Embarrassingly Parallel Computations

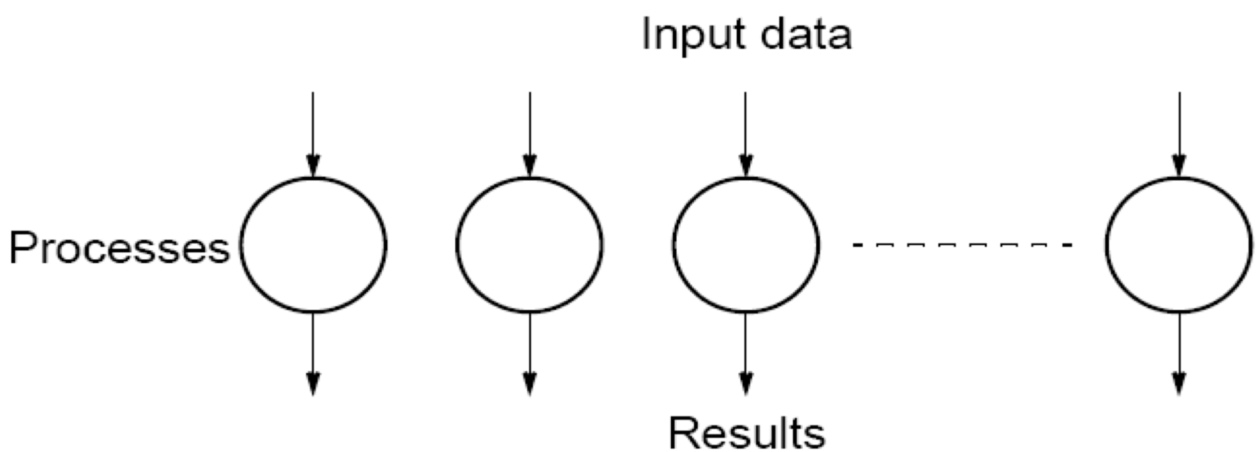
CPSC 424/524
Lecture #7
October 10, 2018

Parallel Techniques

- Embarrassingly Parallel Computations
- Partitioning and Divide-and-Conquer Strategies
- Pipelined Computations
- Load Balancing and Termination Detection

Embarrassingly Parallel Computations (EPCs)

A computation that can be divided into a number of nearly independent parts (tasks), each of which can be executed by a separate entity (thread, cpu, process).



- No (or very little) coordination among tasks
- Tasks have lots of computation relative to the amount of coordination or communication (including startup/shutdown activities)
- Often implemented using "master-worker" or "manager-worker" model

EPC Examples

- Low-level image processing (e.g. rendering)
- Visualization of the Mandelbrot set
- Genomics Algorithms (e.g., BLAST search)
- Monte Carlo Computations

Example: Mandelbrot Set

Set of points c in the complex plane for which the sequence of points z_0 (=0), z_1, z_2, … computed by:

$$z_{k+1} = z_k^2 + c$$

is "quasi-stable" (will increase and decrease, but not exceed some limit). It can be shown that if z_{k+1} ever exceeds 2, then c is not in the set. Computing the set requires testing many points.

Algorithm used for testing membership in the set:

For a given c, tentatively mark c as in the set. Then iterate:

for ($k = 0, z_k = 0.; k < k_{max}; k$++) {

Compute the magnitude of $z_{k+1} = z_k^2 + c$;

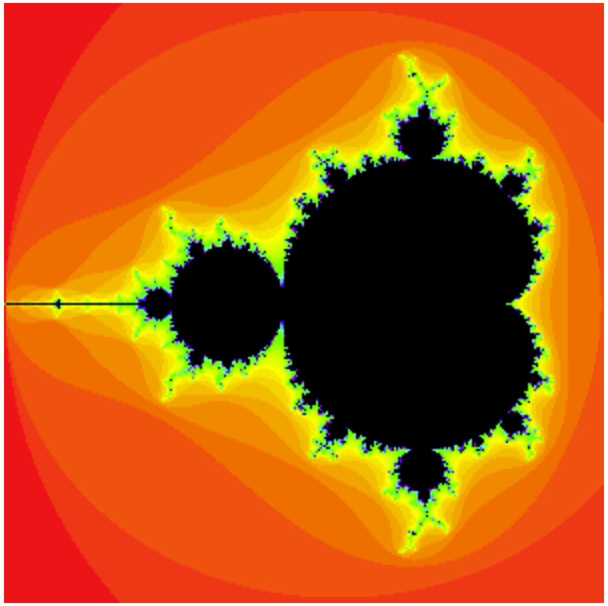
If $|z_{k+1}| > 2$, then mark c as not in the set and break;

