Parallel Design Patterns-L09

Implementation Strategy,

SPMD,

Master/Worker

Course Organiser: Dr Nick Brown
nick.brown@ed.ac.uk
Bayes room 2.13

From Algorithm Strategy to Implementation Strategy

Finding Concurrency
- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, …

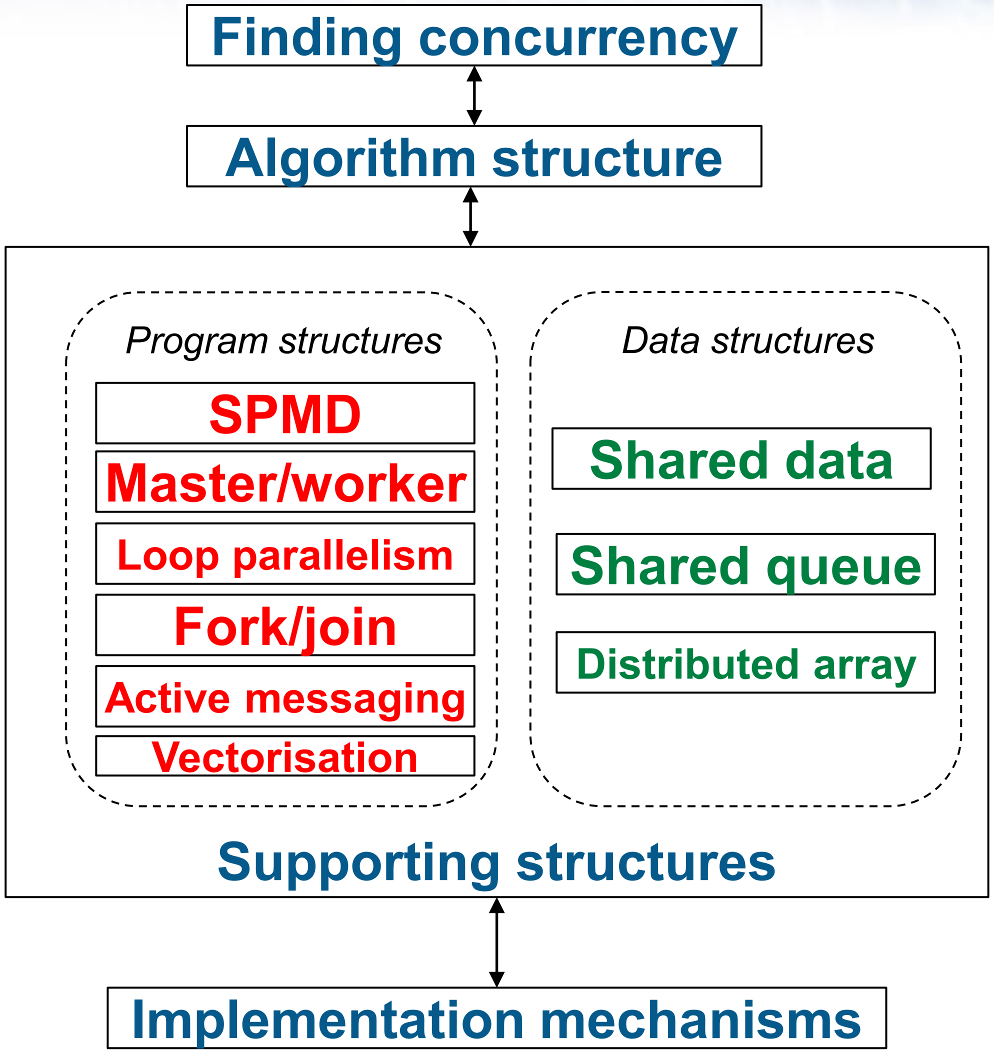
Algorithm Structure
- Tasks Parallelism, Divide and Conquer, Geometric Decomposition, Recursive Data, …

Supporting Structures
- SPMD, Master/Worker, Loop Parallelism, Fork/Join, …

Implementation Mechanisms
- UE Management, Synchronisation, Communication, …

Supporting structures is also known as implementation strategy and we will use these terms interchangeably

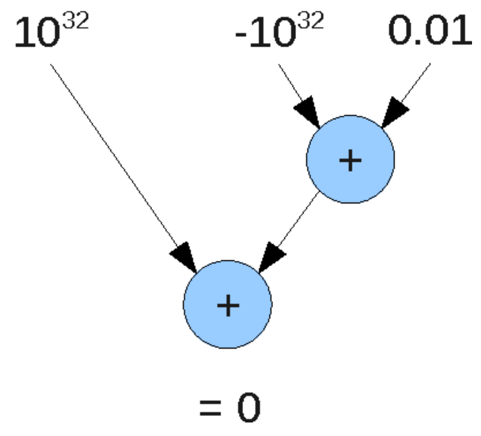
Overview of supporting structures



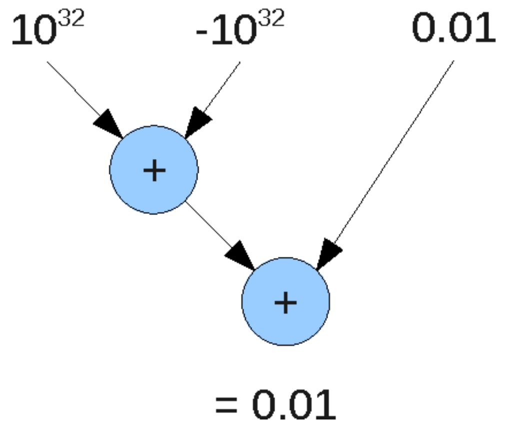
Idioms (or approaches) used by experienced parallel programmers for structuring their software

Supporting structures: Forces

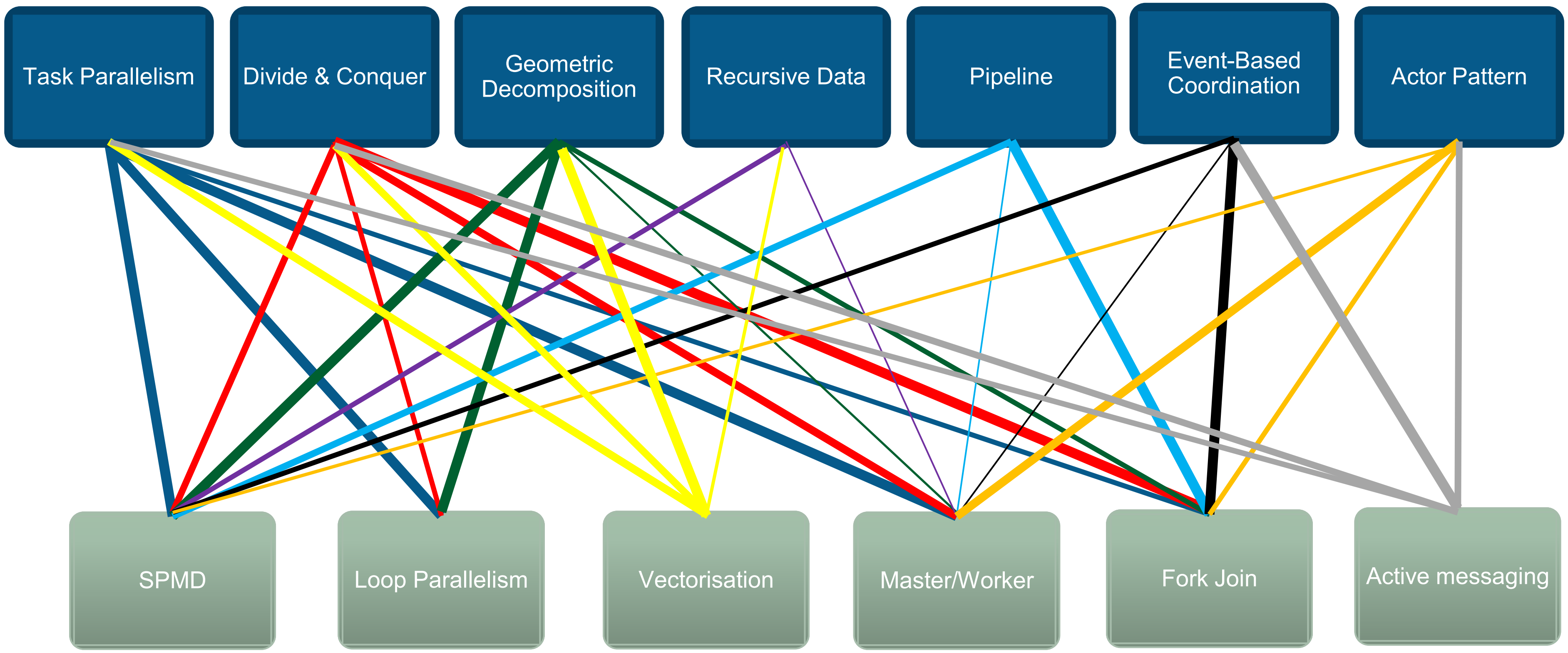
- All address the same basic problem
  - "How do we structure the software to best support algorithm structures of interest?"



