a collection of agents that post events they watch for and issue events for other agents. The architecture enforces a high level abstraction so invocation of an agent is implicit; i.e. not hardwired to a specific controlling agent.
- **Model-view-controller**: An architecture with a central model for the state of the program, a controller that manages the state and one or more agents that export views of the model appropriate to different uses of the model.
- **Bulk Iterative (AKA bulk synchronous)**: A program that proceeds iteratively … update state, check against a termination condition, complete coordination, and proceed to the next iteration.
- **Map reduce**: the program is represented in terms of two classes of functions. One class maps input state (often a collection of files) into an intermediate representation. These results are collected and processed during a reduce phase.
- **Layered systems**: an architecture composed of multiple layers that enforces a separation of concerns wherein (1) only adjacent layers interact and (2) interacting layers are only concerned with the interfaces presented by other layers.
- **Arbitrary static task graph**: the program is represented as a graph that is statically determined meaning that the structure of the graph does not change once the computation is established. This is a broad class of programs in that any arbitrary graph can be used.

Computational Pattern

These patterns describe computations that define the components in a programs architecture.

- **Backtrack, branch and bound**: Used in search problems … where instead of exploring all possible points in the search space, we continuously divide the original problem into smaller subproblems, evaluate characteristics of the subproblems, set up constraints according to the information at hand, and eliminate subproblems that do not satisfy the constraints.
- **Circuits**: used for bit level computations, representing them as Boolean logic or combinational circuits together with state elements such as flip-flops.
- **Dynamic programming**: recursively split a larger problem into subproblems but with memorization to reuse past subsolutions.
- **Dense linear algebra**: represent a problem in terms of dense matrices using standard operations defined in terms of Basic linear algebra (BLAS).
- **Finite state machine**: Used in problems for which the system can be described by a language of strings.
  The problem is to define a piece of software that distinguishes between valid input strings (associated with proper behavior) and invalid input strings (improper behavior).
- **Graph algorithms**: a diverse collection of algorithms that operate on graphs. Solutions involve preparing the best representation of the problem as a graph, and developing a graph traversal that captures the desired computation.
- **Graphical models**: probabilistic reasoning problems where the problem is defined in terms of probability distributions represented as a graphical model.
- **Monte Carlo**: A large class of problems where the computation is replicated over a large space of parameters. In many cases, random sampling is used to avoid exhaustive search strategies.
- **N-body**: Problems in which each member of a system depends on the state of every other particle in the system. The problems typically involve some scheme to approximate the naïve O(N2) exhaustive sum.
- **Sparse Linear Algebra**: Problems represented in terms of sparse matrices. Solutions may be iterative or direct.
- **Spectral methods**: Problems for which the solution is easier to compute once the domain has been transformed into a different representation. Examples include Z-transform, FFT, DCT, etc. The transform itself is included in this class of problems.
- **Structured mesh**: Problem domains are mapped onto a regular mesh and solutions computed as averages over neighborhoods of points (explicit methods) or as solutions to linear systems of equations (implicit methods)
- **Unstructured mesh**: The same as the structured mesh problems, but the mesh lacks structure and hence, the computations involved scatter and gather operations.

Algorithm Patterns

These patterns describe parallel algorithms used to implement the computational patterns.

- **Task parallelism**: Parallelism is expressed as a collection of explicitly defined tasks. This pattern includes the embarrassingly parallel pattern (no dependencies) and separable dependency pattern (replicated data/reduction).
- **Data parallelism**: Parallelism is expressed as a single stream of tasks applied to each element of a data structure. This is generalized as an index space with the stream of tasks applied to each point in the index space.
- **Recursive splitting**: A problem is recursively split into smaller problems until the problem is small enough to solve directly. This includes the divide and conquer pattern as a subset wherein the final result is produce by reversing the splitting process to assemble solutions to the leaf-node problems into the final global result.
- **Pipeline**: Fixed coarse grained tasks with data flowing between them.
- **Geometric decomposition**: A problem is expressed in terms of a domain that is decomposed spatially into smaller chunks. Solution is composed of updates across chunk boundaries, updates of local chunks, and then updates to the boundaries of the chunks.
- **Discrete event**: a collection of tasks that coordinate among themselves through discrete events. This pattern is often used for GUI design and discrete event simulations.
- **Graph partitioning**: Tasks generated by decomposing recursive data structures (graphs)

Software structure pattern

Program structure

- **SPMD**: One program used by all the threads or processes, but based on ID different paths or different segments of data are executed.
- **Strict data parallel**: A single instruction stream is applied to multiple data elements.
  This includes vector processing as a subset.
- **Loop level parallelism**: Parallelism is expressed in terms of loop iterations that are mapped onto multiple threads or processes.
- **Fork/join**: Threads are logically created (forked), used to carry out a computation, and then terminated (joined).
- **Master-worker/Task-queue**: A master sets up a collection or work-items (tasks), a collection of workers pull work-items from the master (a task-queue), carry out the computation, and then go back to the master for more work.
- **Actors**: a collection of active software agents (the actors) interact over distinct channels.
- **BSP**: The Bulk Synchronous model from Leslie Valiant.

Data Structure Patterns

- **Shared queue**: this pattern describes ways to any of the common queue data structures and manage them in parallel
- **Distributed array**: An array data type that is distributed about a threads or processes involved with a parallel computation.
- **Shared hash table**: A hash table shared/distributed among a set of threads or processes with any concurrency issues hidden behind an API.
- **Shared data**: a “catch all” pattern for cases where data is shared within a shared memory region but the data can not be represented in terms of a well defined and common high level data structure.

Execution Patterns

Process/thread control patterns:

- **CSP or Communicating Sequential Processes**: Sequential processes execute independently and coordinate their execution through discrete communication events.
- **Data flow**: sequential processes organized into a static network with data flowing between them.
- **Task-graph**: A directed acyclic graph of threads or processes is defined in software and mapped onto the elements of a parallel computer.
- **SIMD**: A single stream of program instructions execute in parallel for different lanes in a data structure. There is only one program counter for a SIMD program. This pattern includes vector computations.
- **Thread pool**: The system maintains a pool of threads that are utilized dynamically to satisfy the computational needs of a program. The pool of threads work on queues of tasks. Work stealing is often used to enforce a more balanced load.
- **Speculation**: a thread or process is launched to pursue a computation, but any update to the global state is held in reserve to be entered once the computation is verified as valid.

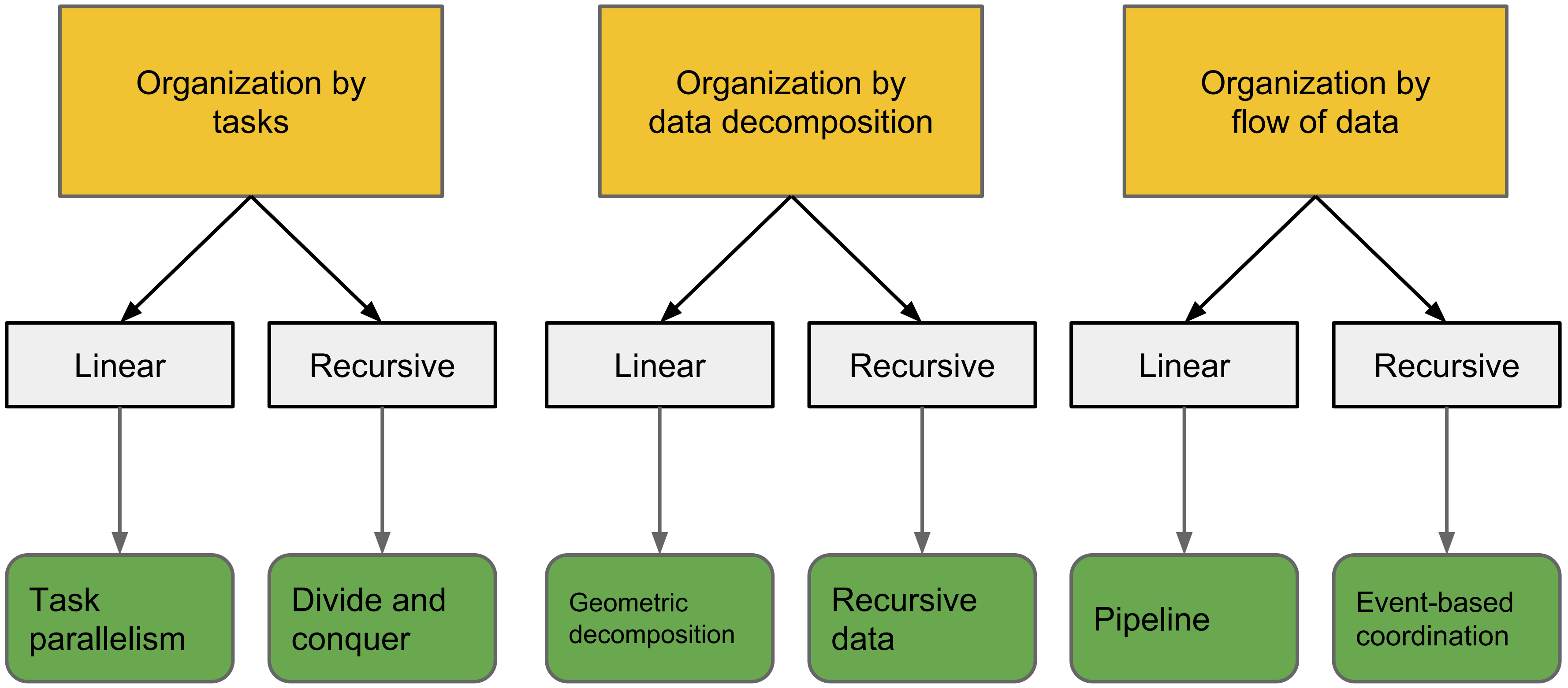
Coordination Patterns:

- **Message passing**: two sided and one sided message passing
- **Collective communication**: reductions, broadcasts, prefix sums, scatter/gather etc.
- **Mutual exclusion**: mutex and locks
- **Point to point synchronization**: condition variables, semaphores
- **Collective synchronization**: e.g. barriers
- **Transactional memory**: transactions with roll-back to handle conflicts.