}

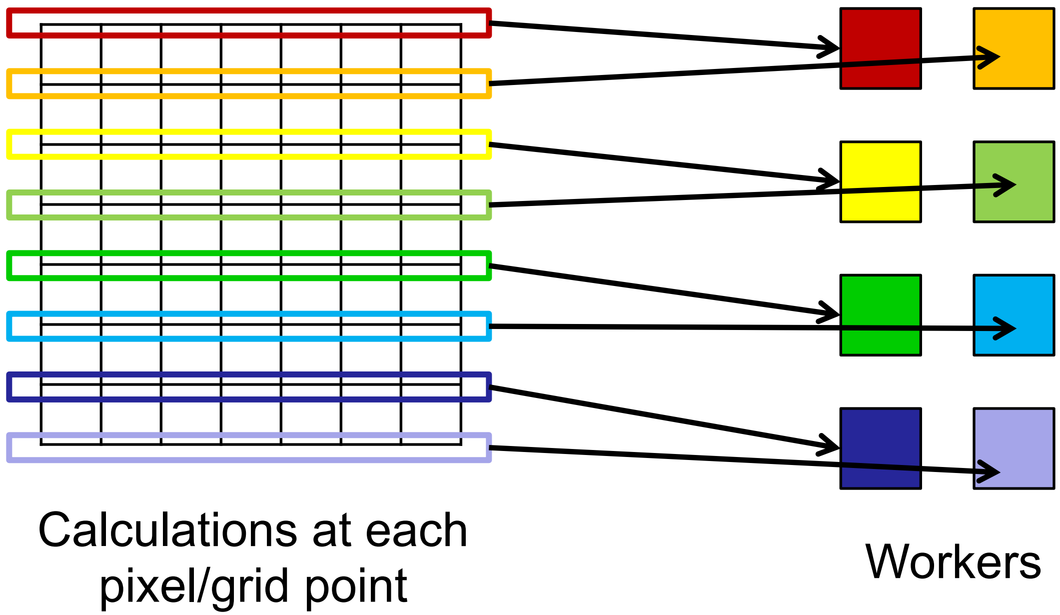
Visualizing the Mandelbrot Set

Superimpose a mesh on a region of the complex plane. Map each point *c* in the mesh to a pixel in a rectangular image, carry out the iteration for *c*, and assign its pixel a color according to:

c in the set:	Black
c not in the set:	Color corresponding to index *k* for which $\lvert z_{k+1} \rvert > 2$



Static ("Up Front") Task Mapping



Problem:

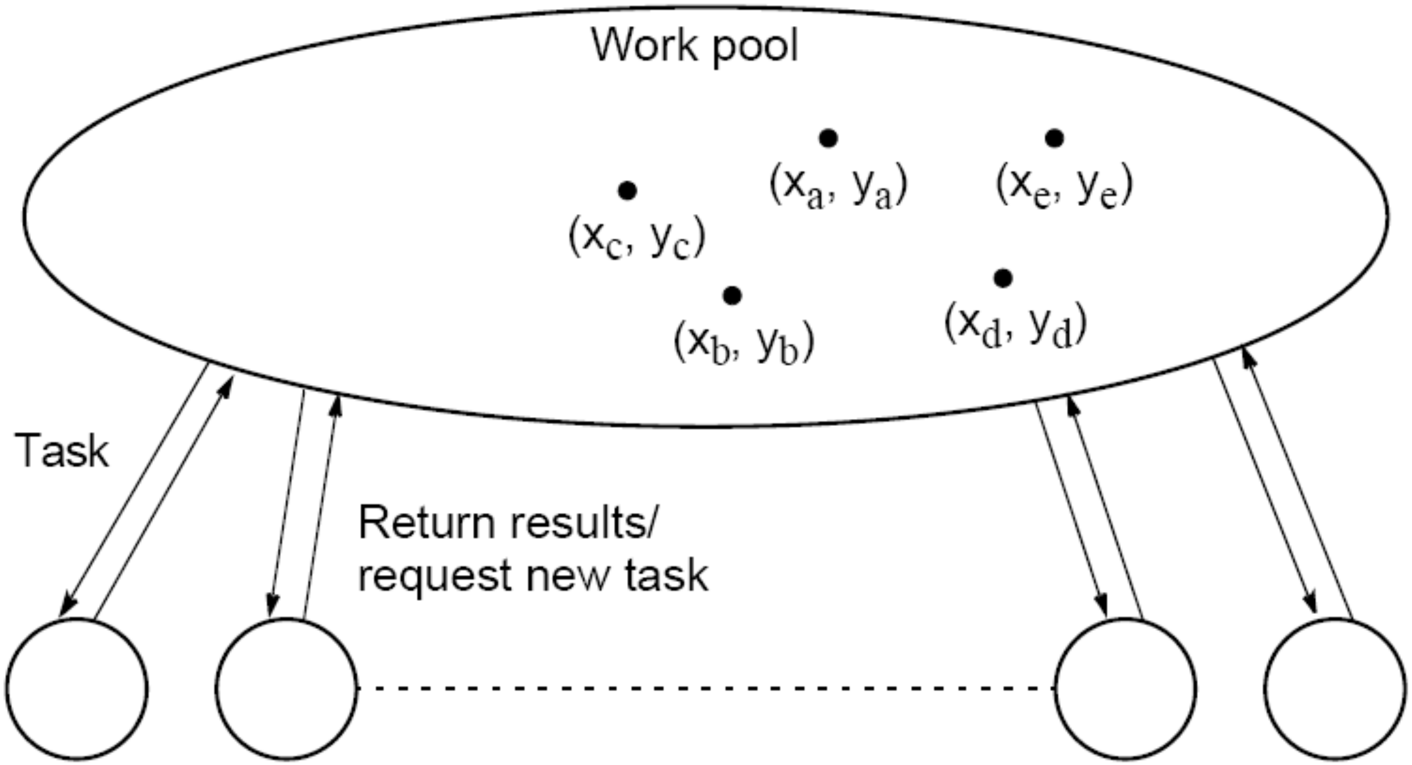
Amount of work per point is unpredictable & highly variable, although there are "good" and "bad" regions where nearby points require similar amounts of work.

