Overview of the PEBBL and PICO Projects: Massively Parallel Branch and Bound

Jonathan Eckstein
Business School and RUTCOR
Rutgers University

Joint work with a large team, mostly from Sandia National Laboratories, and in particular

William E. Hart
and
Cynthia A. Phillips

July, 2006

My work supported by SNL and NSF (CCR 9902092)

(New) Distinction between PEBBL and PICO

Specific applications

PICO -- *Parallel Integer and Combinatorial Optimization*
Specific to mixed integer programming

PEBBL -- *Parallel Enumeration and Branch and Bound Library*
Generic parallel branch and bound

Until summer 2006, PEBBL was part of PICO

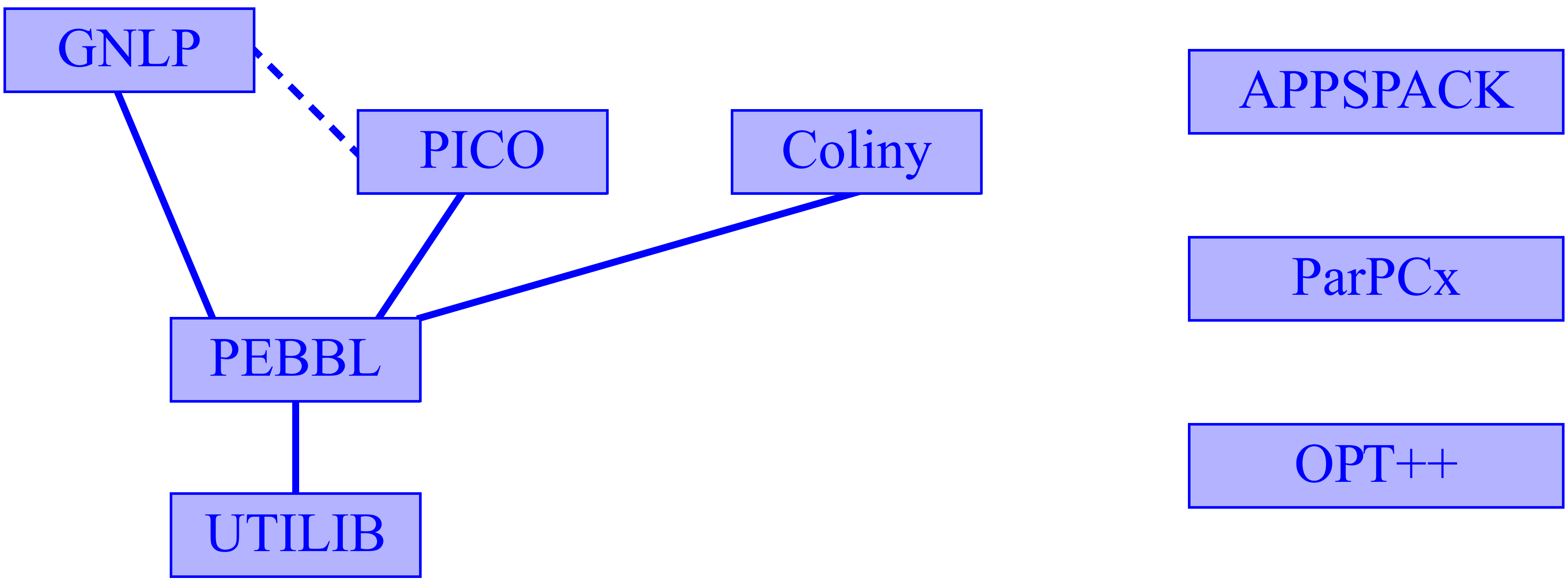
- PEBBL was called the “PICO core”
- What is now PICO was called the “PICO MIP”

PEBBL and PICO are part of ACRO

*A C*ommon *R*epository for *O*ptimizers

`http://software.sandia.gov/acro`

- Collection of open-source software arising from work at Sandia National Laboratories
- Generally lesser GNU public license



PEBBL/PICO Applications

Direct use of PEBBL

- Peptide-protein docking (quadratic semi-assignment)