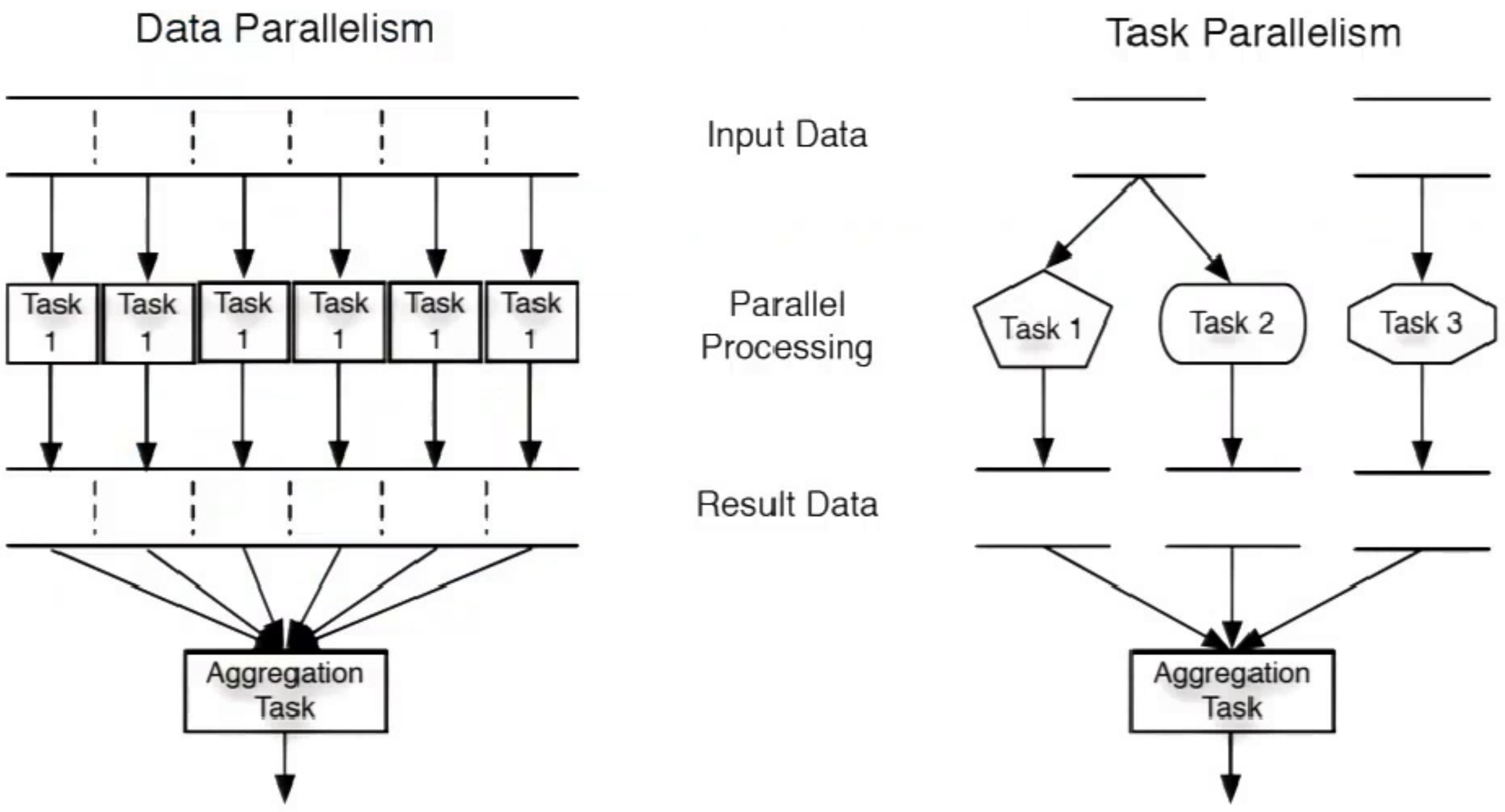
Detailed overview of parallel patterns

*only the most interesting ones

Pattern by algorithm structure



Task parallelism vs Data parallelism



SPMD + SIMD

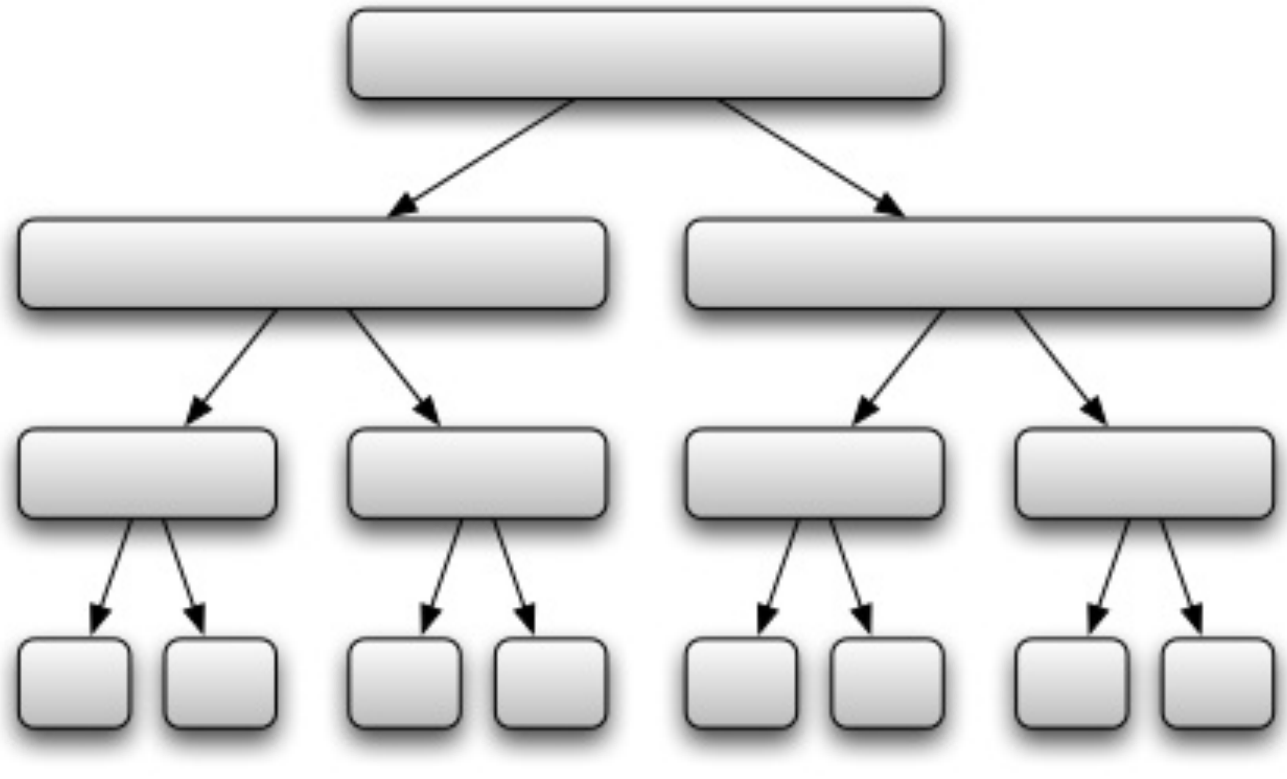
- Single program, multiple data
- Tasks are split up and run simultaneously on multiple processors, each task has its own set of data.
  - Initialize
  - Obtain a unique identifier
  - Run the same program each processor
  - Distributed data
  - Finalize

SPMD usually refers to message passing programming on distributed memory computer architectures. A distributed memory computer consists of a collection of independent computers, called nodes.

SIMD imposes lockstep on different data

SPMD is the most common style of parallel programming

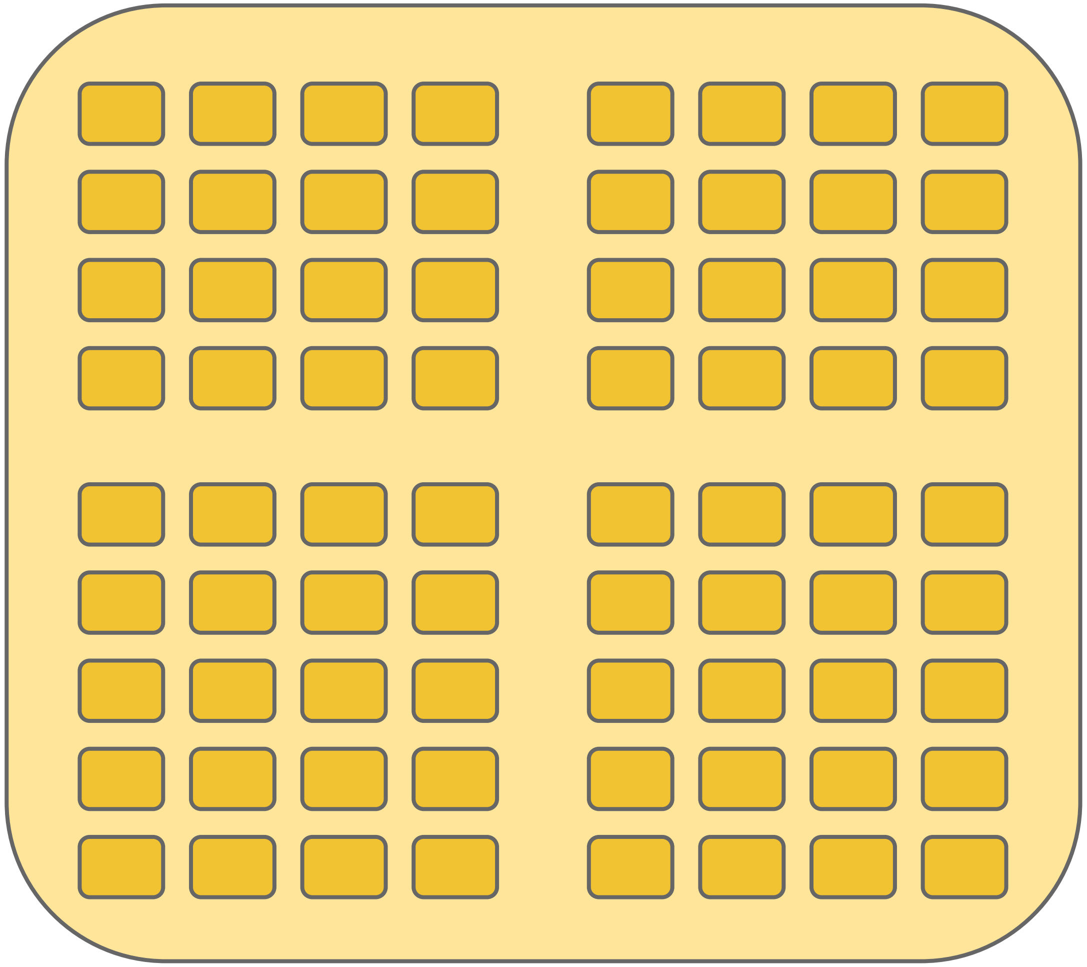
Recursive splitting (data + task)