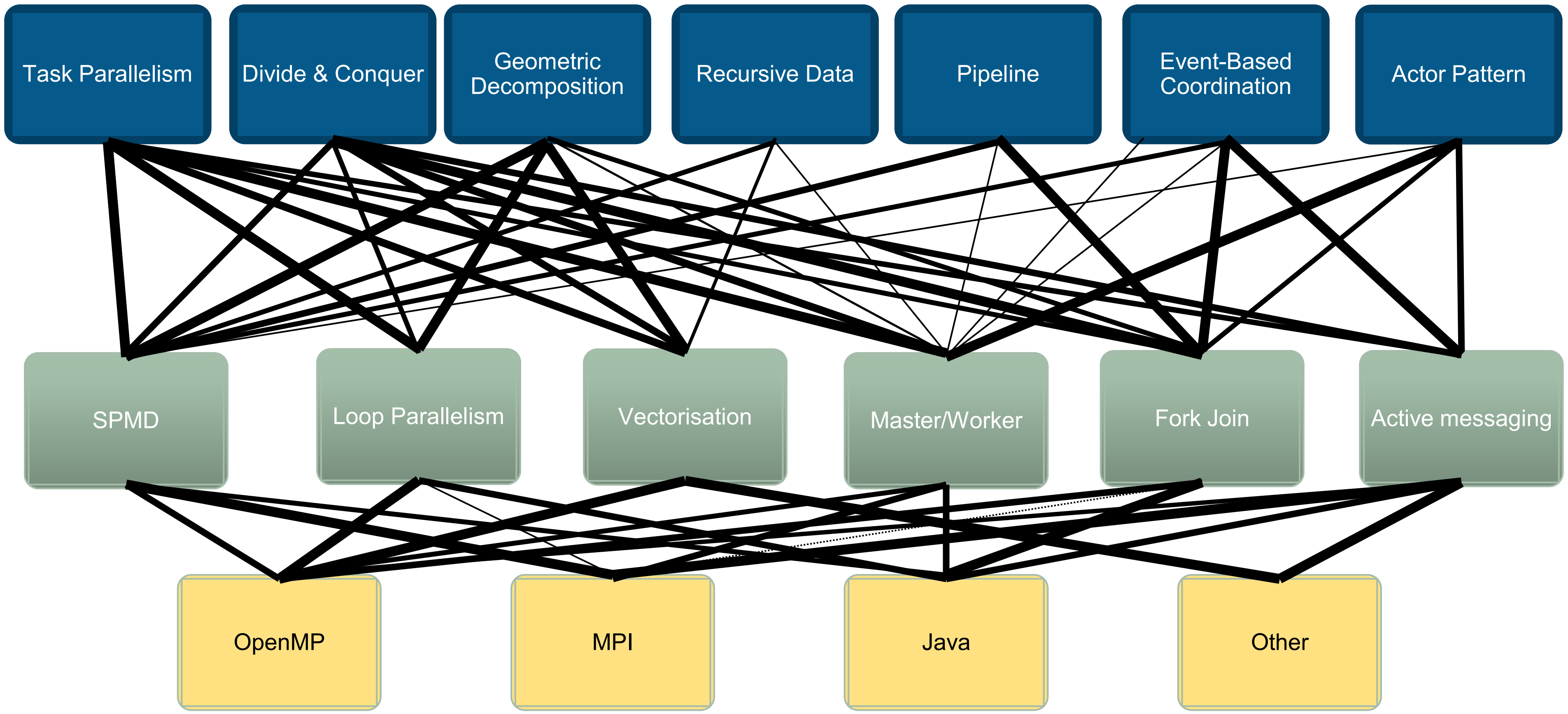
- Clarity of abstraction
- Scalability
- Efficiency
- Maintainability
- Environmental affinity
- Sequential equivalence
  - ➢ $10^{32} + -10^{32} + 0.01$



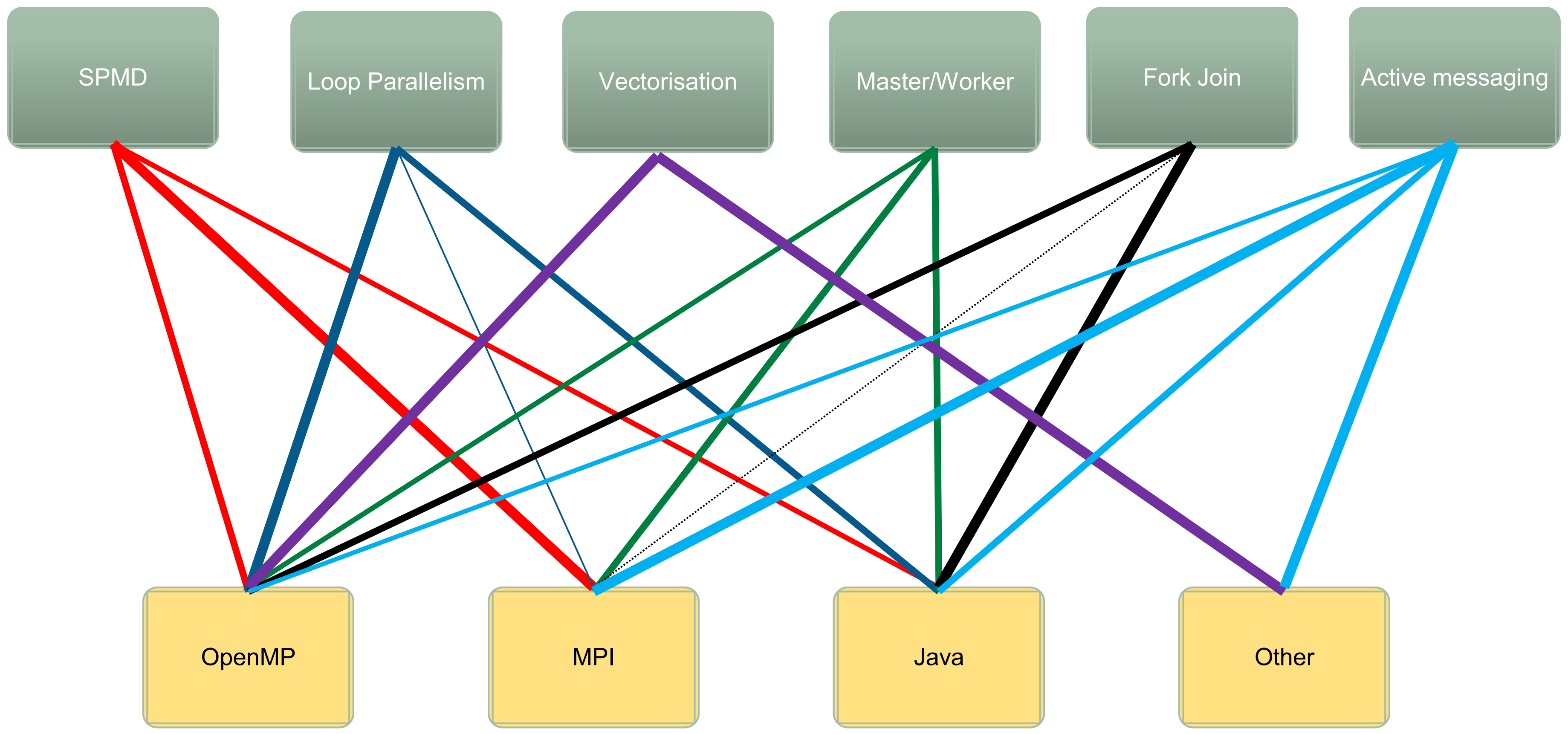
From Algorithm Strategy to Implementation Strategy