One solution:

Use static assignment with random mapping (by pixel)

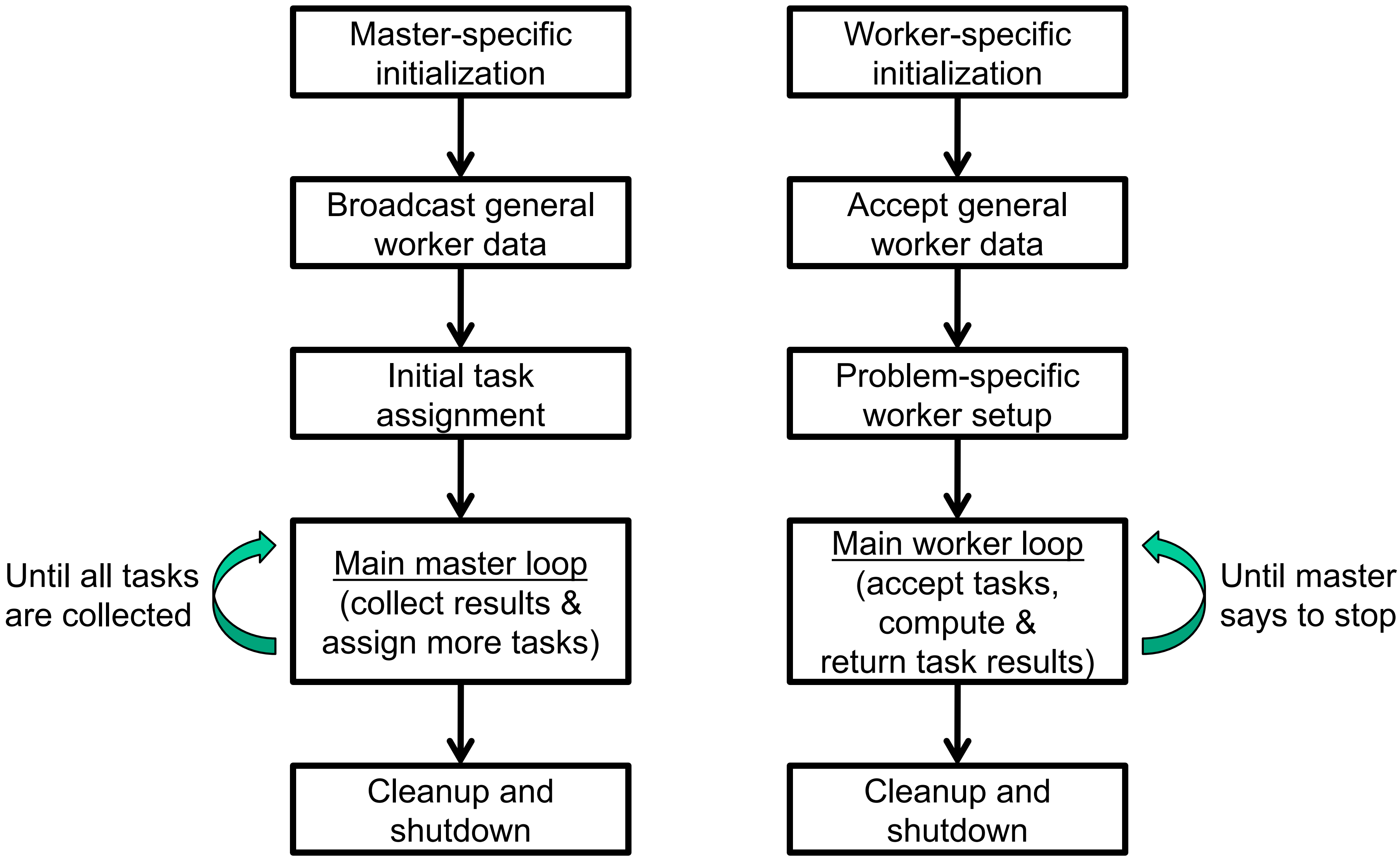
Dynamic ("On-the-Fly") Task Assignment

Idea: Create many tasks (>> number of workers), with each having a large computation-to-coordination ratio. Then give every worker a small number of tasks (often just one) and resupply them on demand.



(How well will this work for the Mandelbrot computation?)

Dynamic Task Pool Master/Worker

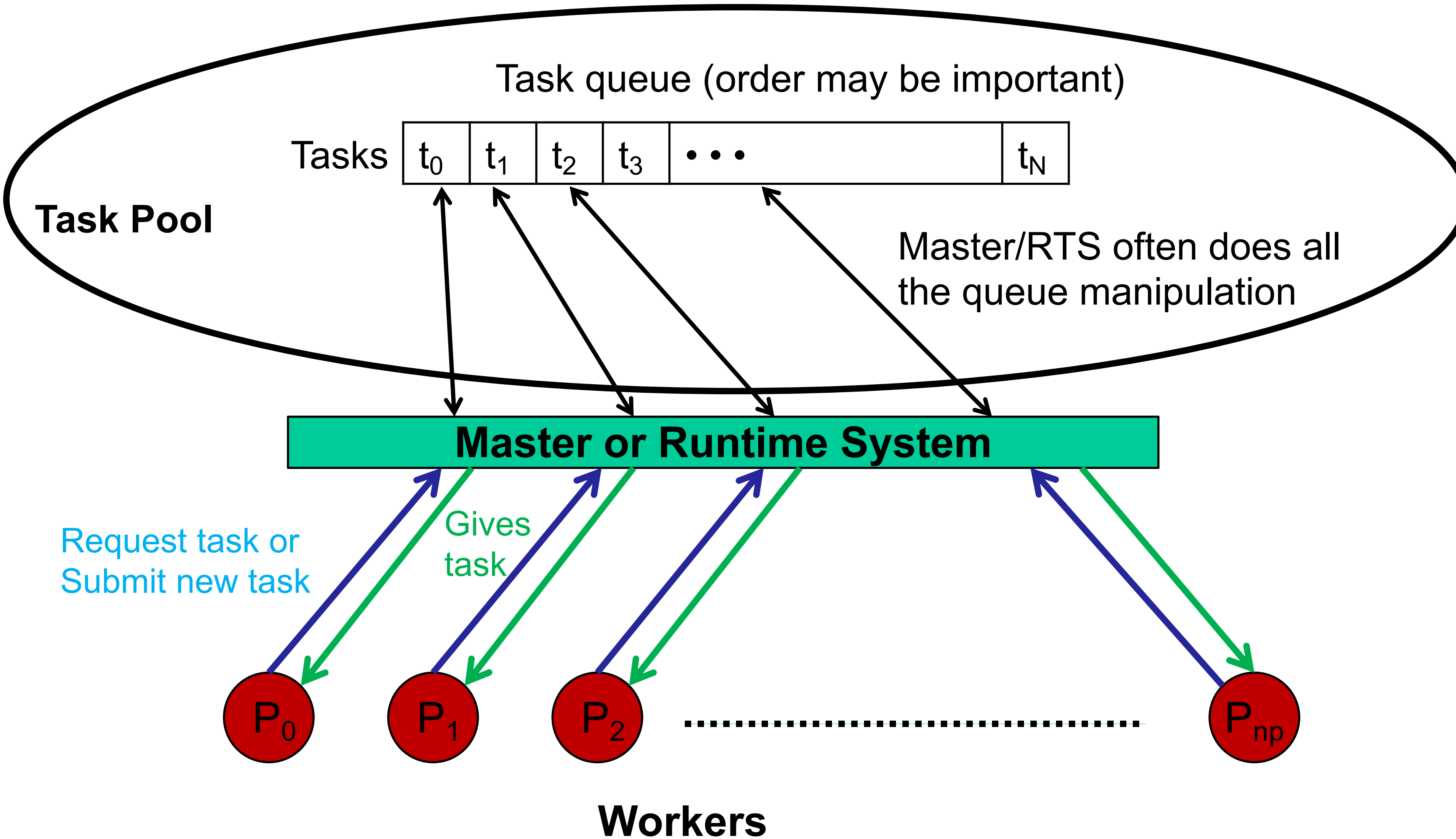


Dynamic Load Balancing

- Centralized Task Pool Management
  - Master creates a task pool ("task bag"). Watermarking may be used.
  - Workers take or receive task assignments when idle. The master may do work or act purely as a manager to balance the load.
  - Workers simply process tasks and (in some applications) create new tasks that are added to the task pool.
  - Termination: Master (or runtime system) terminates when all tasks are done and workers are in states where no more tasks will be added.