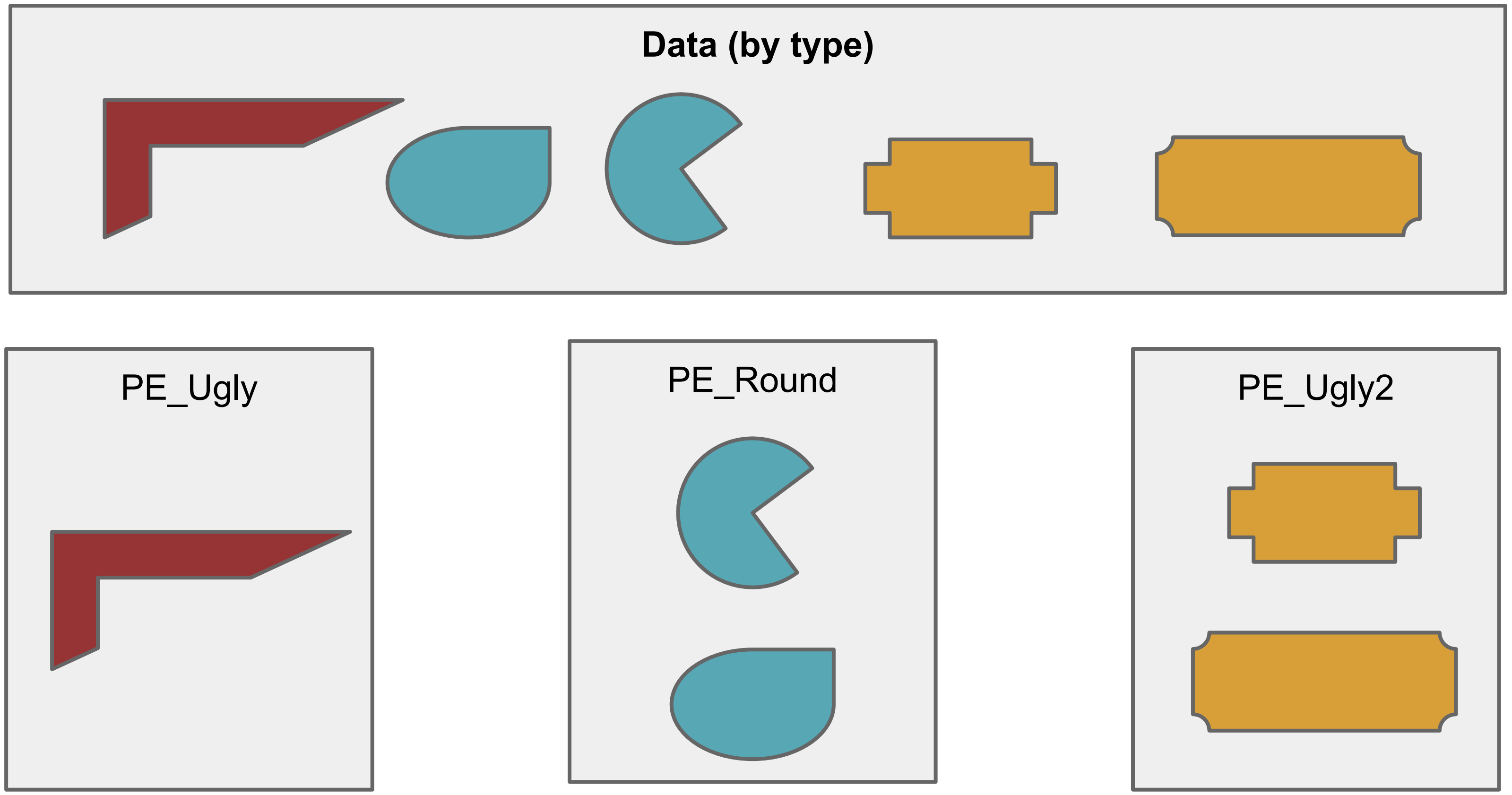
Create small units of work to split over available processor capacity

Geometric decomposition 1

- Geometric decomposition breaks an input collection into sub-collection
- Partition is a special where sub-collections do not overlap
- Does not move data, it just provides an alternative “view” on its organization



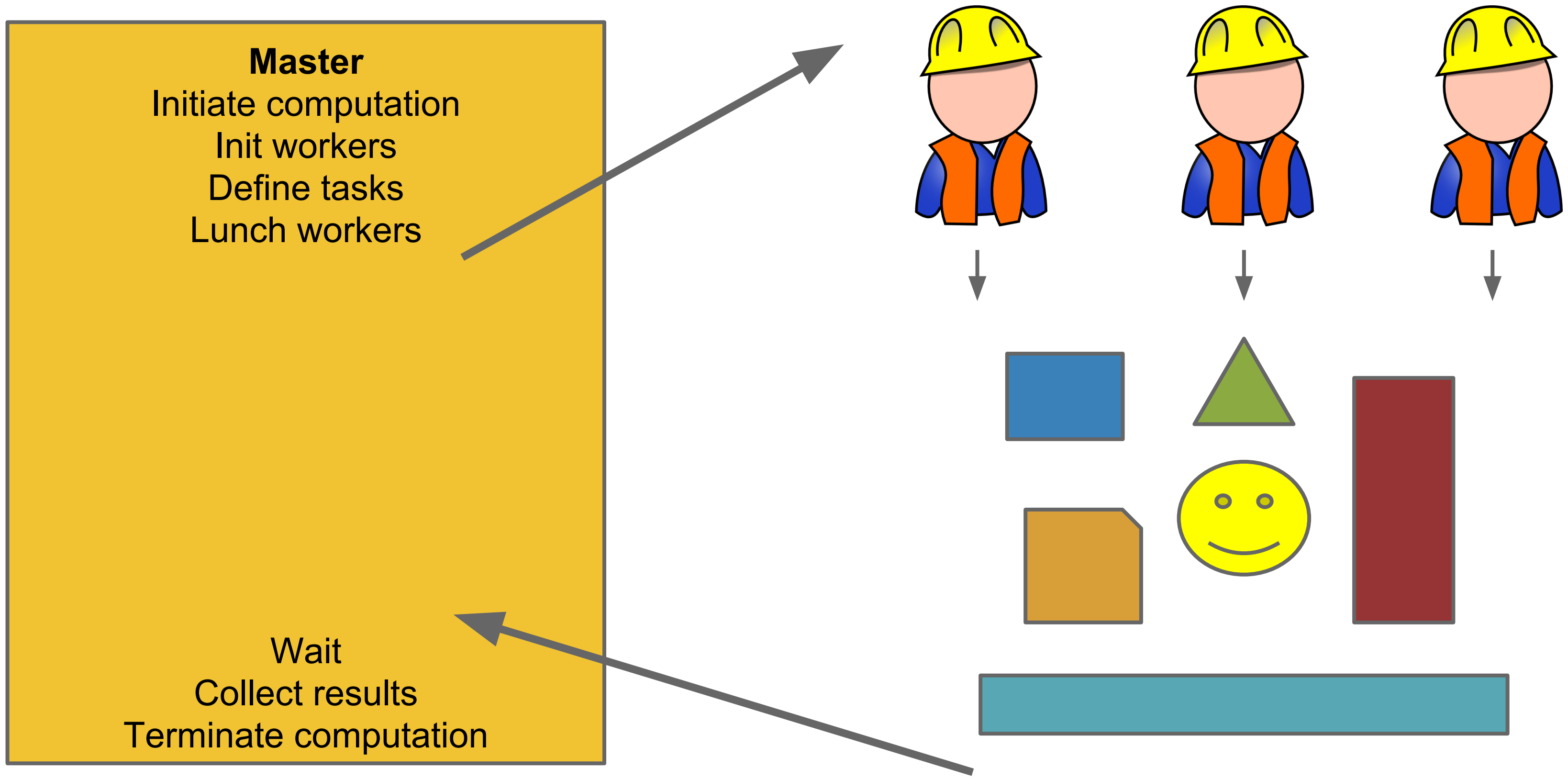
Geometric decomposition 2



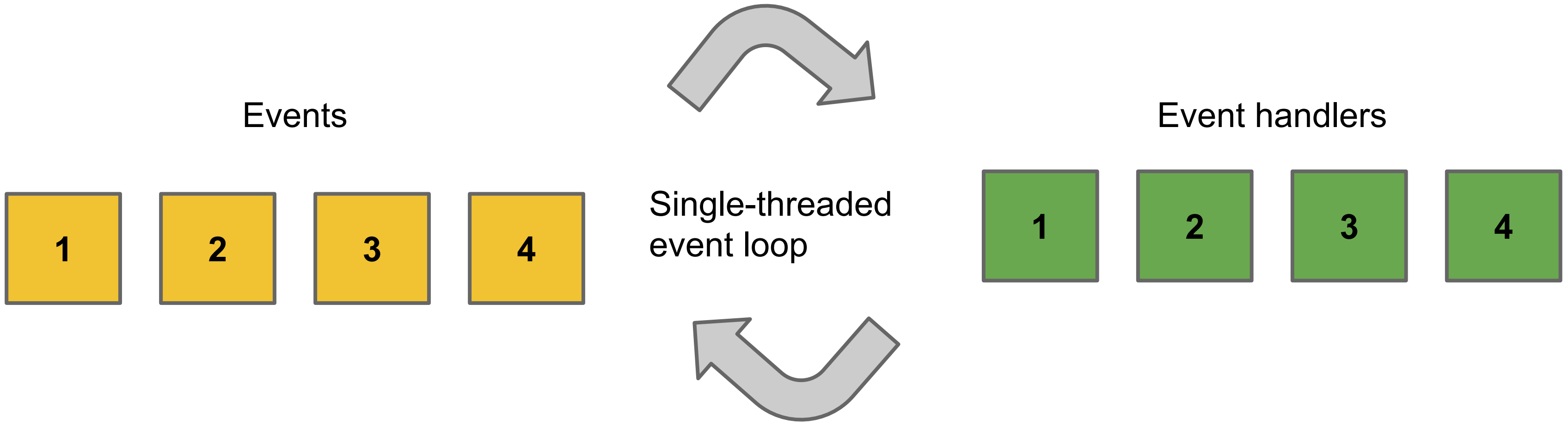
Split big amount of data on groups by types for further processing

