Parallel Design Patterns

Implementation Strategies – SPMD, Master/Worker

Reusing this material

How do we implement the algorithm?

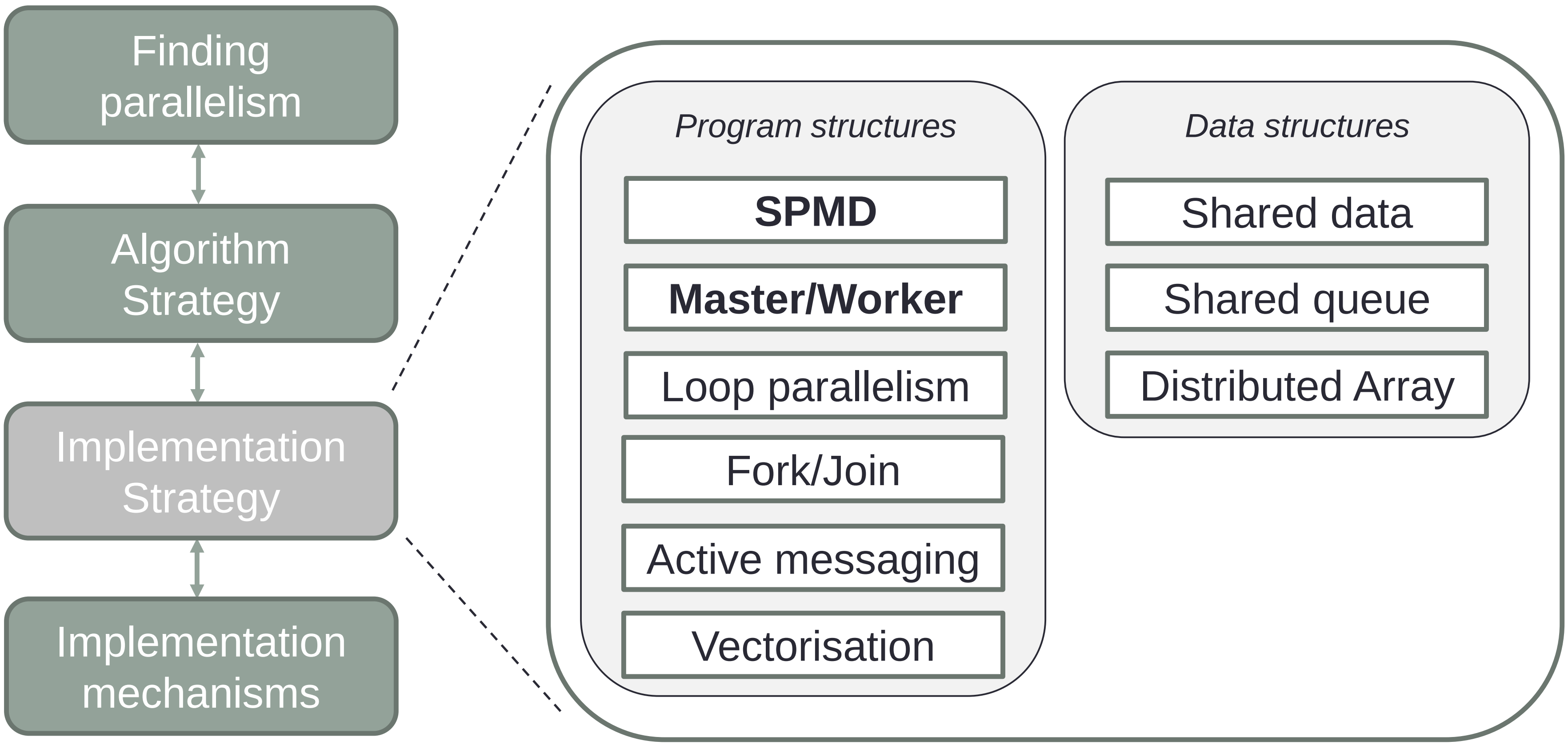
Finding Parallelism
- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, …

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

Implementation Strategy (Supporting Structures)
- SPMD, Master/Worker, Loop Parallelism, Fork/Join, …

Implementation Mechanisms
- UE Management, Synchronisation, Communication, …

Overview of supporting structures



Implementation Strategy – Forces

- How do we structure the software to best support the algorithm(s) of interest?