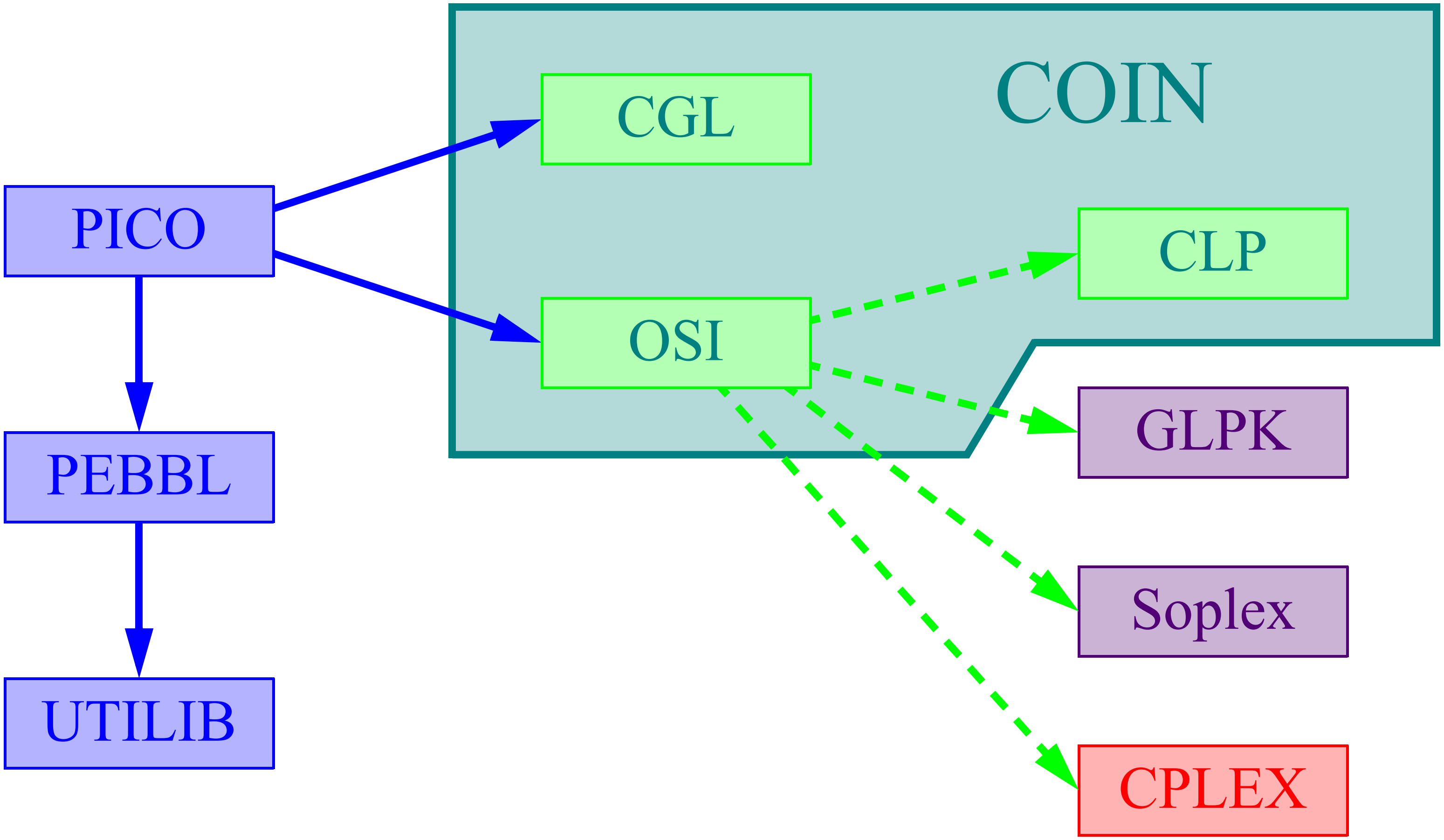
GNLP (includes PEBBL)

- PDE Mesh design
- Electronic package design

PICO (includes PEBBL)

- JSF inventory logistics
- Peptide-protein docking
- Transportation logistics
- Production planning
- Sensor placement
- ...