Master-worker/Thread pool

- Workers execute concurrently, with each worker repeatedly removing a task from pool of the tasks
- Embarrassingly parallel

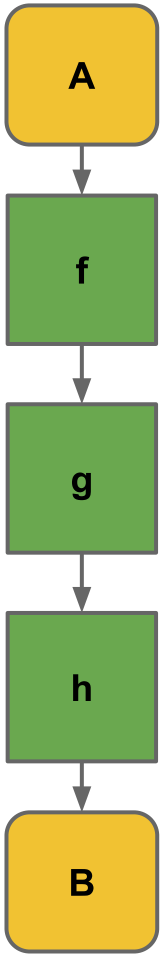


Discrete event (event-based)



Event-driven programming is widely used in GUI, for instance the Android concurrency frameworks are designed using the Half-Sync/Half-Async pattern, where a combination of a single-threaded event loop processing (for the main UI thread) and synchronous threading (for background threads) is used.

Sequence (Serial, not Parallel)



A serial sequence is executed in the exact order given:

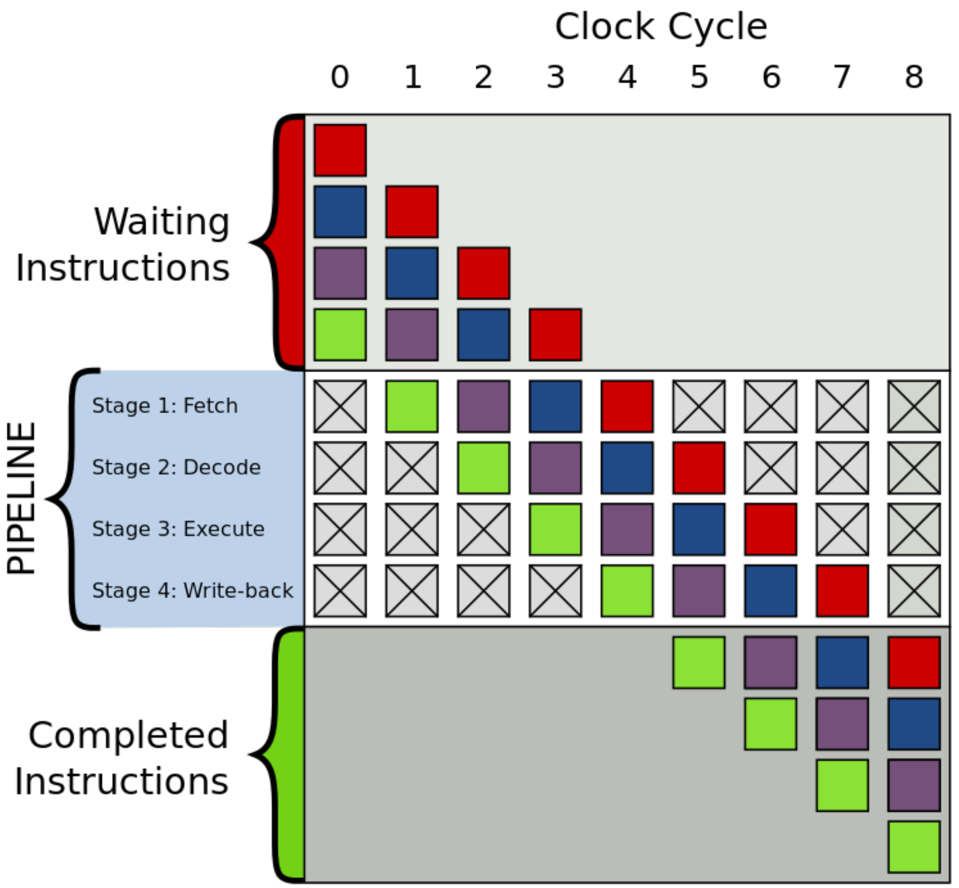
```
F = f(A);
G = g(F);
B = h(G);
```

Pipeline

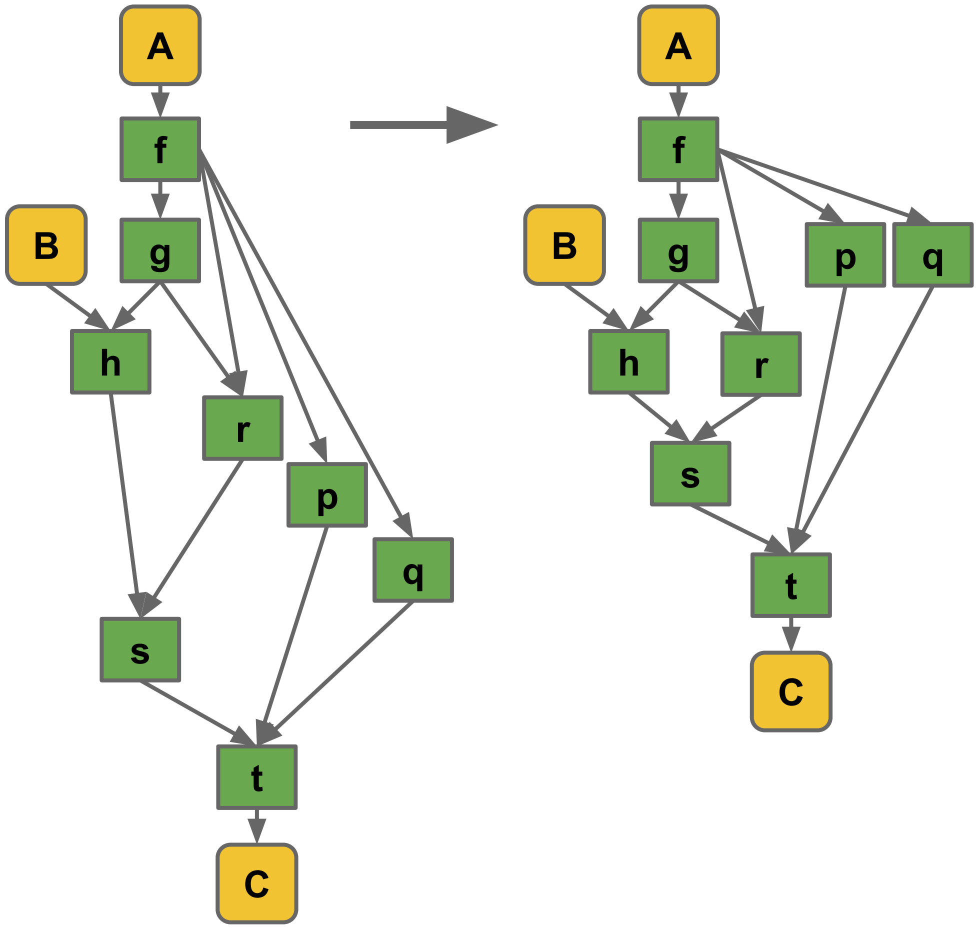
- **Pipeline** uses a sequence of stages that transform a flow of data
- Some stage may retain state

Examples:

- Instruction pipeline in modern CPUs
- Algorithm level pipelining
- Signal processing
- cat main.c | grep ‘todo’ | wc



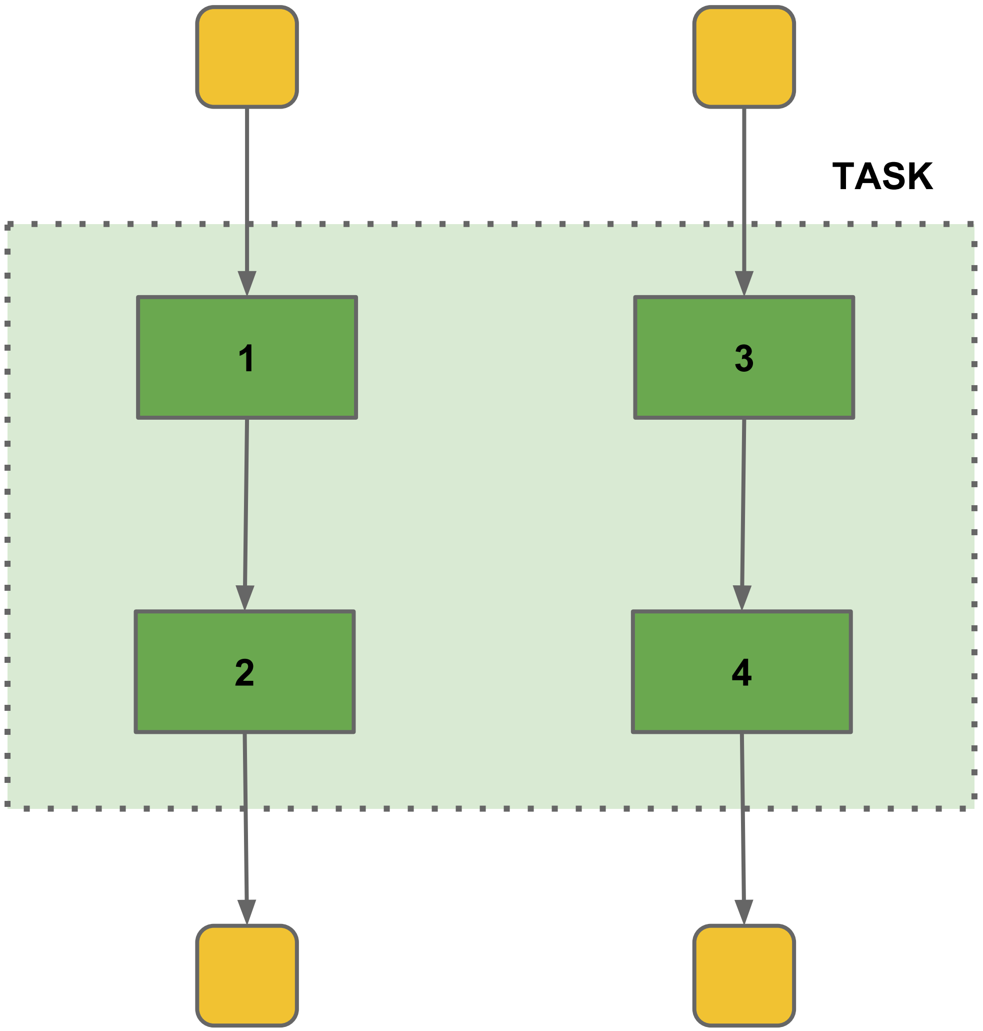
Superscalar Sequence (Data flow)