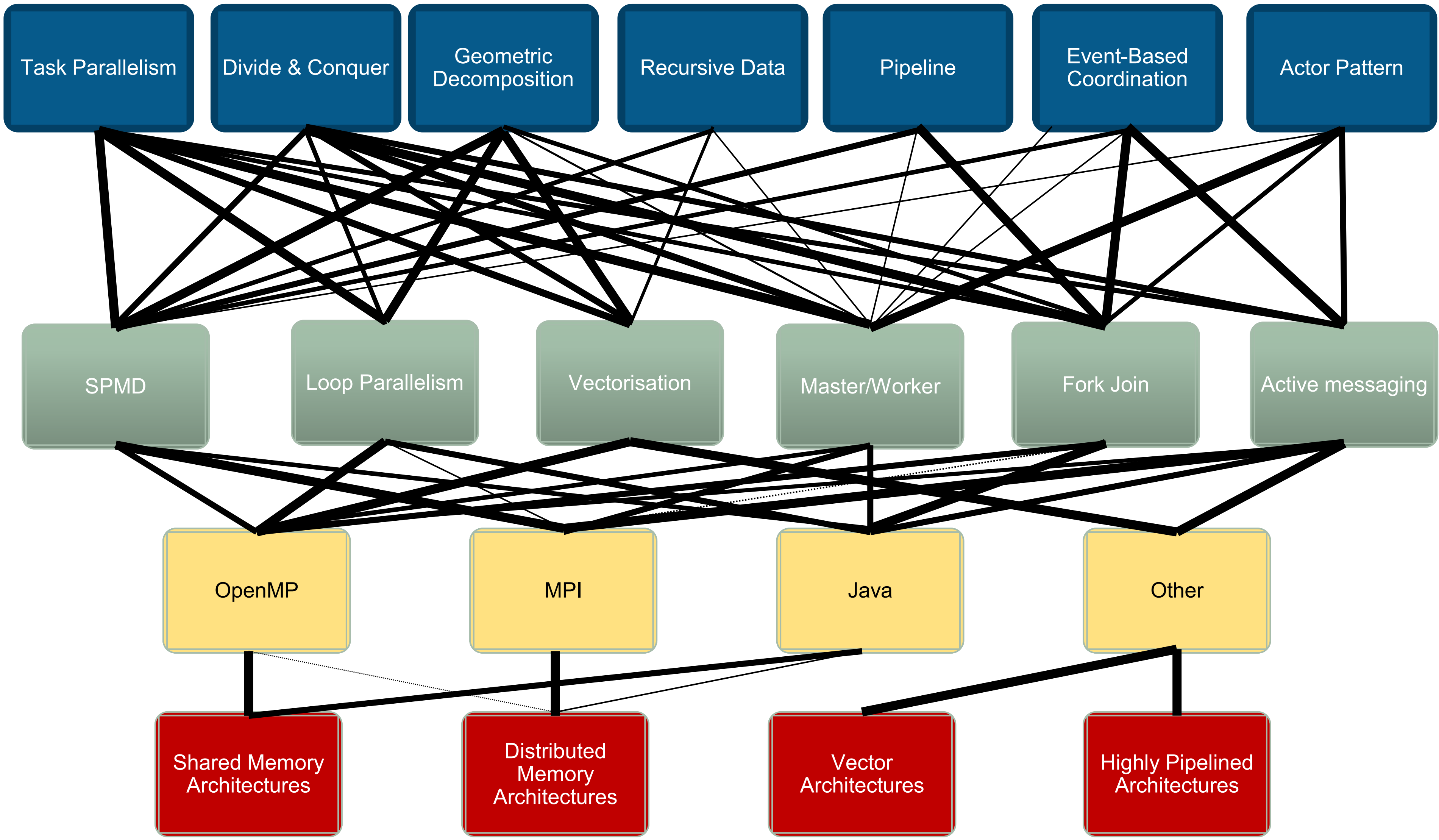
From Algorithm Strategy to Implementation Strategy



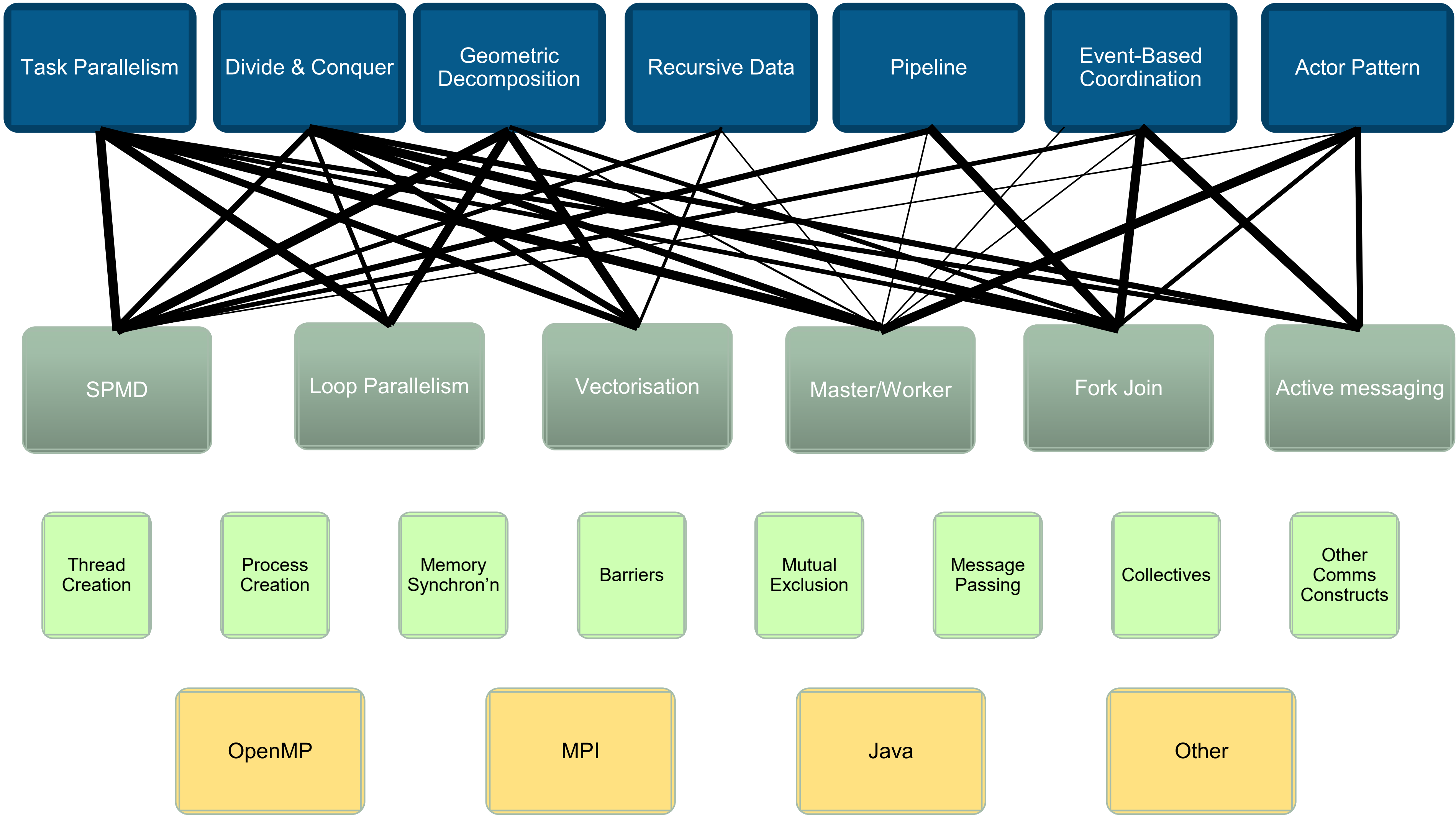
From Algorithm Strategy to Implementation Strategy