PEBBL/PICO Package Relationships



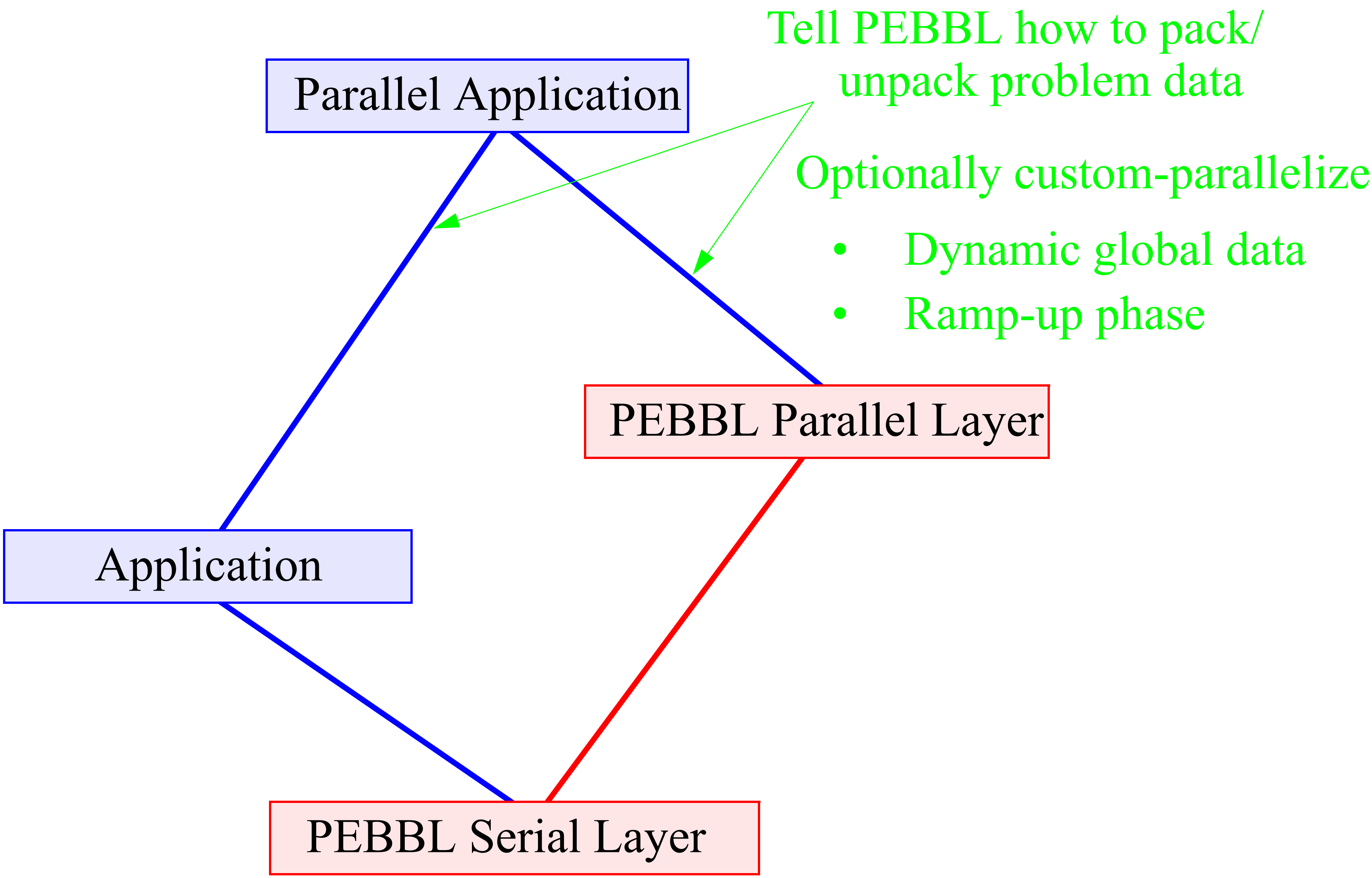
For remainder of talk, focus on PEBBL

PEBBL is a parallel "branch and bound shell"

Key features

- Object oriented design with serial and parallel layers
- Application interface via manipulation of problem states
- Variable search "protocols" as well as search orders
- Flexible, scalable parallel work distribution using processor clusters
- Non-preemptive thread scheduling on each processor
- Checkpointing
- (Enumeration support)
- Alternate parallelism support during ramp-up phase