- Decentralized
  - Master passes out all/initial tasks to workers
  - Workers cooperate to balance the load.
  - Master may behave as a worker in addition to its master role
  - Termination: Often more complex since no single process or thread really knows everything

Centralized Task Pool Model



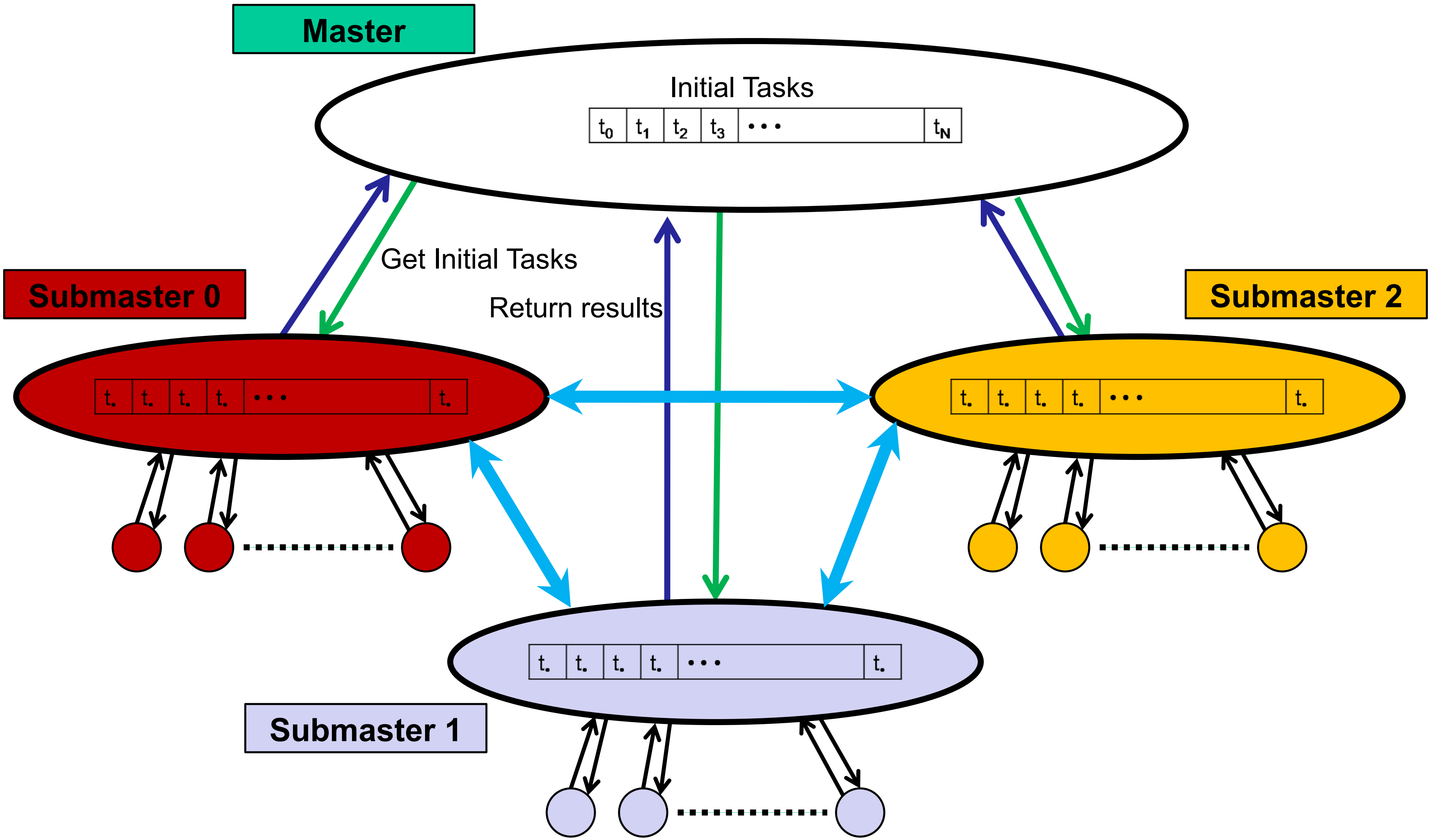
Termination

- Task Queue must be empty **and**
- All tasks that will ever be created must be complete
  - For fixed task set: Master/RTS creates tasks and tracks completion of tasks and collection of results. That's what the Mandelbrot code does−either explicitly (e.g., via OMP loops) or implicitly (e.g., via task synchronization points).
  - For dynamic task set (when workers can submit new tasks):
    - All workers must be idle. (For example, they are all waiting for tasks when none is available.)
    - **Not sufficient** to terminate when task queue is empty if one or more workers might still be able to submit new tasks