F = f(A);
G = g(F);
H = h(B,G);
R = r(G);
P = p(F);
Q = q(F);
S = s(H,R);
C = t(S,P,Q);

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

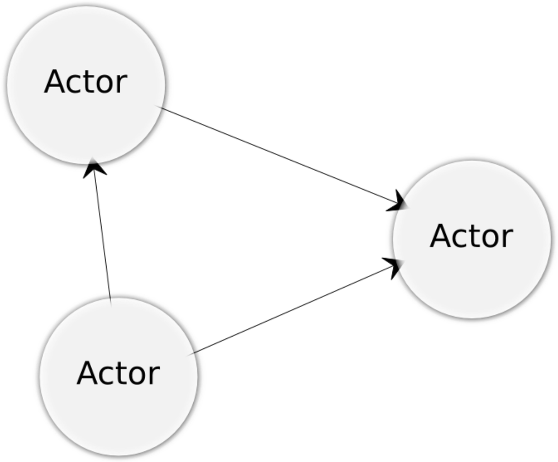
Graph composition (decomposition)

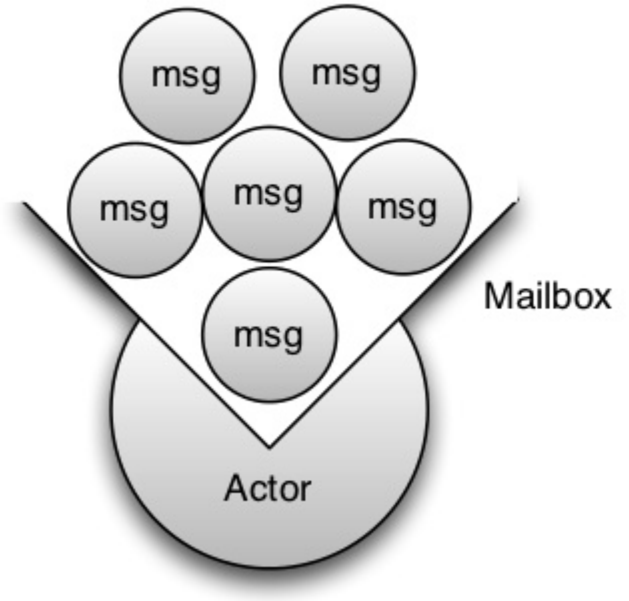


Task-graph (Actors)

An **actor** is a computational entity that, in response to a message it receives, can concurrently:

- send a finite number of messages to other actors;
- create a finite number of new actors;
- designate the behavior to be used for the next message it receives.
- Actor may have several inputs
  - specified messages
- May have input buffer
  - Mailbox
- May have conditions:
  - pre start
  - pre/post restart
  - post stop

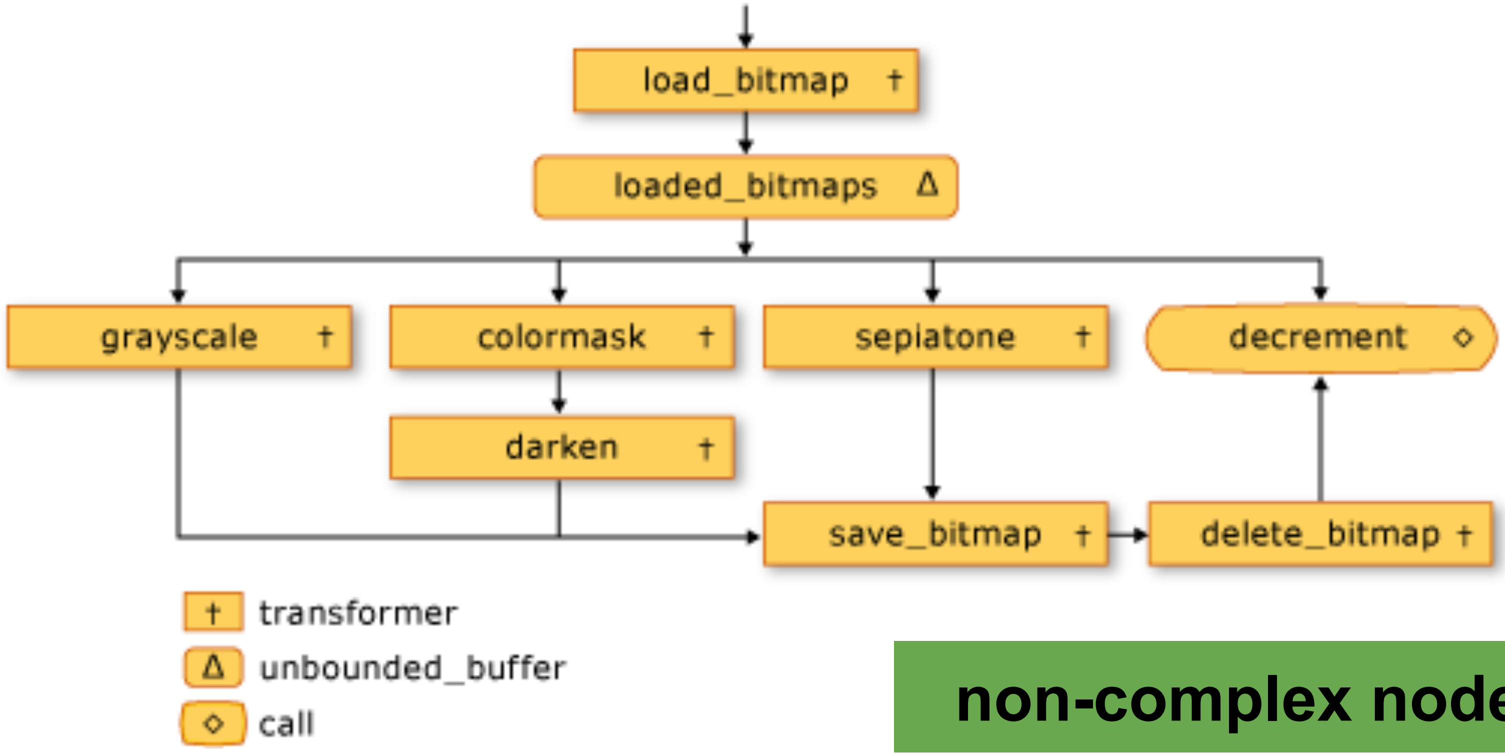




complex nodes

Task-graph (Agents)

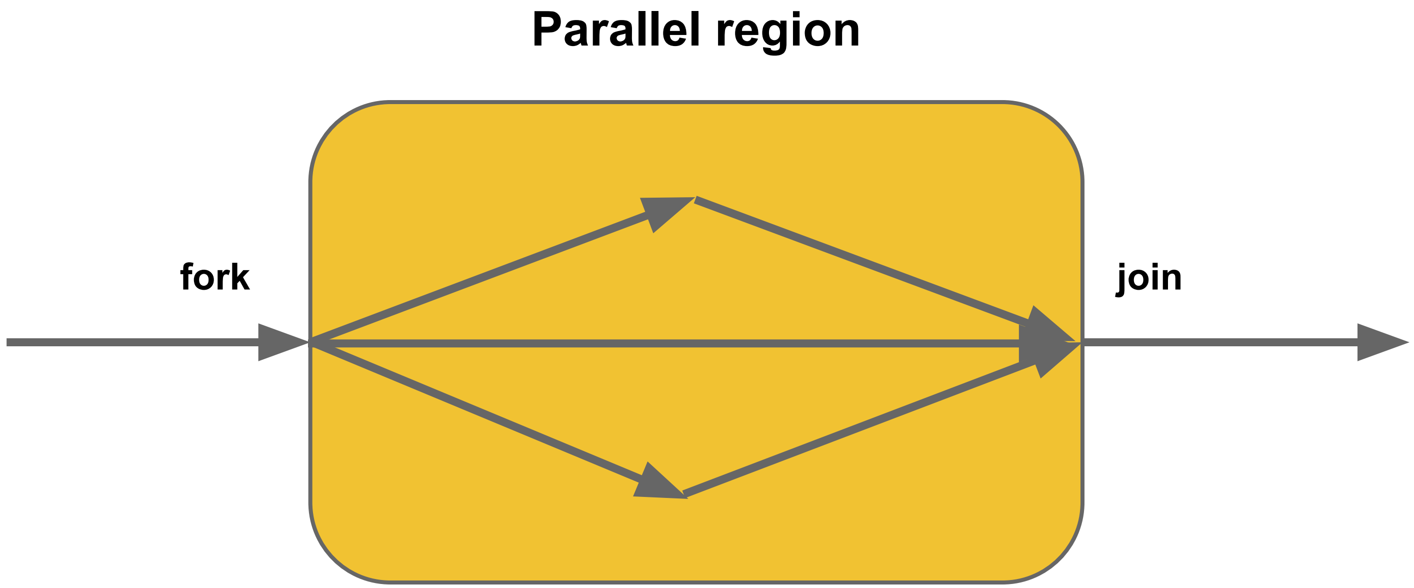
- Same as actors, but has number of limitations
- Cannot create tasks
- One input / One output (Tuple)
- No communication buffer



non-complex nodes

Fork/Join

- Parent tasks creates new task (fork) then waits until all they complete (join) before continuing on with the computation
- Fork/Join can be nested (to implement parallel for)