  - Clarity of abstraction
  - Scalability
  - Efficiency
  - Maintainability
  - Environmental affinity
  - Sequential equivalence

$P \equiv$ # PEs

$$S(P) = \frac{T(1)}{T(P)}$$

$$E(P) = \frac{S(P)}{P} = \frac{T(1)}{PT(P)}$$

Which implementation strategy?

Task Parallelism, Divide & Conquer, Geometric Decomposition, Recursive Data, Pipeline Event-based Coordination, Actor Pattern

Algorithms

- ***SPMD*** and ***Master/Worker*** can be used for all algorithm strategies.
- *Fork/Join*: all except Recursive Data.
- *Vectorisation*: all except Pipeline and Event-based Coordination.
- *Active Messaging*: all except Geometric Decomposition and Recursive Data.
- *Loop Parallelism* can be used with Task Parallelism, Divide & Conquer and Geometric Decomposition.

SPMD, **Master/Worker**, Loop Parallelism, Fork/Join, Vectorisation, Active Messaging

Implementations

Which implementation is likely to be most appropriate?

Algorithm Strategy

Task Parallelism
Geometric Decomposition
Divide & Conquer
Pipeline
Recursive Data
Event-based Coordination
Actor Pattern

SPMD

Implementation Strategy

Task Parallelism
Actor Pattern
Divide & Conquer
Geometric Decomposition
Pipeline
Recursive Data
Event-based Coordination

Master/Worker

Which implementation is likely to be most appropriate?

Algorithm Strategy

Task Parallelism
Geometric Decomposition
Divide & Conquer

Loop Parallelism

Implementation Strategy

Task Parallelism
Geometric Decomposition
Divide & Conquer
Recursive Data

Vectorisation

Which implementation is likely to be most appropriate?

Algorithm Strategy

Divide & Conquer
Pipeline
Event-based Coordination
Task Parallelism
Geometric Decomposition
Actor Pattern

Fork/Join

Implementation Strategy

Event-based Coordination
Actor Pattern
Divide & Conquer
Task Parallelism

Active Messaging

Which implementation mechanism?

SPMD, **Master/Worker**, Loop Parallelism, Fork/Join, Vectorisation, Active Messaging

Implementation Strategies

- *OpenMP* can be used for all implementation strategies.
- *MPI* and *Java*: all except Vectorisation.

OpenMP, MPI, Java

Mechanisms

Which mechanism is likely to be most appropriate?

Implementation Strategy

Loop Parallelism
Vectorisation
SPMD
Fork/Join
Master/Worker
Active Messaging

OpenMP

Mechanism

SPMD
Active Messaging
Master/Worker
Loop Parallelism
Fork/Join

MPI

Fork/Join
Loop Parallelism
Master/Worker
Active Messaging
SPMD

Java

Which implementation technology is best supported?

Shared: OpenMP (strong), Java (strong).

Distributed: MPI (strong), Java (average), OpenMP (weak).

Memory Architectures

- None of the aforementioned implementation mechanisms have specific support for Vector or Highly Pipelined architectures.

SPMD – Single Program Multiple Data

- SPMD is an Implementation Strategy
- 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 – Introduction

- How to manage multiple tasks on multiple UEs?
- The tasks and UEs interact either through exchange of messages or by sharing memory.

- The operations carried out on each UE are similar.
  - data will be different and the details of the calculation might be different for different UEs (e.g. boundary conditions)

SPMD – Introduction

- Since UEs do similar things, it makes sense to encode the parallel algorithm in a single program, executed by all UEs.
  - also means that the interactions between processes are usually described in code right beside the calculations
    - this is convenient, but can also cause problems

- This pattern is so common that it can be hard to recognise as a pattern.