From Algorithm Strategy to Implementation Strategy

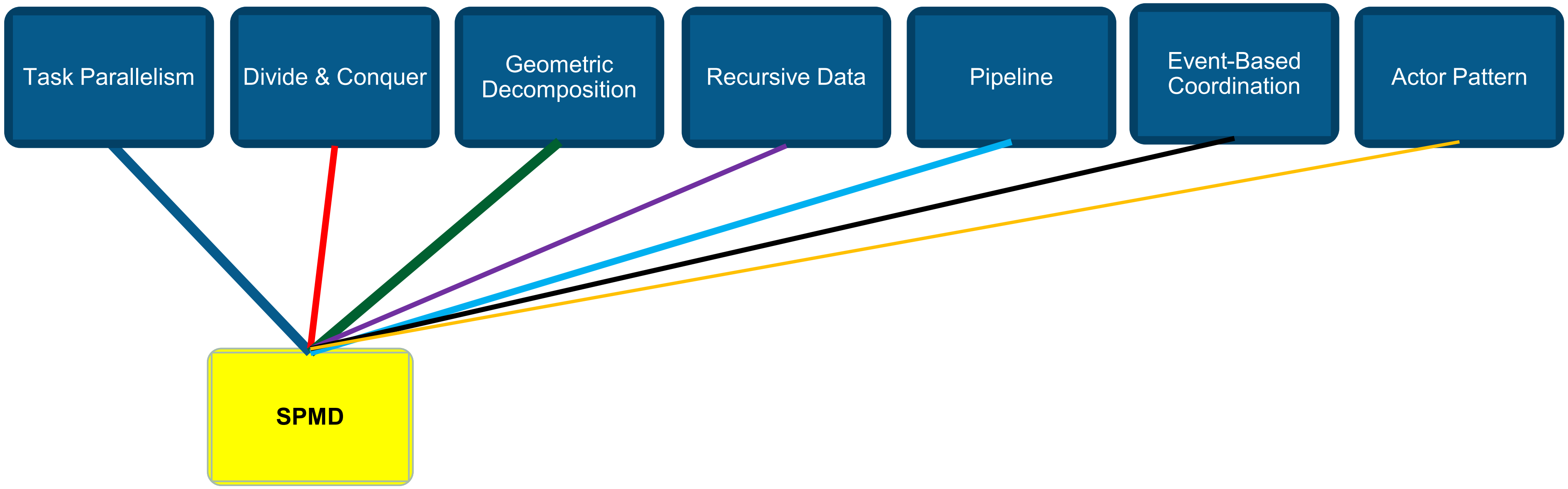


From Algorithm Strategy to Implementation Strategy



SPMD

- SPMD = Single Program Multiple Data
- SPMD is an Implementation Strategy
- Problem:
  - It is the interactions between tasks that introduce most of the difficulty in writing correct and efficient parallel programs. How can parallel programs be structured in order to limit the complexity of the program and ensure these interactions are manageable?



SPMD: Context (1)

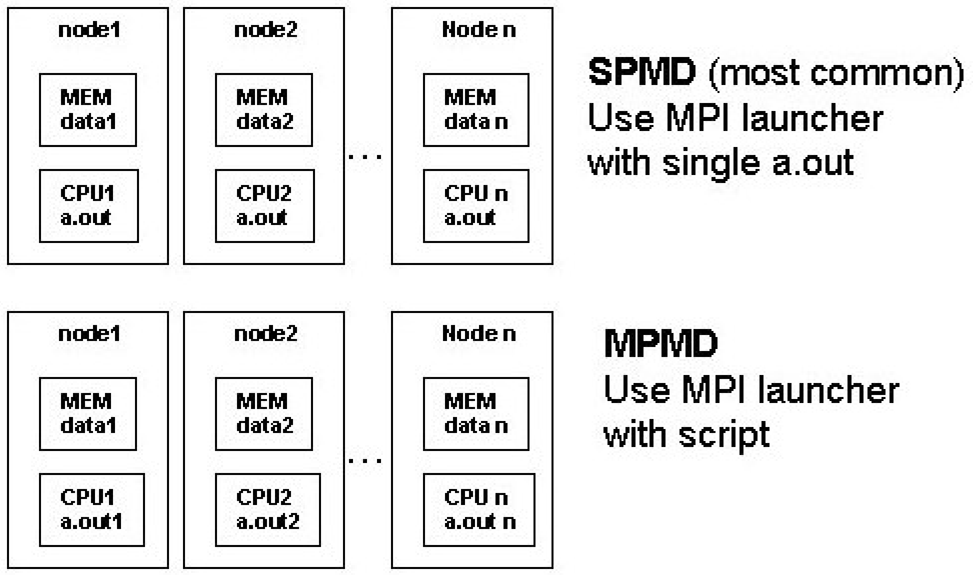
- When writing a parallel program it is (nearly always) the job of the programmer to deal with the challenges of managing multiple tasks on multiple UEs. These tasks and UEs interact, either through exchange of messages or by sharing memory.

- With most parallel algorithms, the operations carried out on each UE are similar
  - Data will be different, and the details of the calculation might be different in different UEs (e.g. boundary conditions)

SPMD: Context (2)

- Since UEs do similar things, it makes sense to encode the parallel algorithm in a single program, executed by all UEs
  - This also means that the interactions between processes are usually described in code right beside the calculations

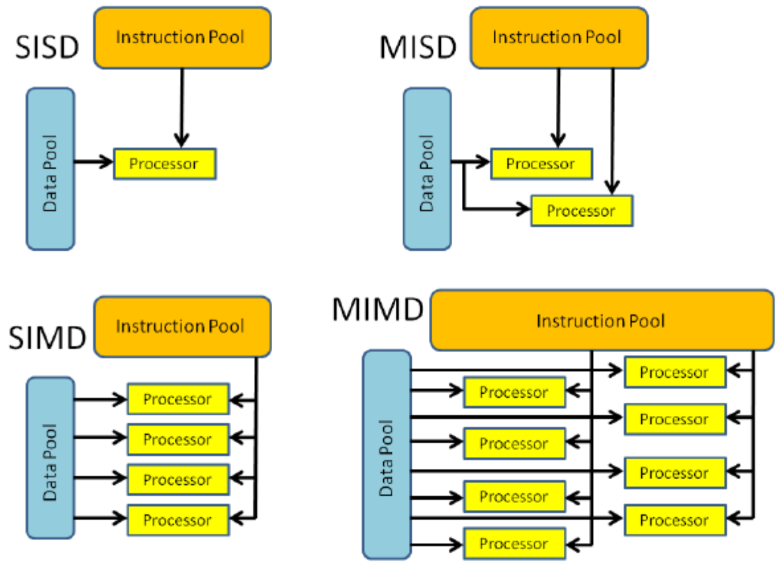
- These are fundamental issues to parallel programming, and this pattern is so common that it can be hard to recognise as a pattern.



SPMD: Context (3)

- Most parallel languages use this pattern as their model of parallelism, so almost any program written in these languages can be described as SPMD

Single Instruction Single Data (*SISD*)
Single Instruction Multiple Data (*SIMD*)
Multiple Instruction Single Data (*MISD*)
Multiple Instruction Multiple Data (*MIMD*)



- SPMD can be, and often is, used in conjunction with other more specialised patterns
- SPMD together with MPMD are considered by some to be the two subdivisions of the MIMD categorisation in Flynn's Taxonomy
- SPMD is fundamental to MPI and also very important to OpenMP

SPMD: Forces

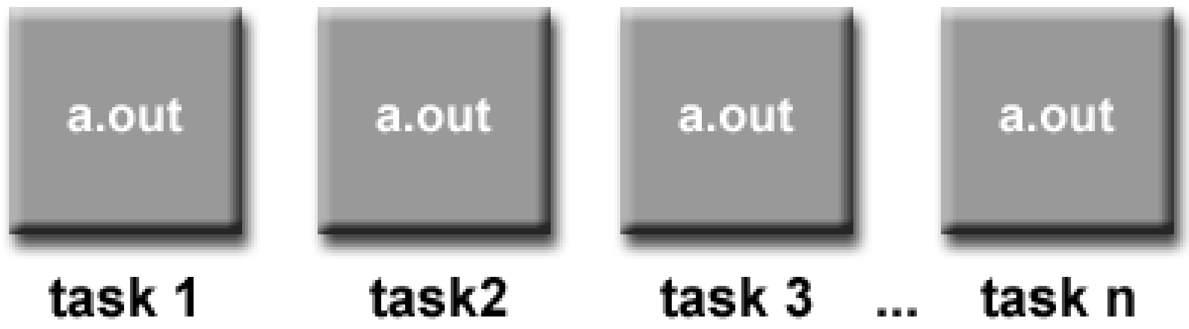
- Using similar code on each UE is easier for the programmer but still allows for different UEs to operate on different data, and run some different operations

- Software typically outlives any given parallel computer
  - This encourages programmers to assume the lowest common denominator in programming environments

- Achieving the highest performance from a given architecture requires that a program be well aligned with the computer's architecture

SPMD: Solution

- Use a single source-code image (i.e. identical binary executables) that runs on each of UEs



- The program will have the following elements:
  - Initialise
  - Obtain a unique identifier
  - Distribute data
  - Run the same program on each UEs, using the unique ID to differentiate between behaviour on different UEs
  - Finalise

SPMD: Comments

- It's easy to write bad (opaque) code in this pattern, particularly if the unique identifier is used in complex indexing algebra

- Highly optimised SPMD codes can sometimes bear little resemblance to the equivalent serial code
  - e.g. they end up being structured around the communication pattern