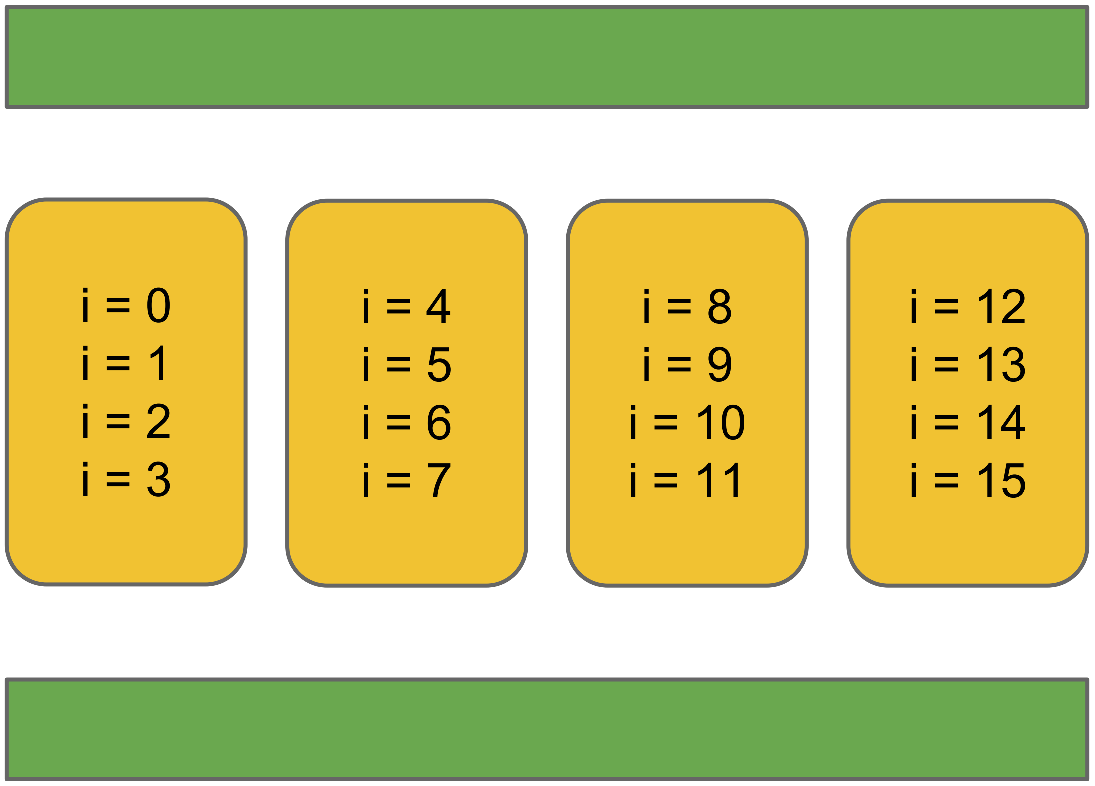
Loop-based parallelism

The iteration pattern repeats some section of code as long as condition holds

```
while ( C ) {
    f();
}
```

Each iteration **can** depend on values computed in any earlies iteration

Loop can be terminated at any point based on computations in any iteration



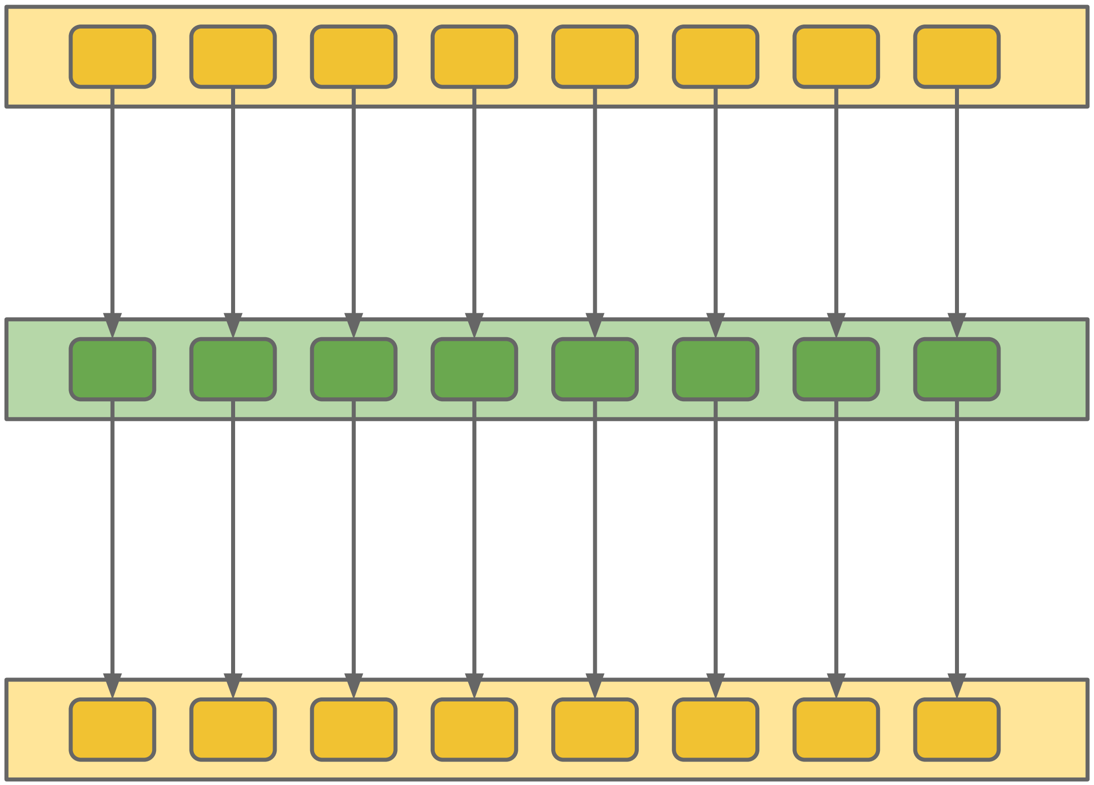
for (i = 0; i < 16; i++)

C[i] = A[i]+B[i];

Loop-based parallelism (map/foreach)

- **Map** replicates a function over every element of an index set
- The index set may be abstract or associated with the elements of an array

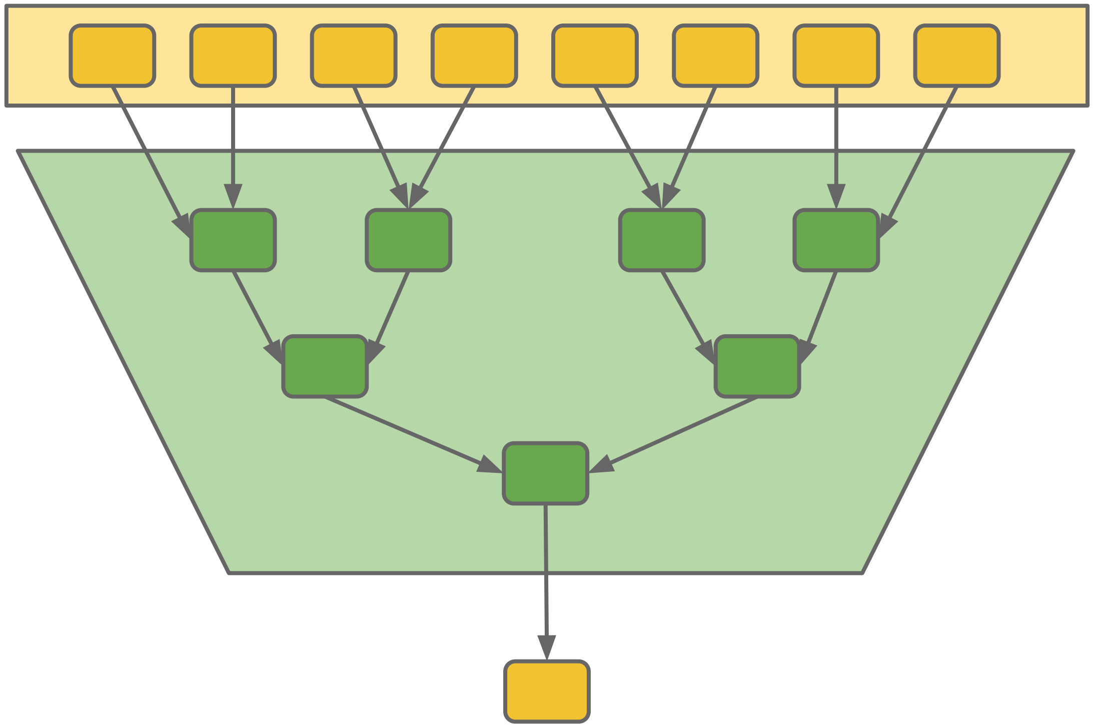
```
for (i=0; i<n; ++i) {
    f(A[i]);
}
```



Loop-based parallelism (reduction)

- **Reduction** combines every element in a collection into one element using an associative operator

```
b = 0;
for (i=0; i<n; ++i) {
    b += f(B[i]);
}
```