SPMD – Introduction

- Most parallel languages use the SPMD pattern as their model of parallelism.
  - almost any program written in a parallel language can be described as SPMD
  - SPMD is often used in conjunction with other more specialised patterns
  - SPMD together with MPMD are considered by some to be the two subdivisions of MIMD (from Flynn's Taxonomy)
  - SPMD is fundamental to MPI and also very important to OpenMP
    - A “pure” SPMD OpenMP program would consist of a single parallel region

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 different operations.

- Software typically outlives any given parallel computer.
  - 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 UE.

- The program will have the following parts.
    1. initialise
    2. obtain unique identifier
    3. distribute data
    4. run the same program on each UE using the unique ID to differentiate between behaviour on different UEs
    5. 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.
  - code becomes 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.
  - up to 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 (or Task Farming)

- 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?

- The Master/Worker pattern is very good for addressing load balancing issues.
  - workloads associated with the tasks are highly variable and unpredictable
  - capabilities of the PEs available differ across the system, or over time
  - parts of the hardware might fail during code run

Master/Worker – Introduction

- 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.

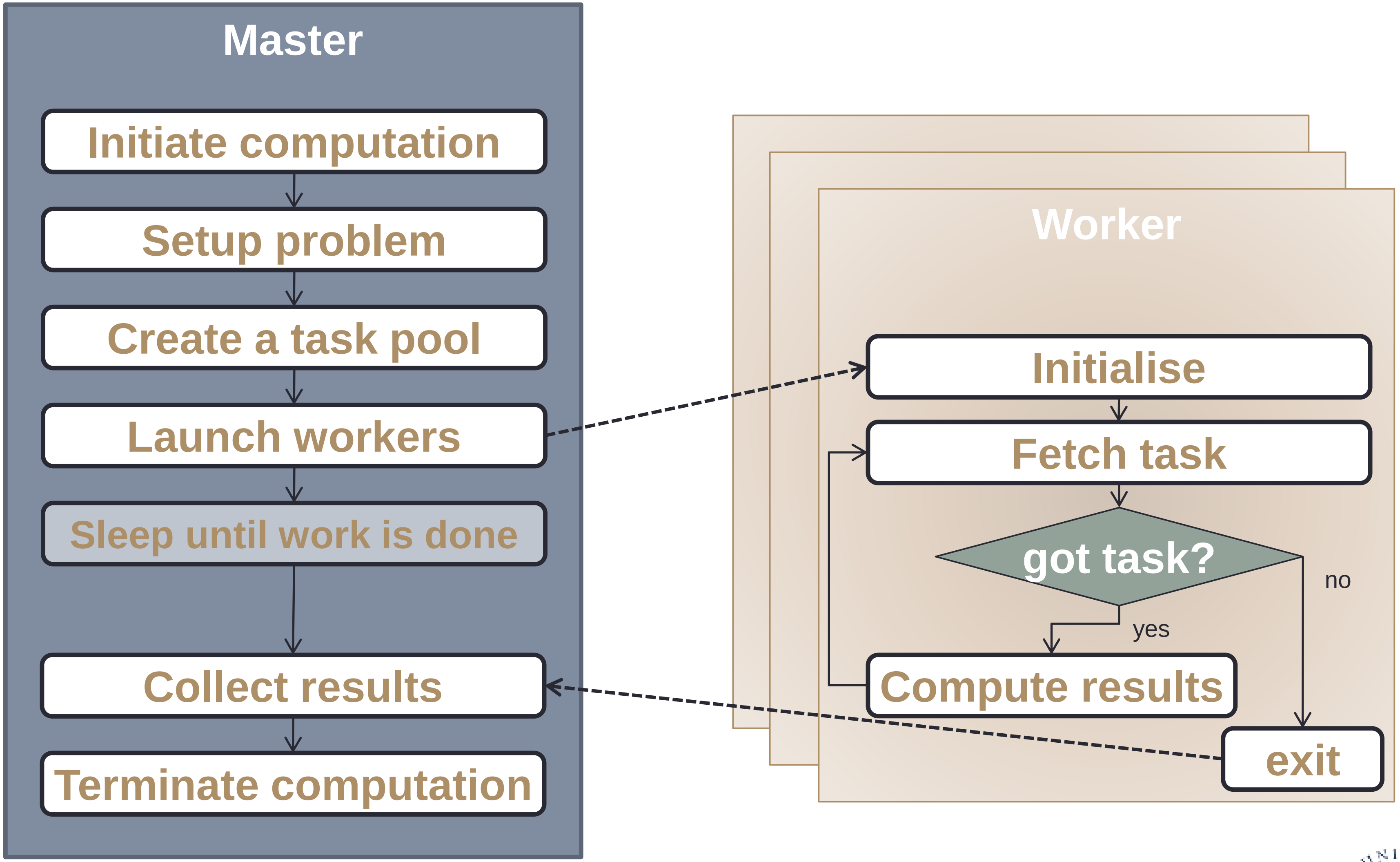
Master/Worker – The Forces

- The 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.