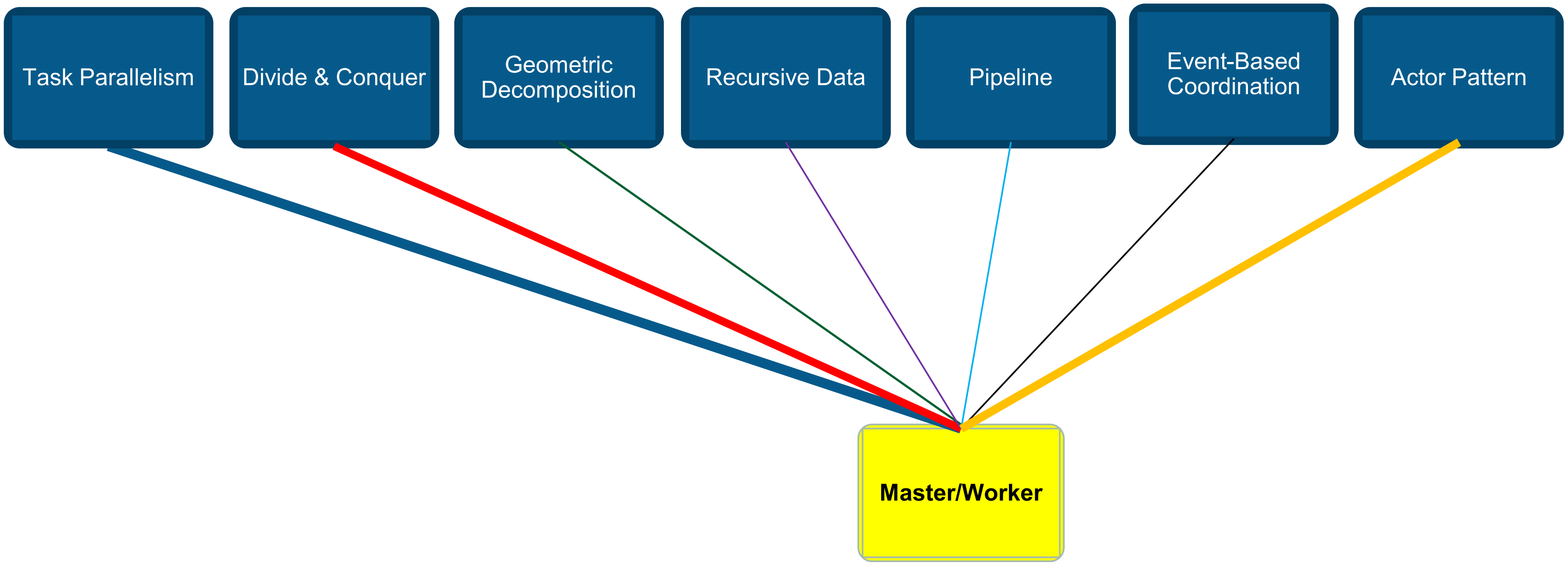
- An important advantage of SPMD is that overheads associated with start-up and termination occur only at the start and end of the program

SPMD: Comments

- SPMD can be highly scalable – many thousands of cores
  - There are often lots of options for complex handling of parallelism, but this is a trade off against simplicity
- Closely aligned with environments based on message passing
  - SPMD is a natural fit with MPI
- SPMD is very general and can be used to implement many of the other patterns

Master/Worker: The Problem

- Master/Worker is an Implementation Strategy, also referred to as *Task Farming*
- Problem
  - How should a program be organised when the design is dominated by a need to dynamically balance the work on a set of tasks among the UEs?



Master/Worker: Context

- Cases where this situation arises often have one or more of the following characteristics:
    1. Workloads associated with the tasks are highly variable and unpredictable
    2. Capabilities of the PEs available differ across the system, or over time
    3. Parts of the hardware might fail during code run

Load balancing issues

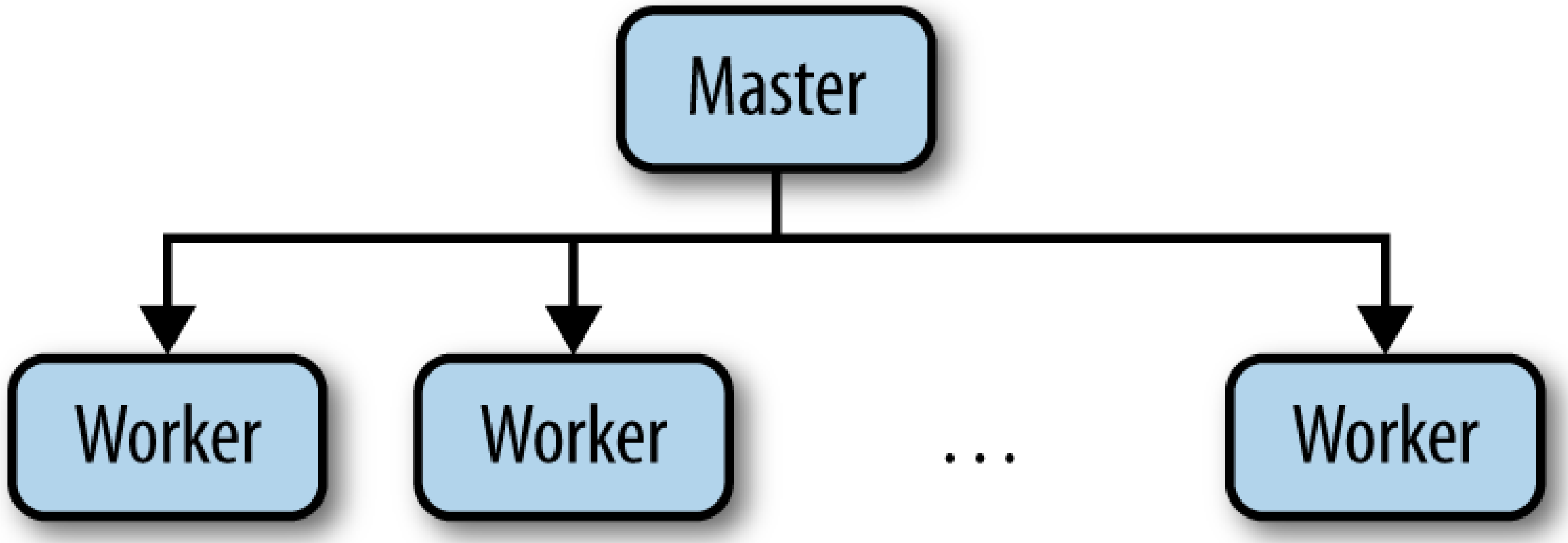
- Tasks are not tightly coupled
    - They don't need to be active at the same time in order to share data
- Particularly relevant for problems following the Task Parallelism pattern where there are no dependencies between tasks