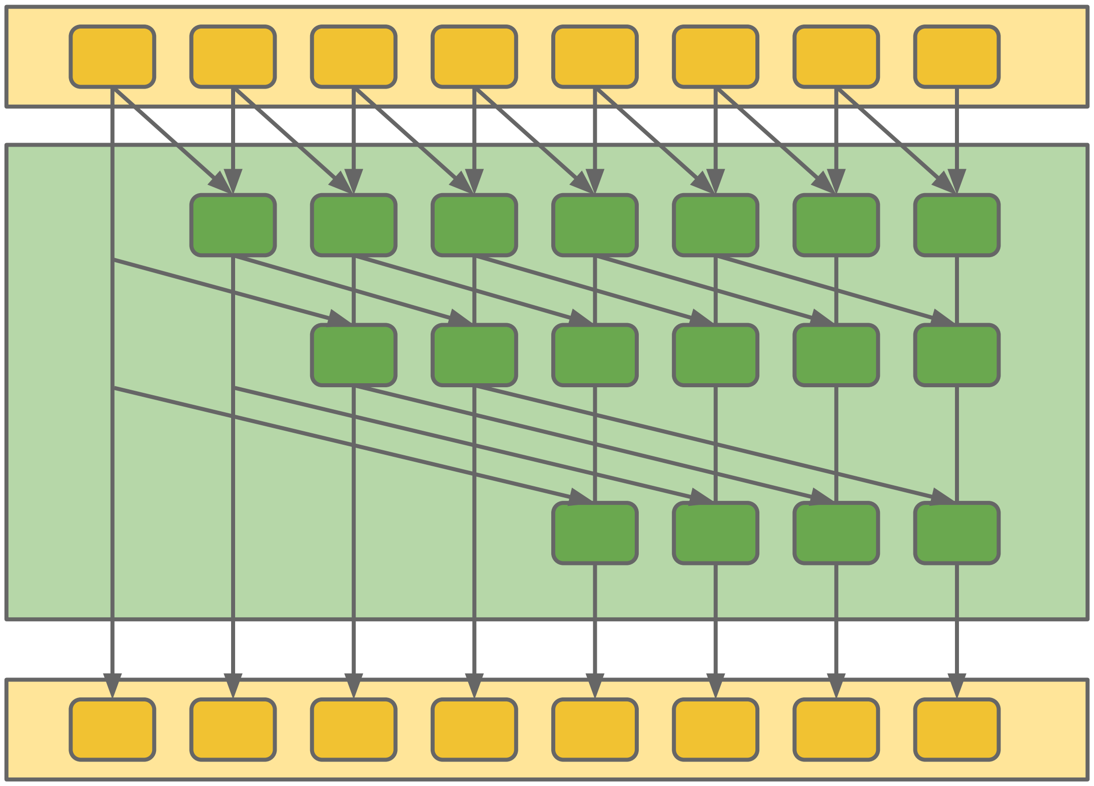
Loop-based parallelism (scan)

- **Scan** computes all partial reductions of a collection

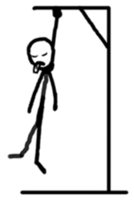
```
A[0] = B[0];
for (i=0; i<n; ++i) {
    A[i] = B[i] + A[i-1];
}
```

- Operator must be at least associative

Example: Random number generation

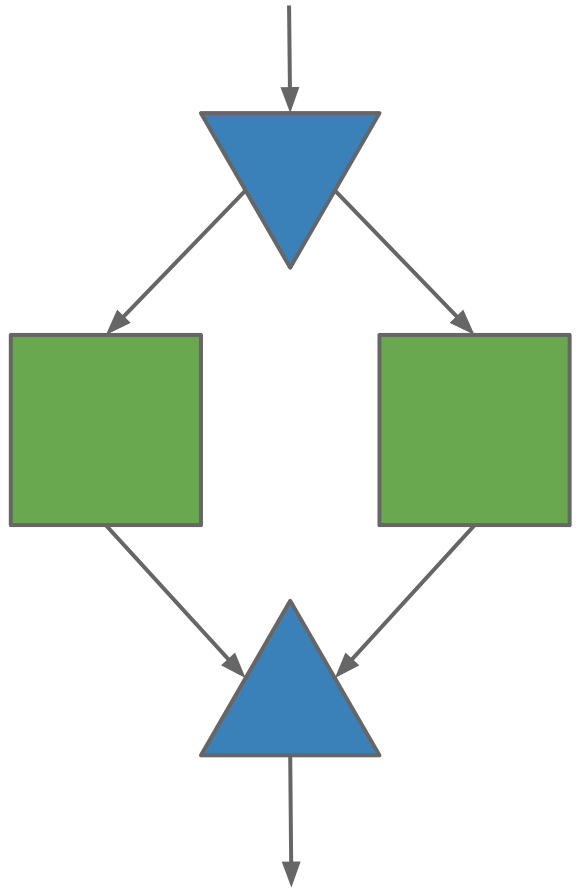


*one possible implementation

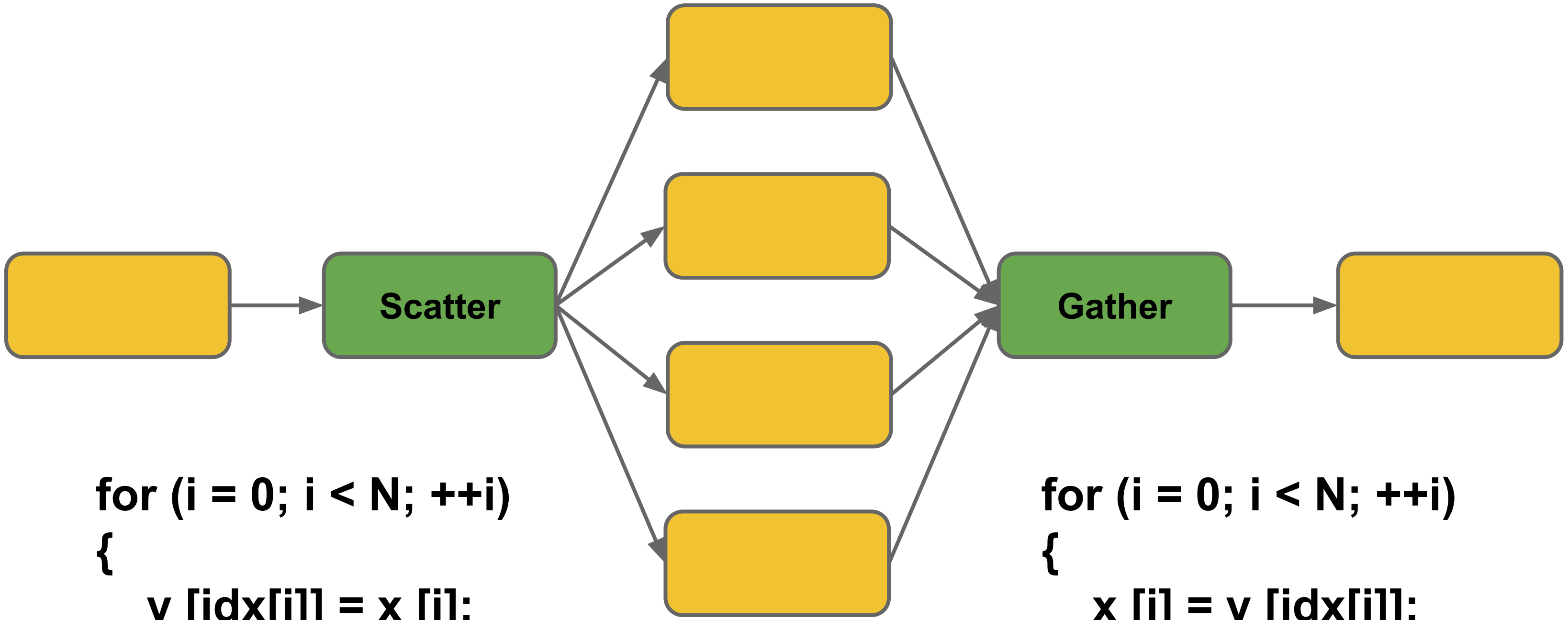


Speculative selection

- **Selection** chooses the results from one of two alternative tasks based on a condition.
- Both alternative tasks are executed in parallel while the condition is being evaluated.
- This pattern is most useful for complex conditions
- Once the condition is evaluated one of the two alternative tasks needs to be cancelled



Scatter/Gather

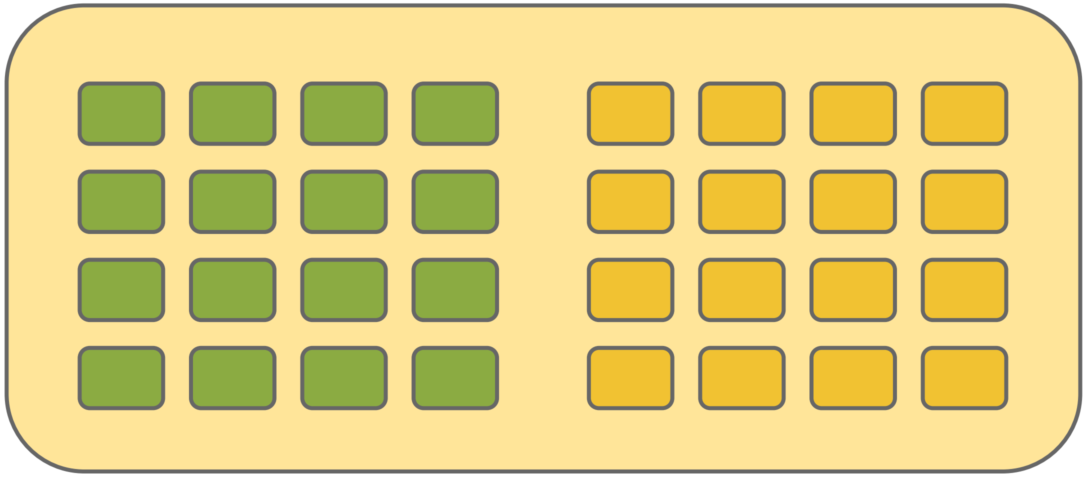


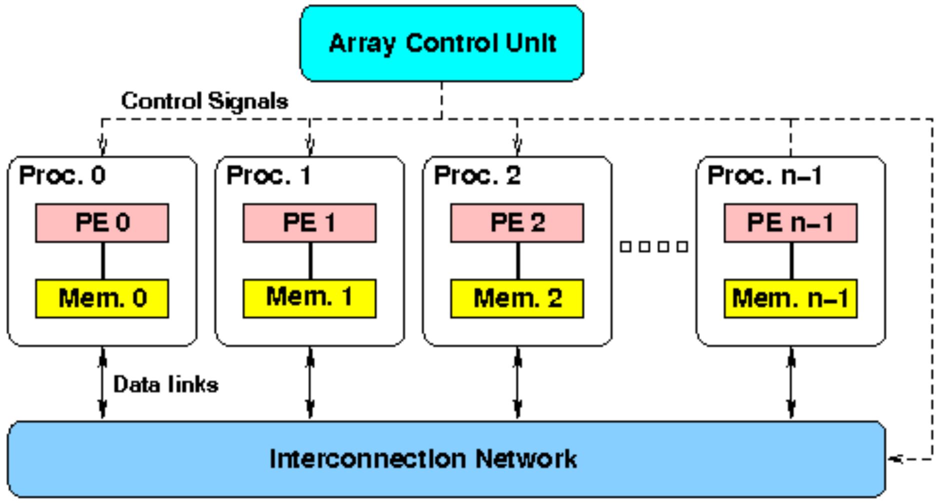
```
for (i = 0; i < N; ++i)
{
    y [idx[i]] = x [i];
}
```

```
for (i = 0; i < N; ++i)
{
    x [i] = y [idx[i]];
}
```