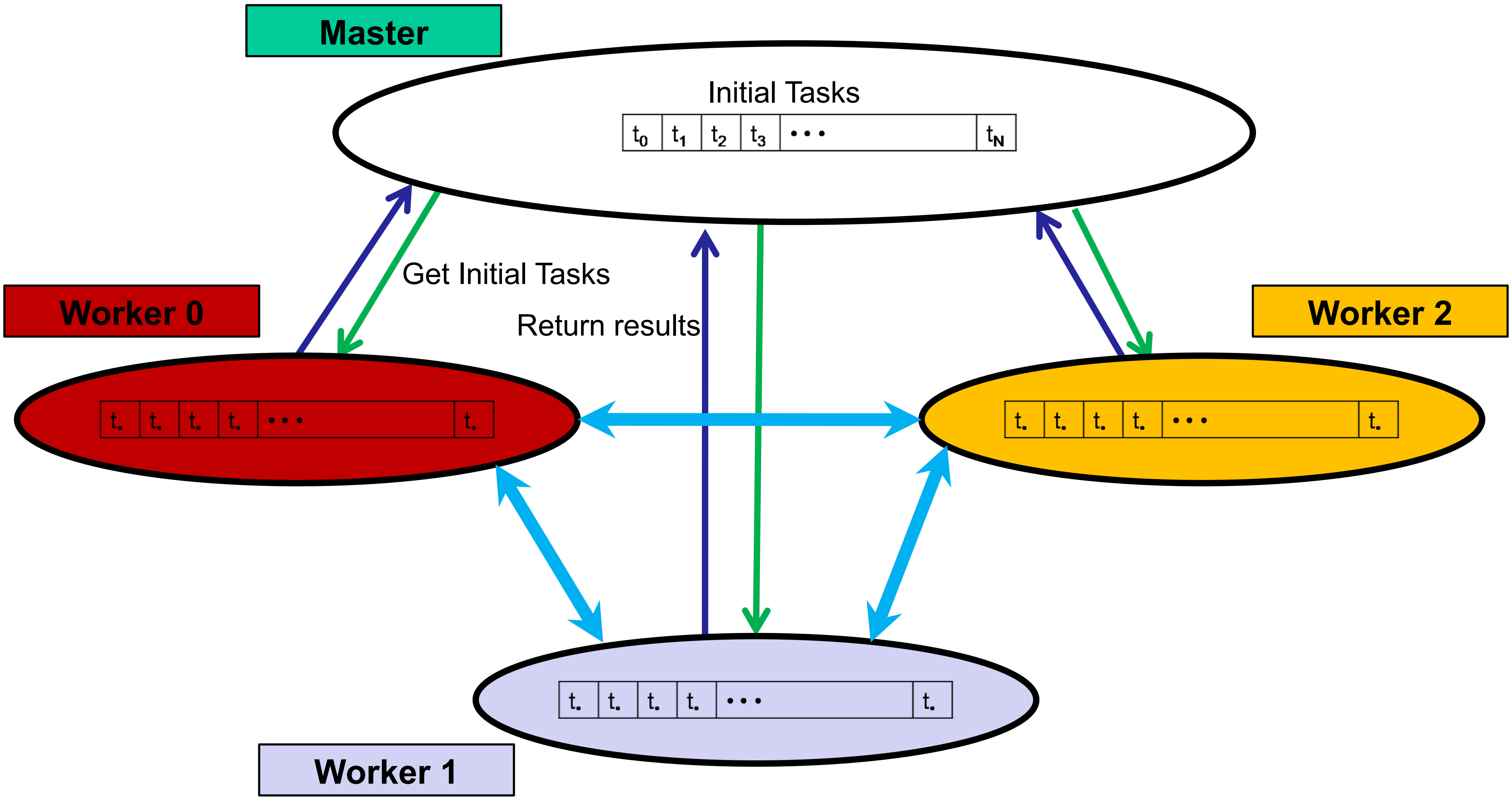
Decentralized Task Pool

- The task pool is distributed among several processes, e.g.:
  - Hierarchy of “submasters,” each with separate worker sets, or
  - “Dual-mode” workers, acting as both submaster and worker

Decentralized Task Pool Model (Submasters)



Decentralized Task Pool Model (Dual-mode workers)