Basic C++ Class Structure: Serial and Parallel Layers



PEBBL Structure: Serial and Parallel Layers
Application Development Sequence

Describe application to PEBBL

↓

Debug in serial environment

↓

Tell PEBBL how to pack and unpack problem/subproblem messages

↓

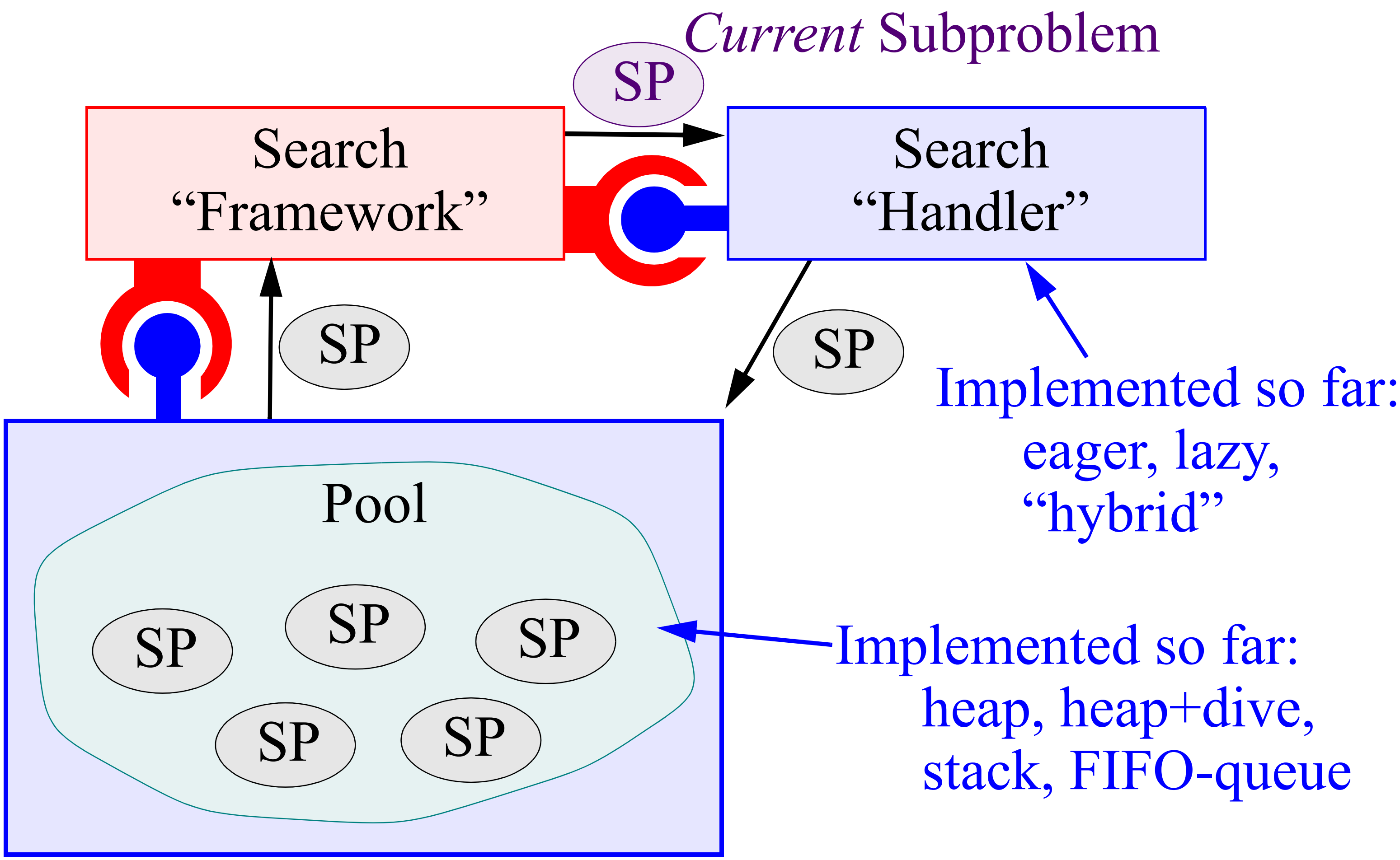
Run in parallel environment without additional programming effort

↓ (optional)

Enhance default parallelization: global information, ramp-up, etc.