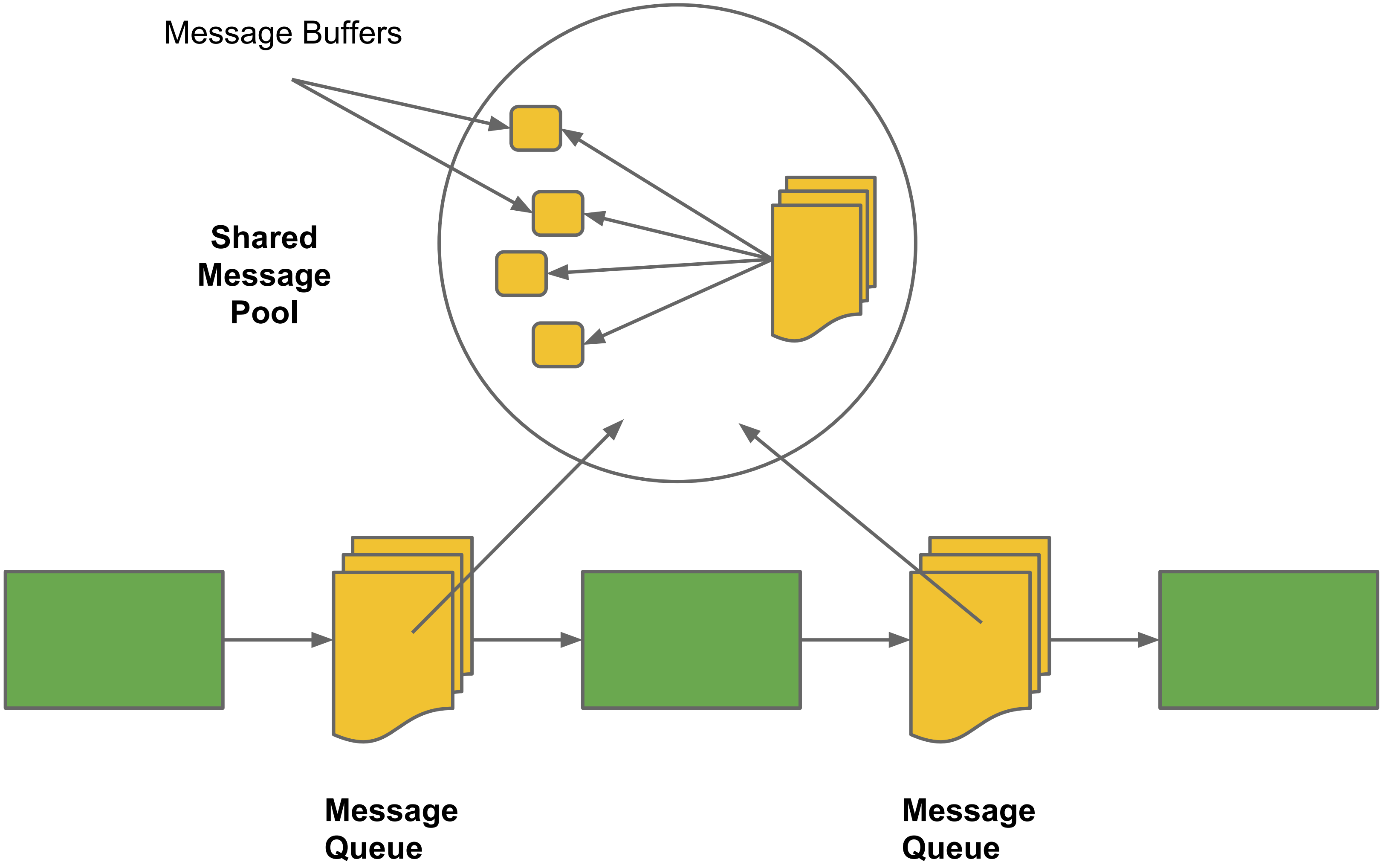
Distributed array

- **Distributed array** has physically distributed data
- But can be access through a shared data-like style





Shared Queue



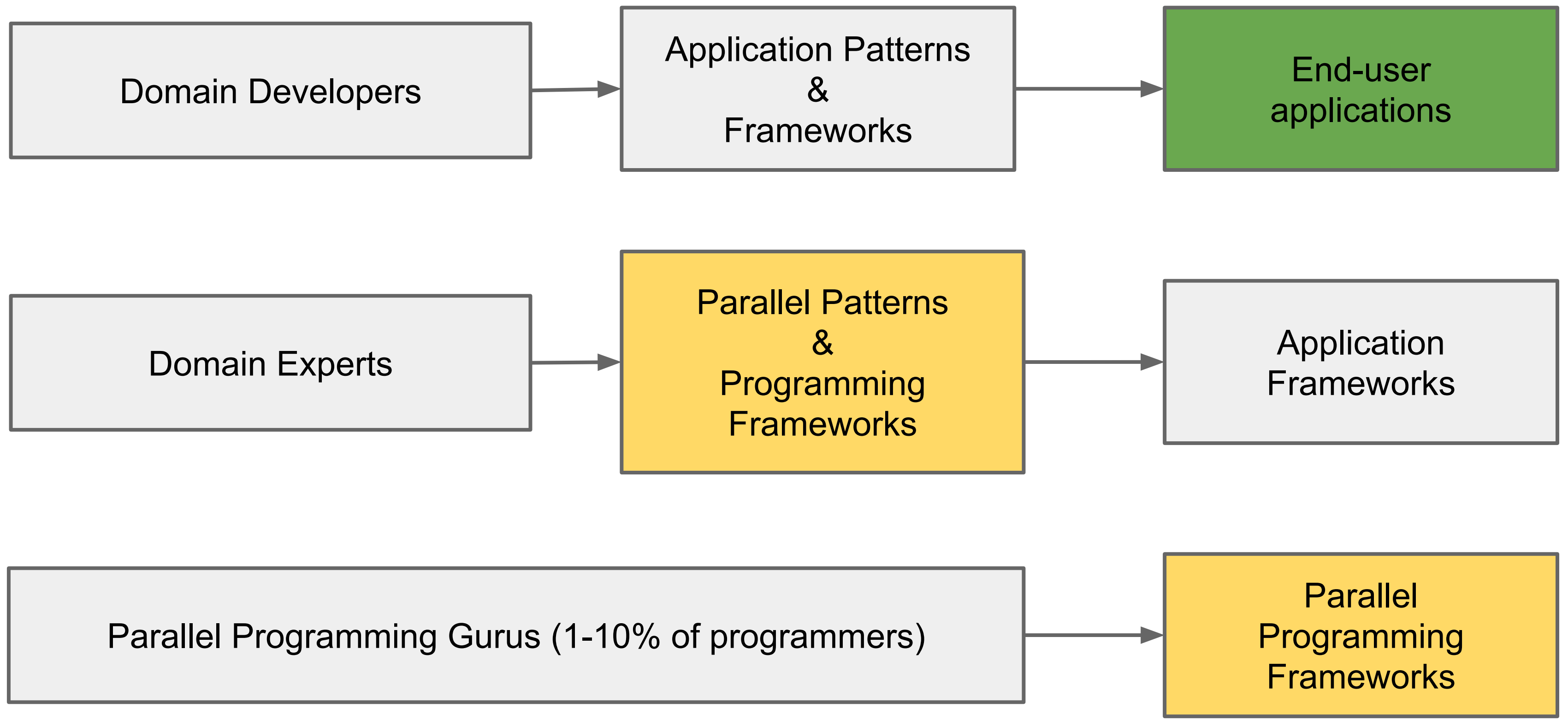
Summary

Conclusion

- All these patterns looks great!
  But some are useful anyway...

- People have been creating parallel programming languages and frameworks for many years…

Our goal: Use patterns to create frameworks


Domain Developers
Application Patterns & Frameworks
End-user applications
Domain Experts
Parallel Patterns & Programming Frameworks
Application Frameworks
Parallel Programming Gurus (1-10% of programmers)
Parallel Programming Frameworks

Domain Developers should be able to create parallel applications with little or no understanding of parallel programming

Programmability evaluation

- Define set of programmability benchmarks
  - Must cover the major classes of application and parallel algorithms
- Programmability benchmarks must be:
  - Provided as serial code in C
  - Contain lots of concurrency
  - Produce ‘right’ result that can be easily verified
  - Short

- Maybe we could use an “interesting” subset of The thirteen dwarves:
- Dense Linear Alg.
- Sparse Lin. Alg.
- Spectral methods
- N-body methods
- Structured grids
- Unstruc. grids
- MapReduce
- Combinatorial logic
- Graph traversal
- Dynamic prog
- Back-track/branch and bound
- Graphical methods
- Finite state mach.

Useful patterns

- Event-based for UI only
- Loop-based
  - ordered sequential loop (for)
  - parallel loop
- Parallel-friendly collections and methods
  - Distributed Array
  - Loop-based patterns applied to collections
    - map/reduce/scan/recurrence/scatter/gather
- Task pool for user tasks
  - And maybe for ourselves
  - See next slide…

Using these patterns, threads and vector intrinsics can (mostly) be eliminated and the maintainability of software improved

Nice Task Pool

- Difficult to tune properly to obtain the best performance
- Prohibit any concurrency outside
  the Task Pool

- Task = actor
- Intercommunication through mailboxes
- Actor may spawn other actors
  - parallel loop is also an actor
- Joint execution runtime for all applications