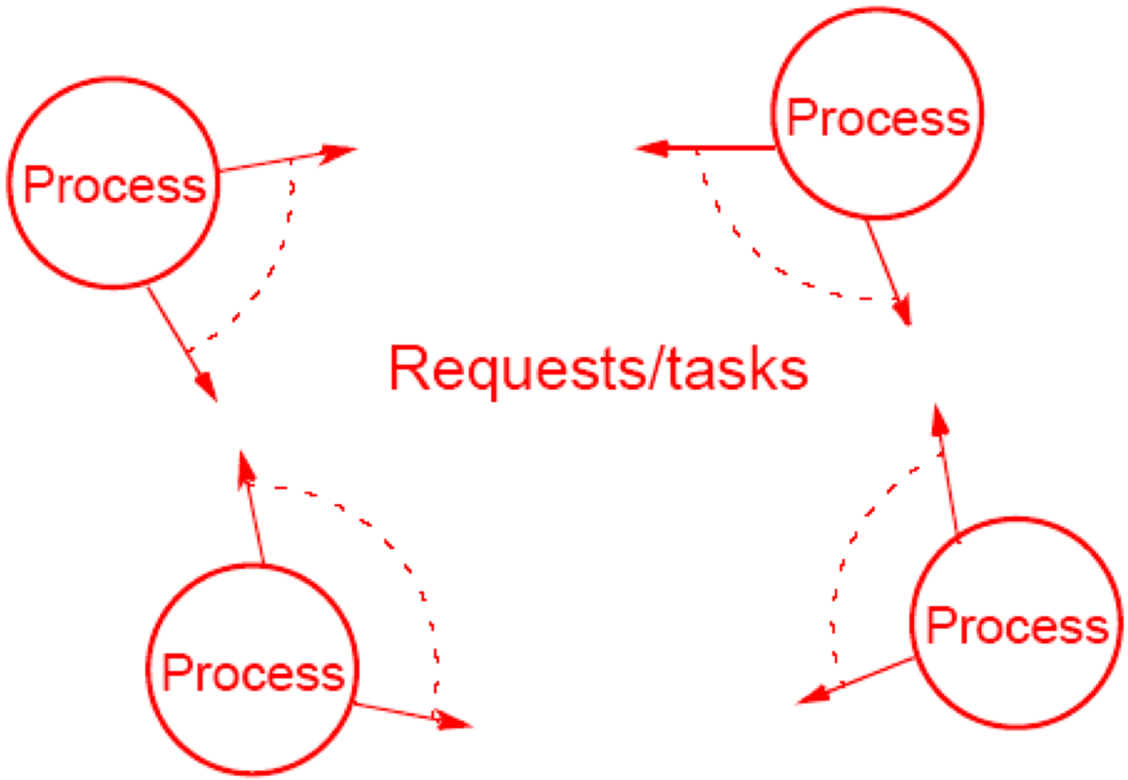


Decentralized Task Pool

- The task pool is distributed among several processes, e.g.:
  - Hierarchy of "submasters," each with separate worker sets, or
  - "Dual-mode" workers, acting as both submaster and worker
- Results all go back to the original master
- Termination: involves both local conditions (nothing to do right now) and global conditions (to guarantee that no additional tasks will show up)
- For the case of "dual-mode" workers:
  - Workers process tasks and possibly create new tasks
  - Workers pass tasks around among themselves to balance the load:
    - May "shed" (get rid of) tasks when they're too busy
    - May request tasks when they're free or don't have many tasks queued up (like a watermarking approach)

Task Transfer Mechanisms

Task pool is effectively "fully distributed" among the submasters or the dual-mode workers (which may include the master). The participants pass tasks among themselves to balance the load