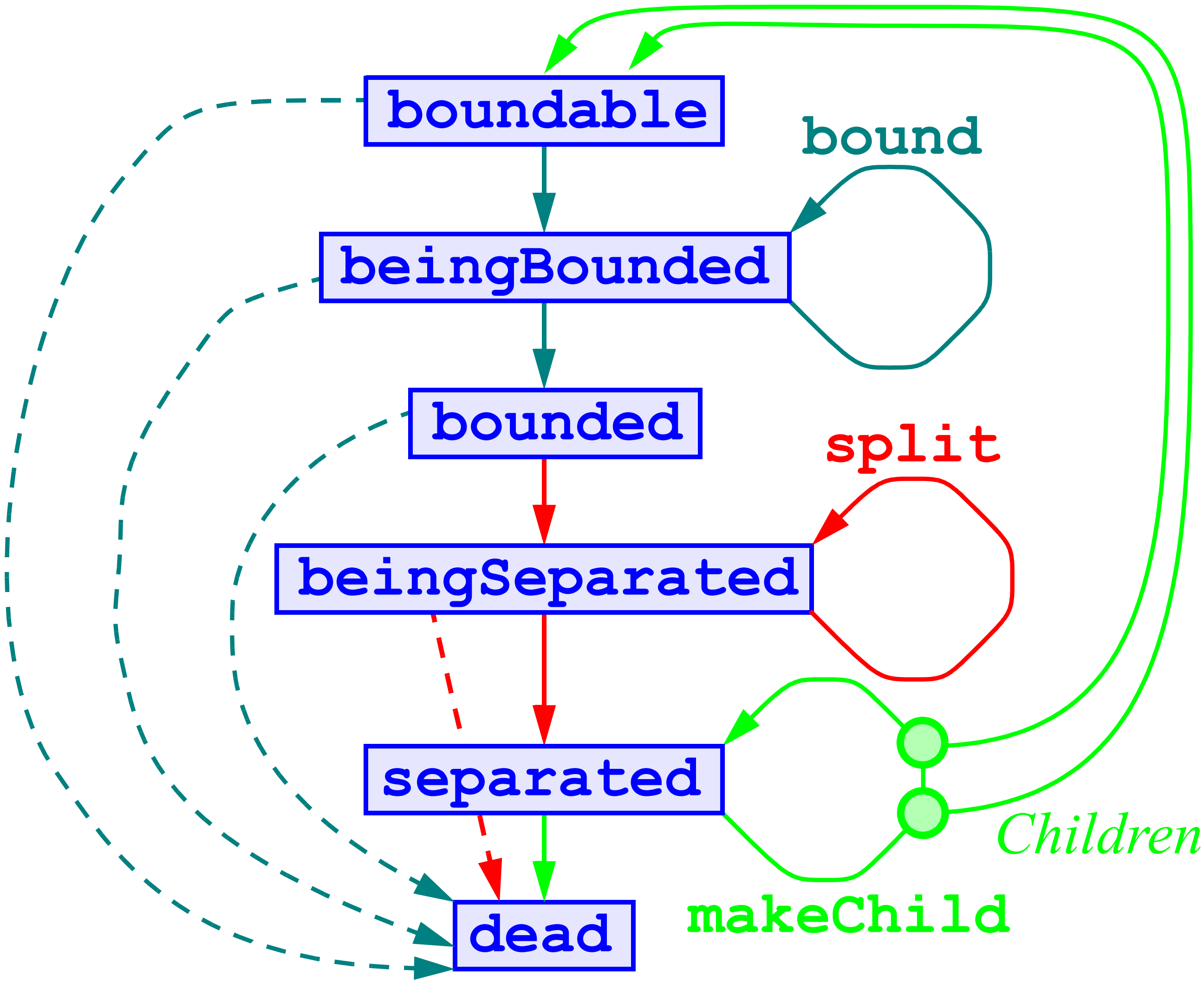
PEBBL Serial Layer Design

- Class derived from **`branching`** holds data global to problem.
- Class derived from **`branchSub`** holds subproblem data and pointer back to global data (as in ABACUS).



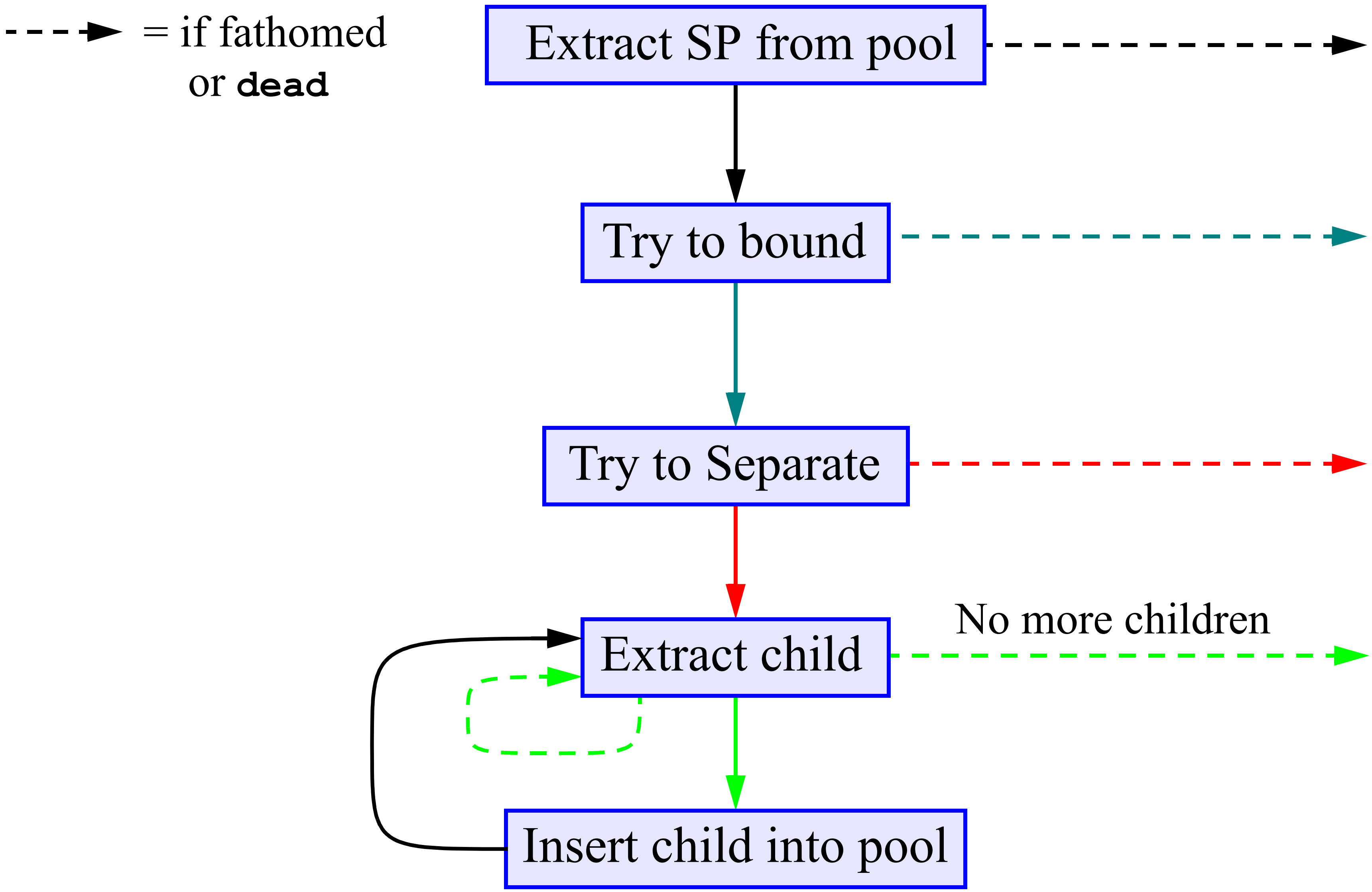
- Key point: problems in the pool remember their *state*.