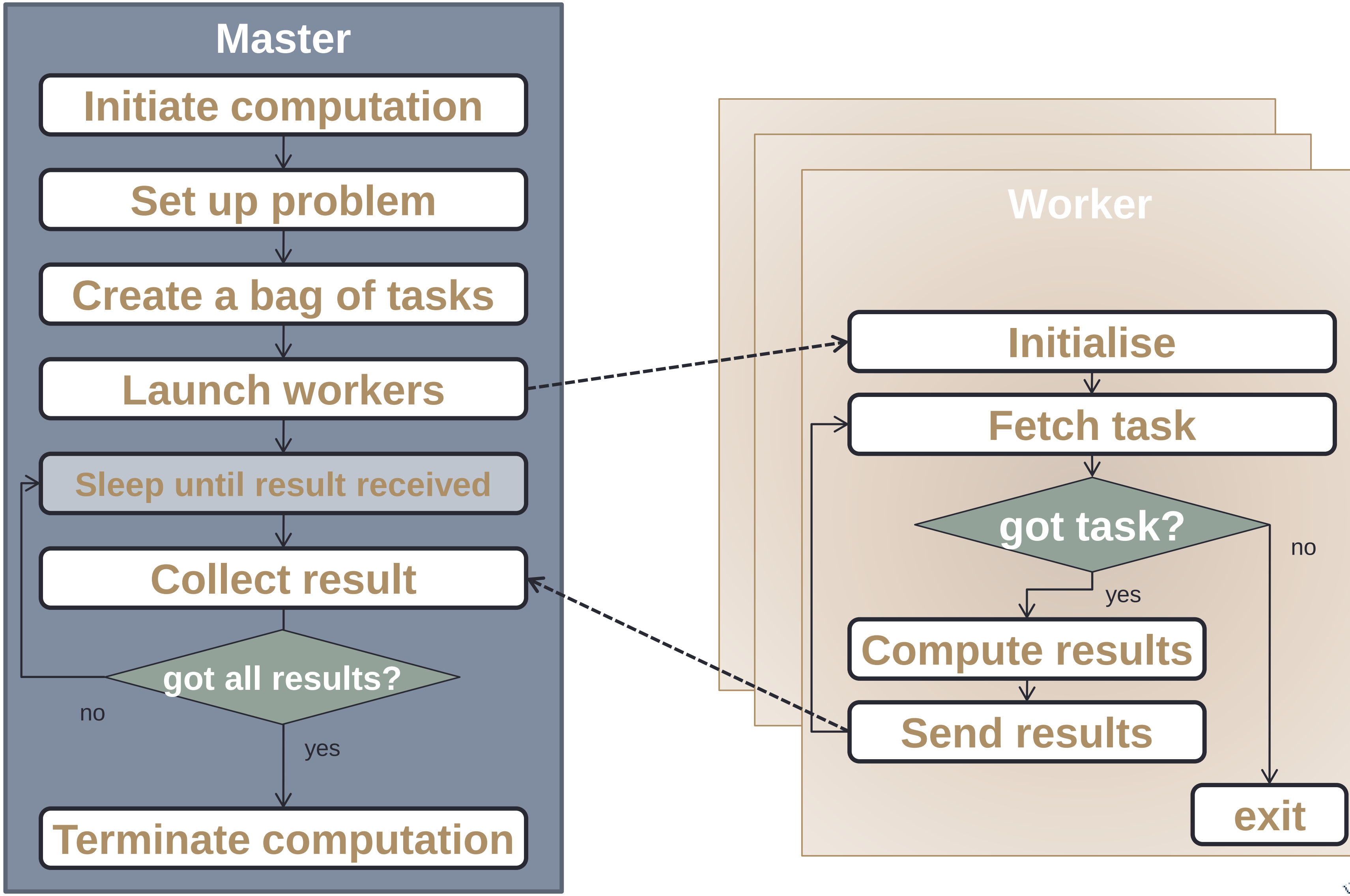
Master/Worker – Solution

- 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.
  - typically, workers report their results back to the master

Master/Worker – Basic solution



Master/Worker – Variation



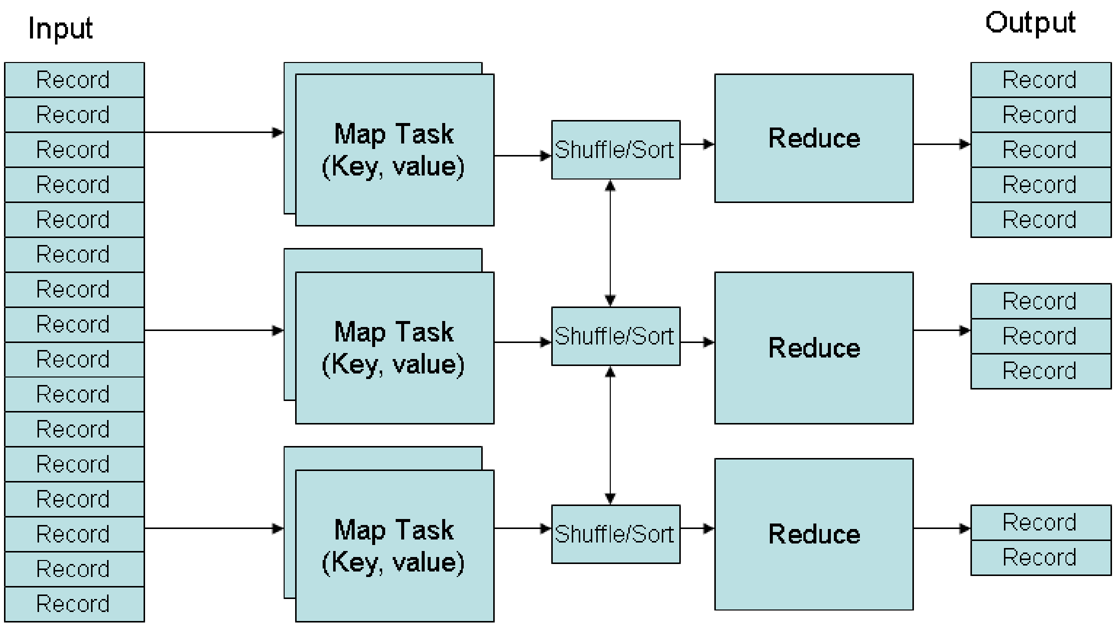
Other Master/Worker variations

- Avoid a centralised task pool (which can be a bottleneck) by implementing work-stealing.
  - instead of sleeping, the master task embarks on one of the unassigned tasks in the pool
    - can be complicated if it must also be ready to receive messages from workers (in order to assign new tasks)