Task Transfer Mechanisms

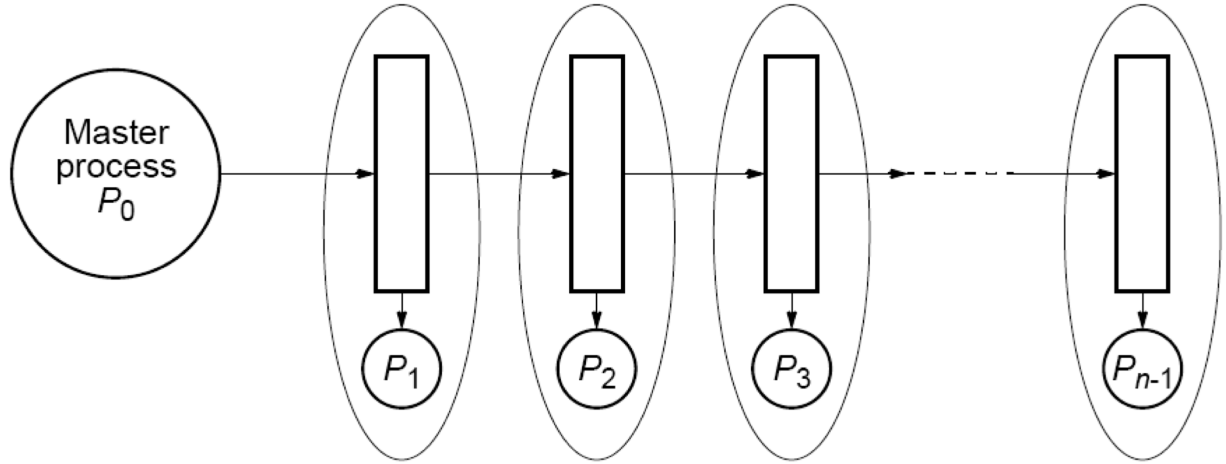
- Receiver-Initiated
  - When its local task queue nears empty, worker requests tasks from one or more other workers that it selects
  - May work well when there are overloaded workers, but it is difficult to determine which workers those are
- Sender-Initiated
  - Workers "shed" work when they get backed up (lots of tasks waiting in their local queues)
  - Generally works well if some workers have light loads, but, again, it may be difficult to determine which ones those are.
- Practical implementations may combine the above and may use a "load-independent task" movement algorithm
- Some systems use a "bulletin board" model−possibly hosted by the master (e.g. "Linda" or database systems, where all tasks are visible via query)

Worker Selection for Task Transfer

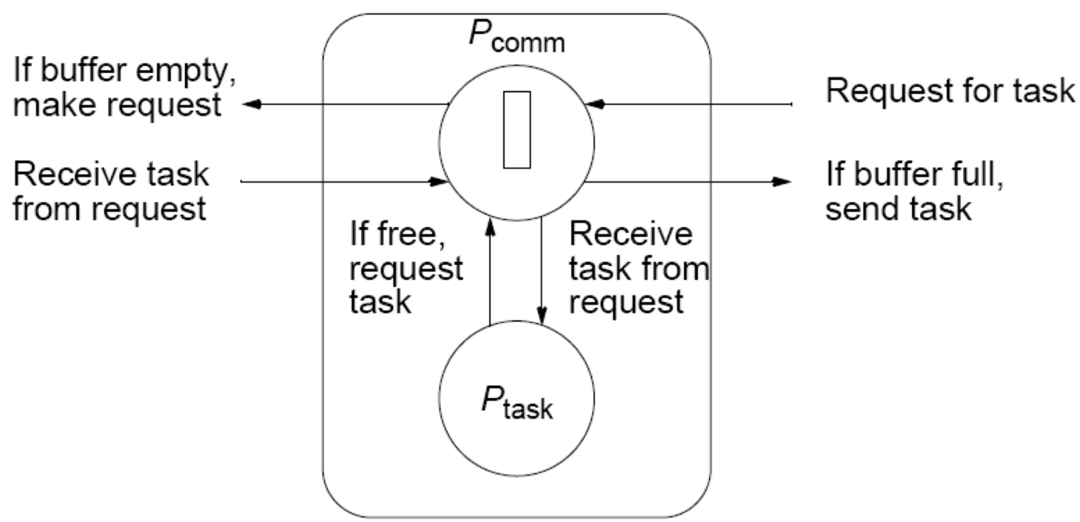
- Round Robin
  - Each worker transfers tasks to or from a process it selects using a counter that rotates around all the workers
    - Initially: *counter* for p_i is p_{i+1} (mod *nw*) for *nw* workers
    - After an attempt to transfer: *counter* = *counter+1* (mod *nw*), leaving out the process itself
- Random Polling
  - Worker p_i selects worker p_x where x is an integer in [*0*, *nw-1*], excluding i
- Fixed Topology Task Shifting
  - Workers receive tasks from specific worker(s) and pass them on until they reach lightly-loaded worker(s)

Line Structure Task Shifting

Master feeds tasks to p_1, and the tasks are then shifted down the line to idle workers.



Possible implementation might use two threads or two processes per worker (on a single cpu): one for communication about tasks, and one for real work



Distributed Termination Detection

- Conditions:
  - Application-specific local termination conditions on each worker
  - No task assignments pending or "in transit" (difficult to know)
- More general and safer approach:
  - Each worker has 2 states: "*Inactive*" (ready to terminate) or "*Active.*" Worker starts *Inactive* and becomes *Active* when it first receives a task. Source of that task becomes the worker's "*Parent*"
  - Worker immediately acknowledges all tasks received, except its initial task, which is only acknowledged just before it becomes inactive
  - Once active, a worker stays active until it:
    - meets local termination conditions and has no pending tasks
    - has received all acknowledgements due for tasks it sent out
    - has acknowledged all tasks it received other than the first one
    - acknowledges initial task to its parent (last step before becoming inactive)
  - Termination occurs when master becomes inactive

Termination Using Acknowledgements