Standard Subproblem State Sequence



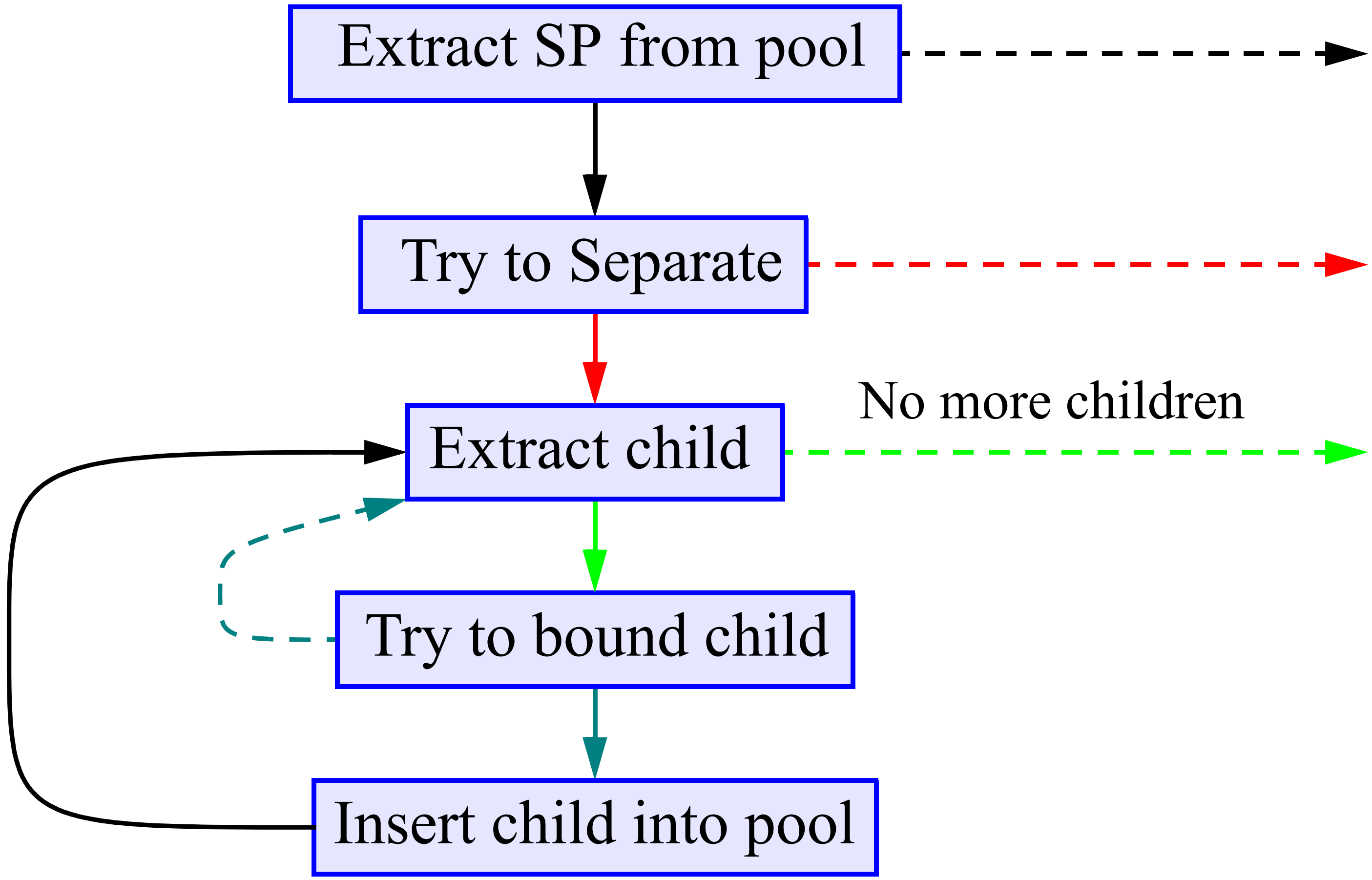
PEBBL interacts with the application solely through virtual functions that cause state transitions ( /  / )