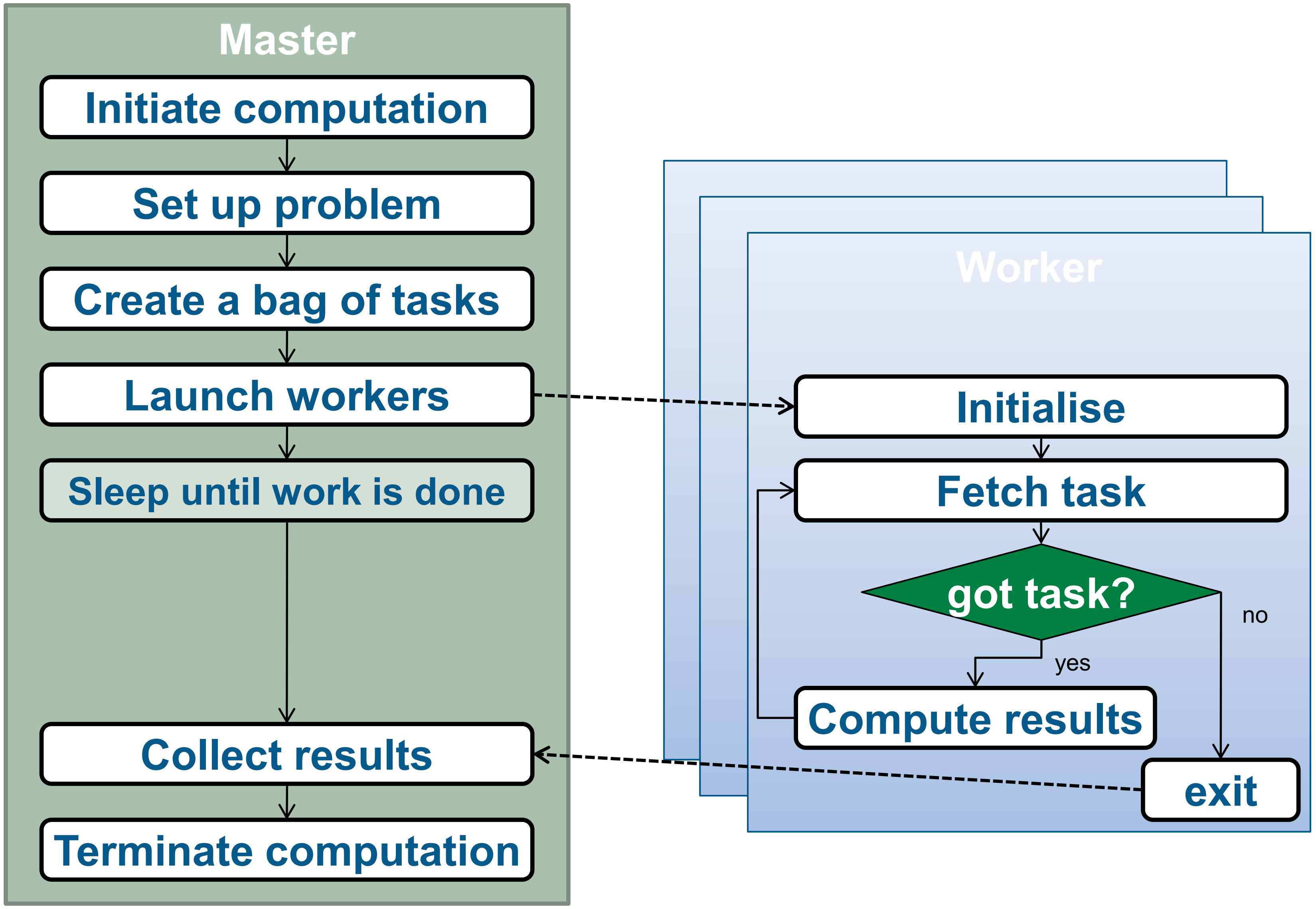
Master/Worker: The Forces

- Work for each task (and possibly capabilities of the PE) varies unpredictably
- Operations to balance load tend to impose communications overhead
  - A balance is therefore required between having a smaller number of larger tasks with fewer communications, and a larger number of smaller tasks which are easier to load-balance.
- Programming logic required to balance load can complicate the implementation of a program and needs to be balanced against code complexity

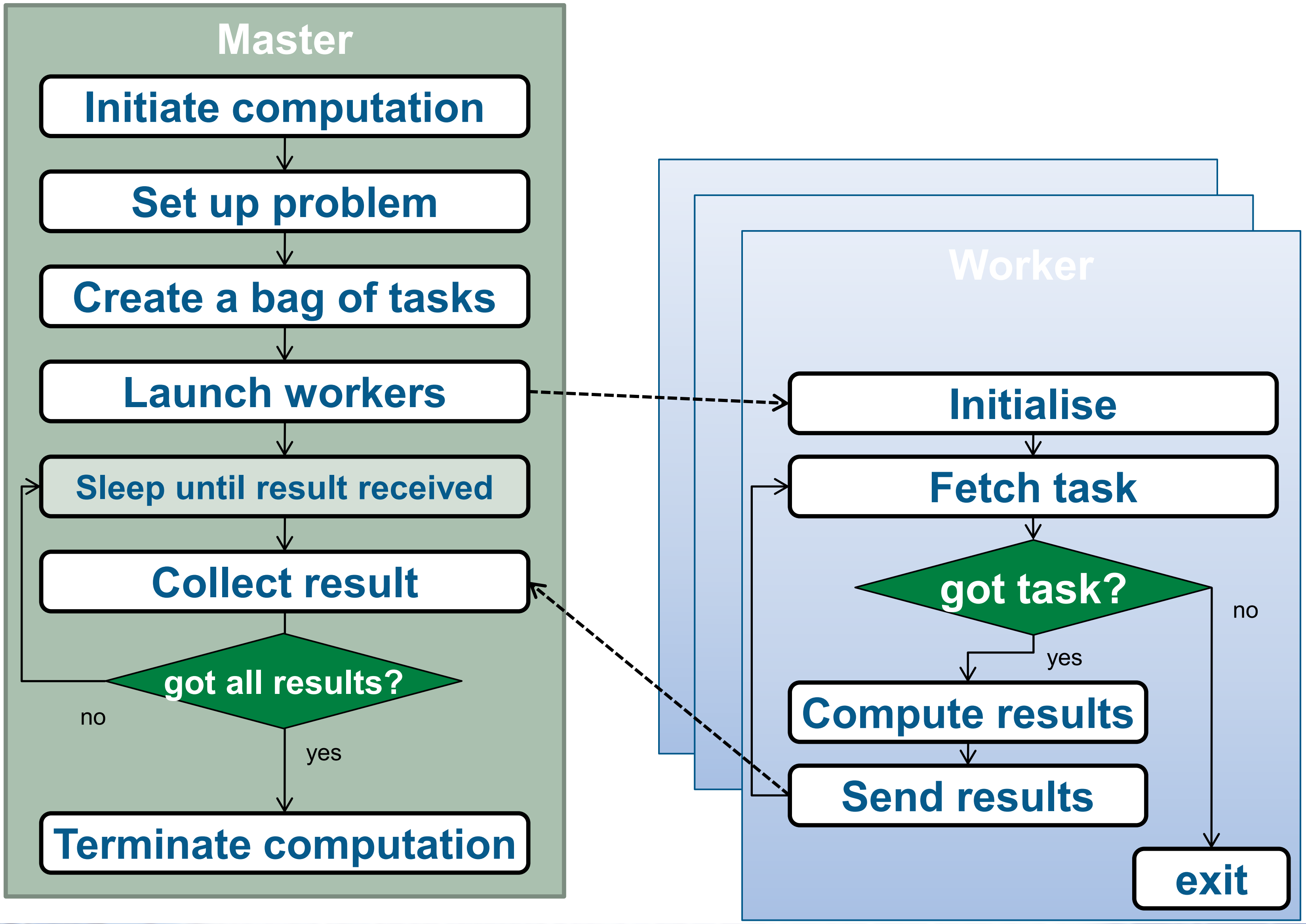
Master/Worker: Solution

- Have the work distributed amongst one logical entity (the master) and one or more other entities (the workers) in such a way that the master splits up the problem and allocates tasks to workers (which can report the results of their task back to the master)



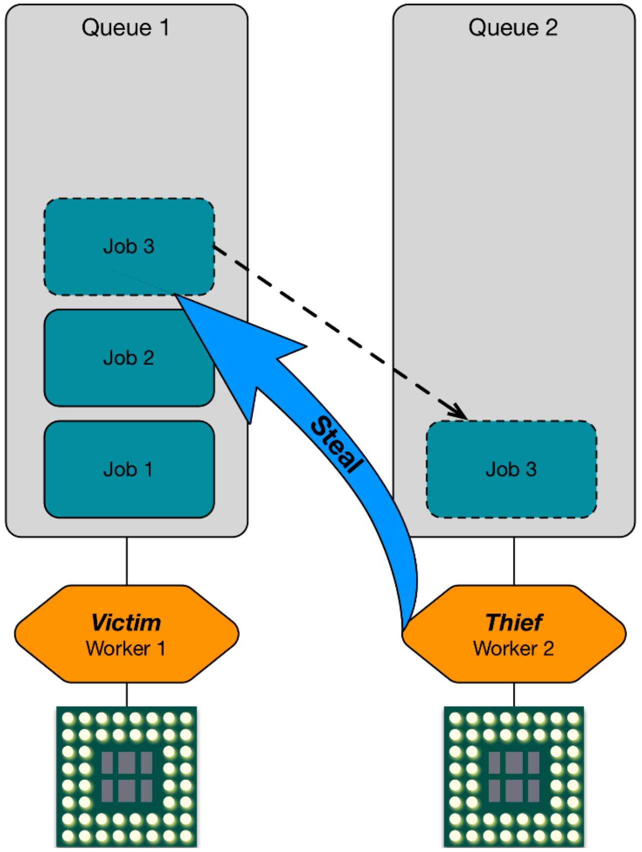

Master
Initiate computation
Set up problem
Create a bag of tasks
Launch workers
Sleep until work is done
Collect results
Terminate computation
Worker
Initialise
Fetch task
got task?
no
yes
Compute results
exit

Most common variation



Other variations

- Avoid a centralised bag of tasks (which can be a bottleneck) by implementing work-stealing