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

Extensions

- Setup 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 pool 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 and then terminates
- Alternatively, Master checks for global completion condition then places "*poison pills*" in the shared queue of tasks (Master could also empty task queue/pool).
  - worker detects completion criteria and then reports this back to master along with results

Detecting Completion – More complex

- The hardest case is when tasks can be created as the program runs (as for Divide & Conquer pattern).
- An empty task pool does not necessarily mean that there is no more work to do.
    - need to ensure that the task pool is empty and that 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
- There are several known algorithms that solve this problem.
    - choice depends on logic that controls when tasks branch

Implementation Points

- Task pool can be implemented with Shared Queue, although many other mechanisms are possible.
  - tuple space, distributed queue, monotonic counter

- Master/Worker pattern works well on clusters and SMP machines.
  - SMP beneficial if input or output data for tasks is large
- Beneficial if platform provides mechanism for managing the task pool.
  - 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

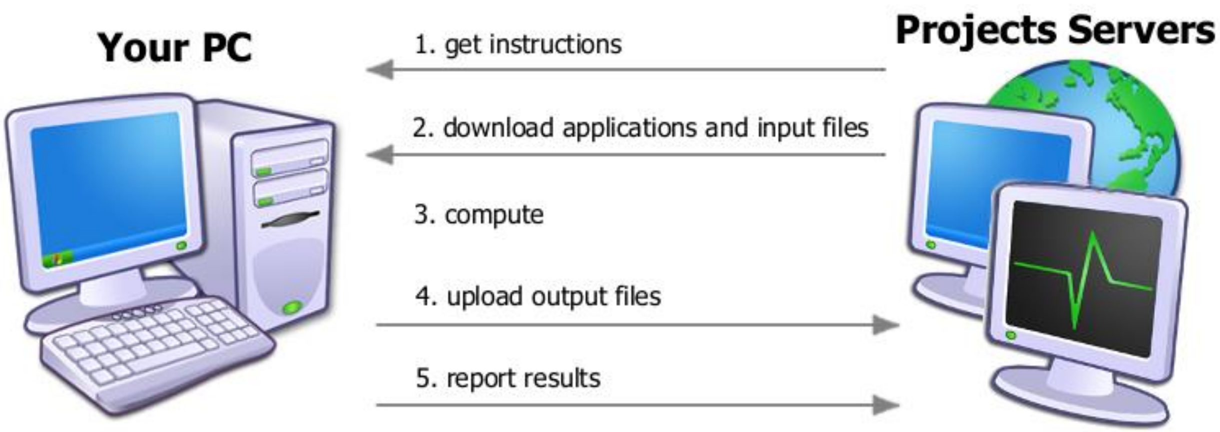
- The Fork/Join pattern can be used to implement Master/Worker.
  - and vica versa

- This possibility depends on what support is provided by the programming environment.
  - 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 at any one time, approximately 16 PFLOPS overall.

Master/Worker example – BOINC



- Project servers (masters) split the problem into work units which are then sent to volunteer machines (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

- 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

SETI@HOME

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

EINSTEIN@HOME
EINSTEIN@HOME
Catch a Wave From Space

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

Rosetta@home
Protein Folding, Design, and Docking

PrimeGrid

Master/Worker – Comments

- Master/Worker algorithms have good scalability as long as...
  - number of tasks greatly exceeds number of workers
  - load imbalance is handled appropriately
    - avoid having large tasks execute last, otherwise other UEs could be left idle
- Management of the task pool can require global communication.
- Master/Worker not tied to any particular platform, but useful if there are structures to support managing the task pool.
- Master/Worker is closely related to Loop Parallelism with dynamic scheduling.
- Pattern can be applied to large scale distributed computing.

Master/Worker – Conclusions

- One master UE, hands out work amongst many worker UEs as they become available.
- Master/Worker works well when you have lots of independent (or very-nearly independent) tasks.
- Particularly useful when the work associated with tasks has the following properties.
  - involve unequal or unknown amounts of work
  - can give rise to other tasks as the program progresses