Search Handler: Lazy



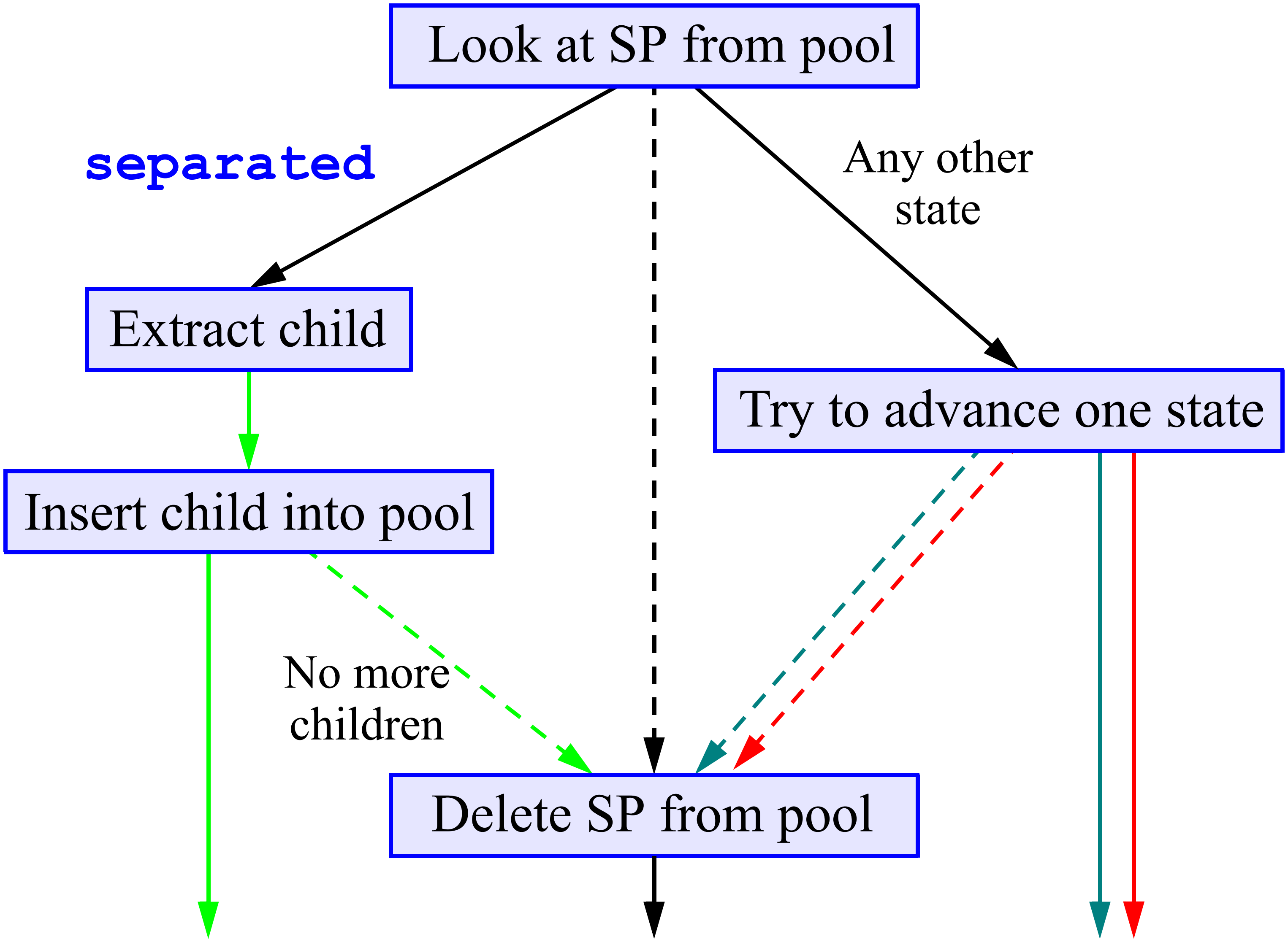
Pool consists of **`boundable`** subproblems

Search Handler: Eager



Pool consists of **`bounded`** subproblems

Search Handler: “Hybrid”/General



Pool can contain problems in any mix of states.