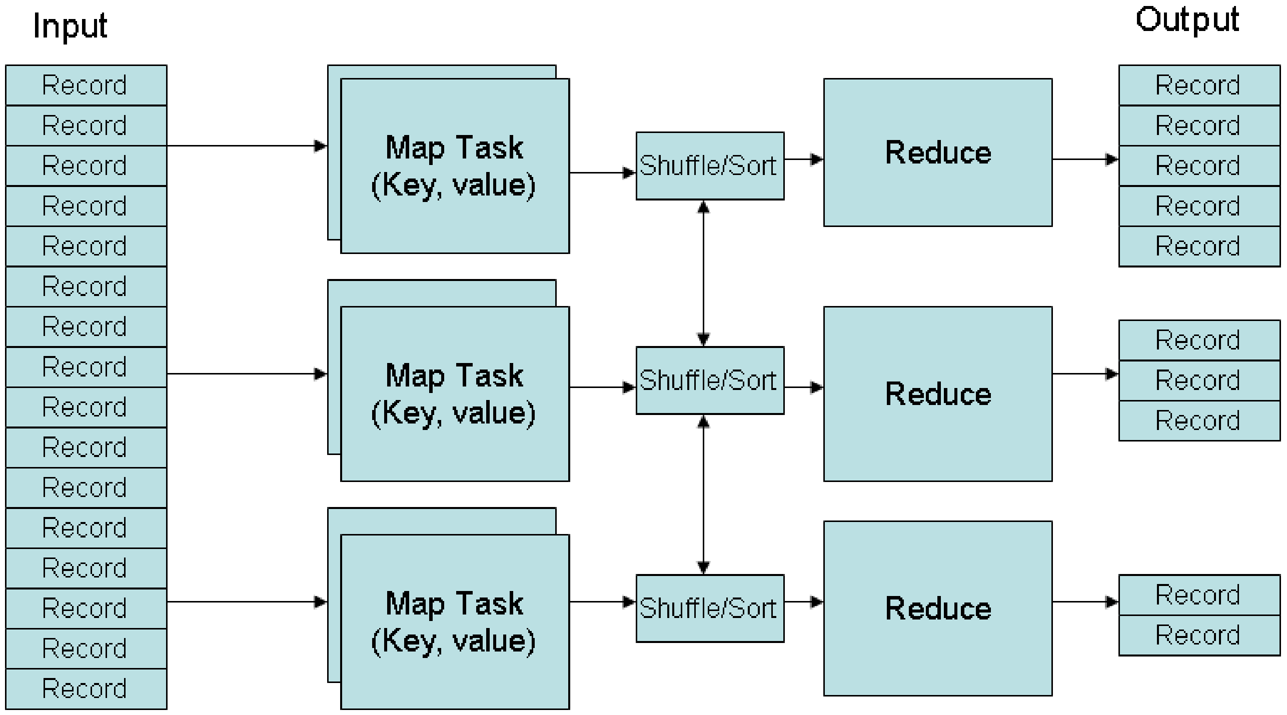


- Instead of sleeping, the master task embarks on one of the unassigned tasks in the bag
  - Can be complicated if it must also be ready to receive messages from workers, and assign new tasks

- Various implementation details can vary, e.g. whether tasks and results are pushed by the master or pulled by the worker

Extensions

- Keep a second queue of assigned tasks
  - Can be used to introduce a level of fault-tolerance
- Pass the results to a different entity from that which produces the tasks
  - MapReduce is based on this idea



Detecting Completion – Simple case

- In the simplest case, all tasks added to bag before workers begin
  - Workers continue until there are no tasks left, and then terminate
  - Master continues until it has received (and processed) results from all tasks, then terminates
- Alternative case: Master checks for global completion condition, then places “poison pills” in the shared queue of tasks (or empties this queue)
- Alternative case: Worker detects completion criteria
  - Worker reports this along with results to master, which proceeds as above

Detecting Completion – More complex

- The hardest case: problems where tasks can be created as the program runs (e.g. divide & conquer pattern)

- Not necessarily true that when a worker finishes a task and there are no tasks left in the bag that there is no more work to do
  - Need to ensure that there are no tasks left in the bag *and* there are no workers still running
  - Particular care must be taken when asynchronous messages are used to ensure that there are no tasks in-transit. Can be very hard to do efficiently & simply.

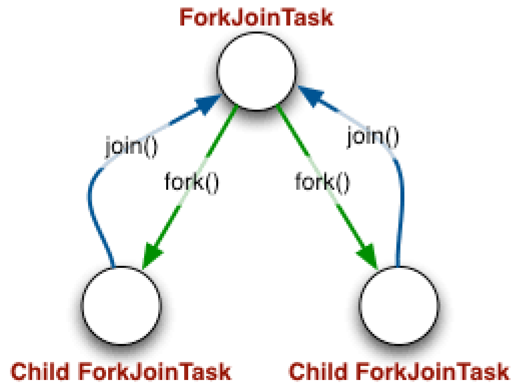
- There are several known algorithms that solve this problem
  - Choice depends on logic of when tasks branch

Implementation Points

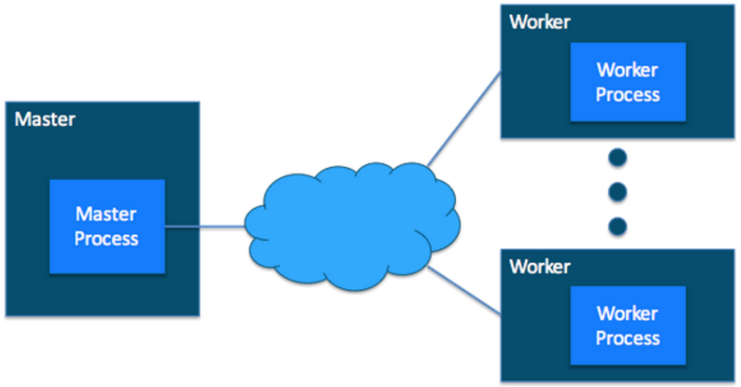
- Bag of tasks can be implemented with *Shared Queue*, although many other mechanisms are possible, e.g.,
  - Tuple space, distributed queue, monotonic counter
- In MPI non-blocking P2P messages work well
- Beneficial if platform provides mechanism for managing the bag of tasks
  - Either with a full implementation of a shared queue, or at least being able to asynchronously respond to requests for work

Master/Worker and Fork/Join

- The *Fork/Join* pattern can be used to implement master/worker
- Master/Worker can also be used to implement fork/join



- It depends on what support is provided by the programming environment, e.g.:
  - With MPI you typically have a fixed number of processes running
    - These processes can form a process pool, and the processes could be assigned to tasks when a fork needs to be implemented

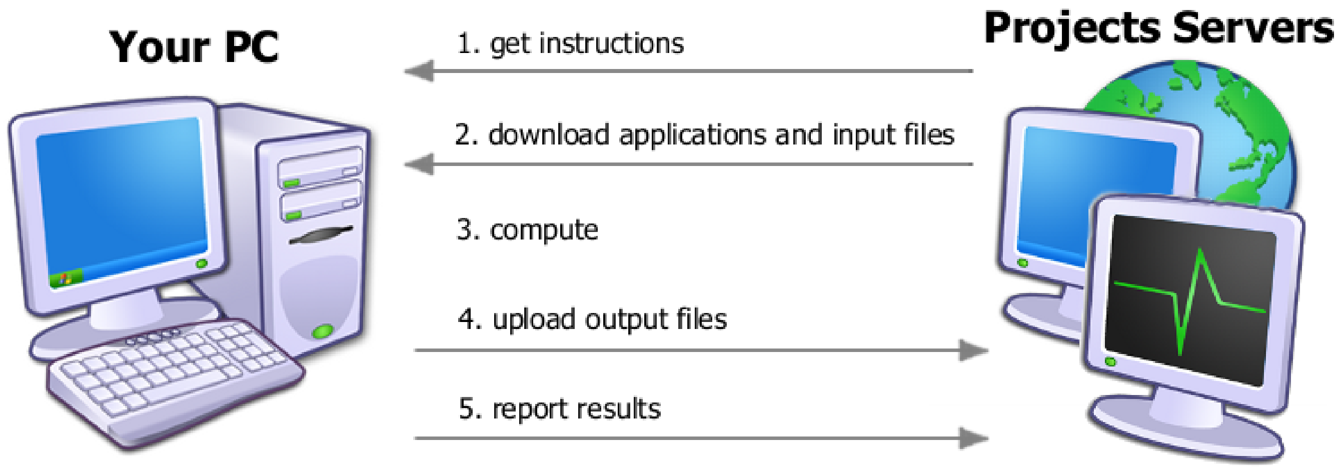


- With OpenMP you have a way of forking processes. These could be used to create a set of threads that could be used as the worker threads in a master/worker model

Master/Worker example: BOINC

- Open source middleware for supporting volunteer distributed computing
  - People donate their CPU time to different projects, often contributing when their machine is idle
  - Over 60 projects listed
- Over 4 million concurrent users, over 400,000 actively computing machines at any one time, approximately 16 PFLOPS overall

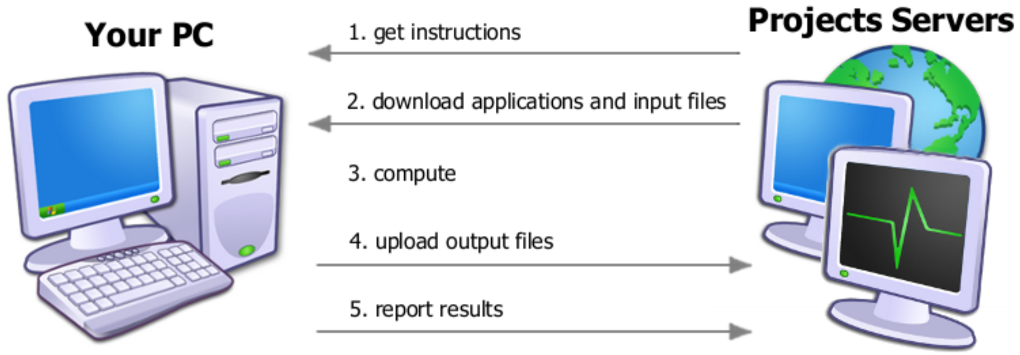
Master/Worker example: BOINC



- Project servers split the problem into work units which are then sent to **two** volunteer machines
  - Project servers are the master
  - Volunteer machines are the workers
  - Two workers for each work unit for correctness reasons
- Flexible enough to take advantage of many different architectures including GPUs

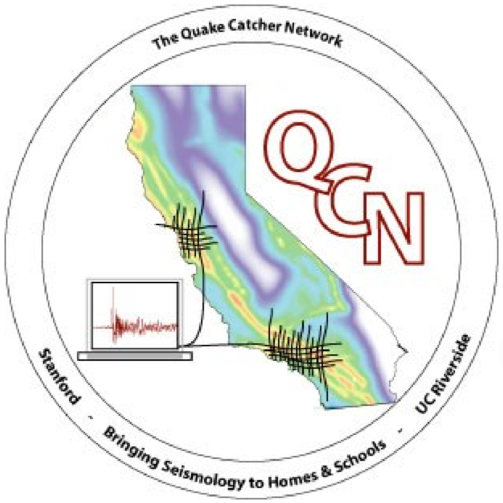
Master/Worker example: BOINC

- The project servers are architected specially for distributed computing
  - Work unit trickling where partial results can be sent back before the overall computation has been completed by the worker
  - Locality scheduling where the master attempts to send units of work to workers who already have some or all of the necessary data files
  - Optimisation of work distribution based on volunteer machines, where tasks are selected based on the capabilities of a worker
  - Different ways of validating the results of work units – from bit comparison to fuzzy matching
  - Multiple project servers (masters) can work together seamlessly



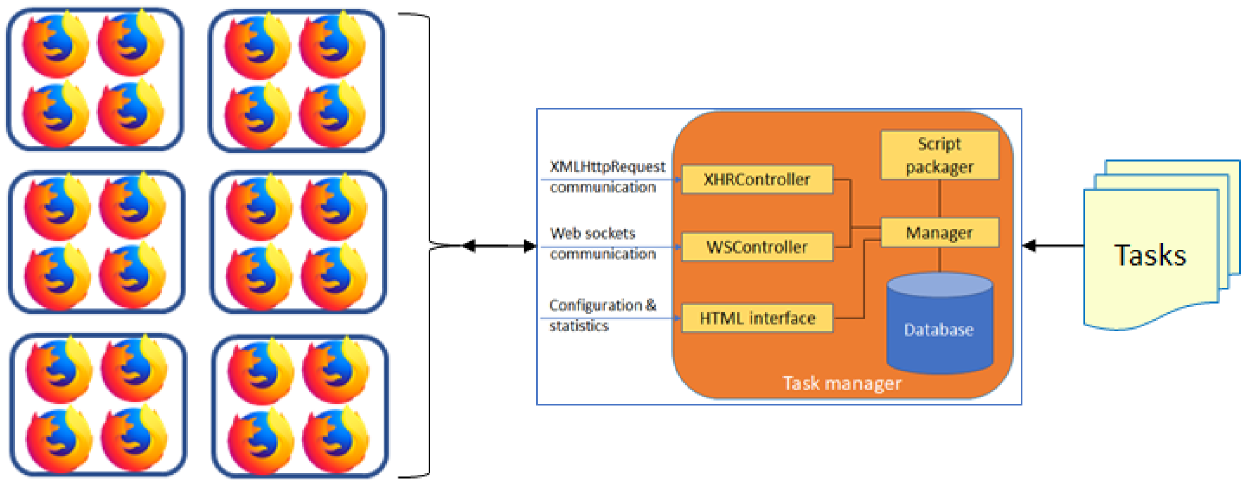
Master/Worker example: BOINC

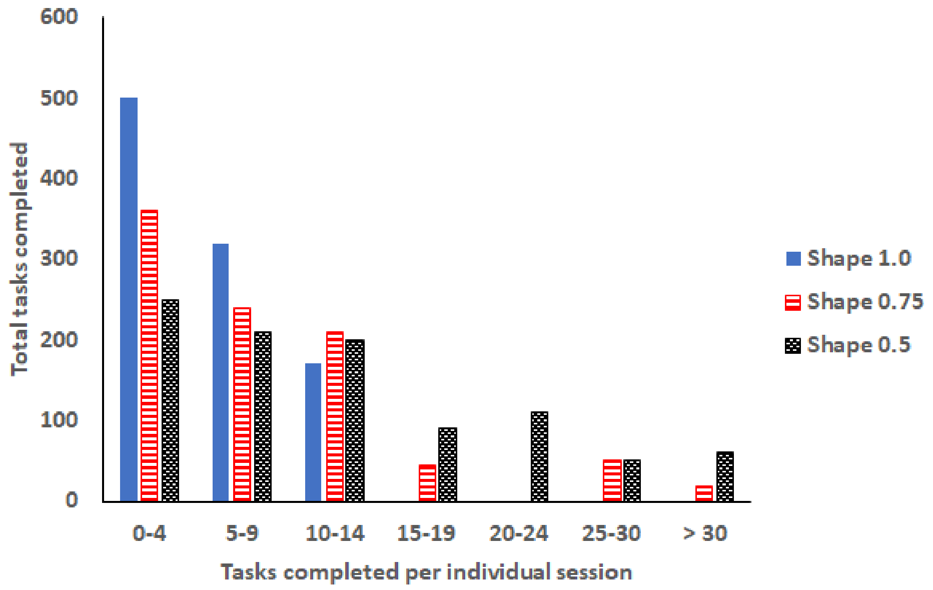
climateprediction.net
the world's largest climate modelling experiment for the 21st century

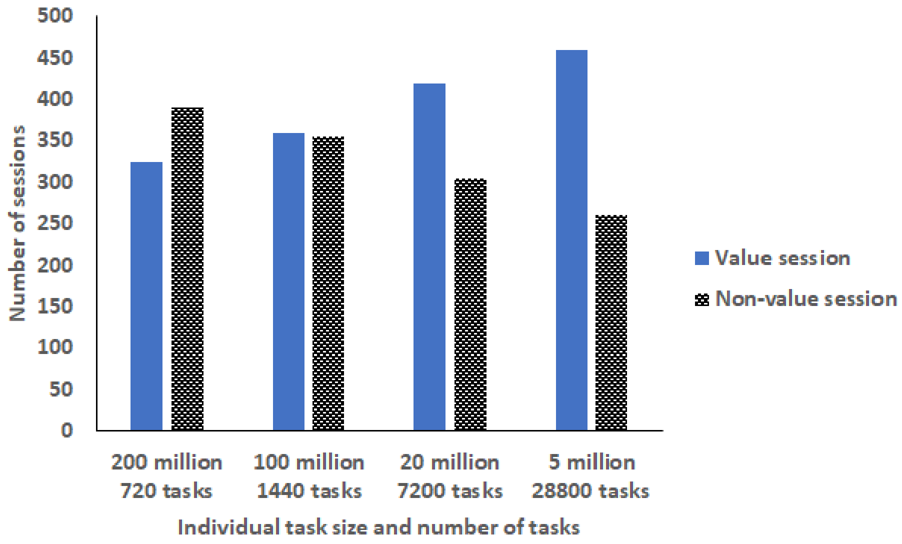

The Quake Catcher Network
QCN
Stanford - Bringing Seismology to Homes & Schools - UC Riverside

BOINC over the web

- On connection to website will forward to task manager which sends over work that is executed via Javascript webworkers in background







Master/Worker: Conclusions

- One master UE, hands out work amongst many worker UEs as they become available
  - Works well when you have lots of independent (or very-nearly independent) tasks
  - Ideally many more tasks than workers
  - Generally no mechanism to ensure “large” tasks don't execute last
- Particularly useful when the work associated with tasks that
  - Involve unequal or unknown amounts of work
  - Can give rise to other tasks as the program progresses
- Pattern closely related to loop parallelism with dynamic scheduling