Generality of Approach

Naturally accommodates an wide range of branch-and-bound algorithm variations

Most known variations are possible by combining

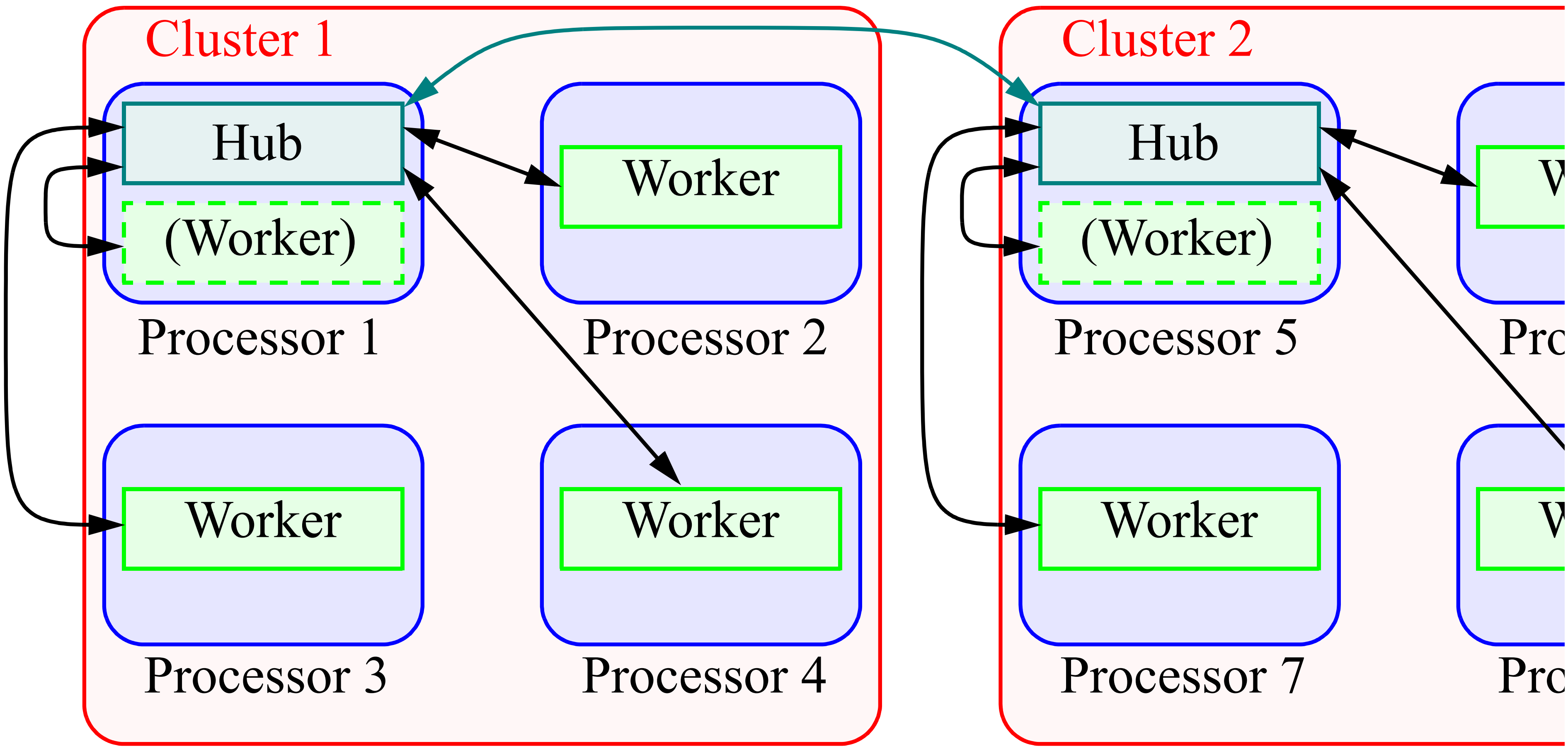
- Three existing handlers
- Stack and heap pools
- Proper implementation of virtual functions for application

Also:

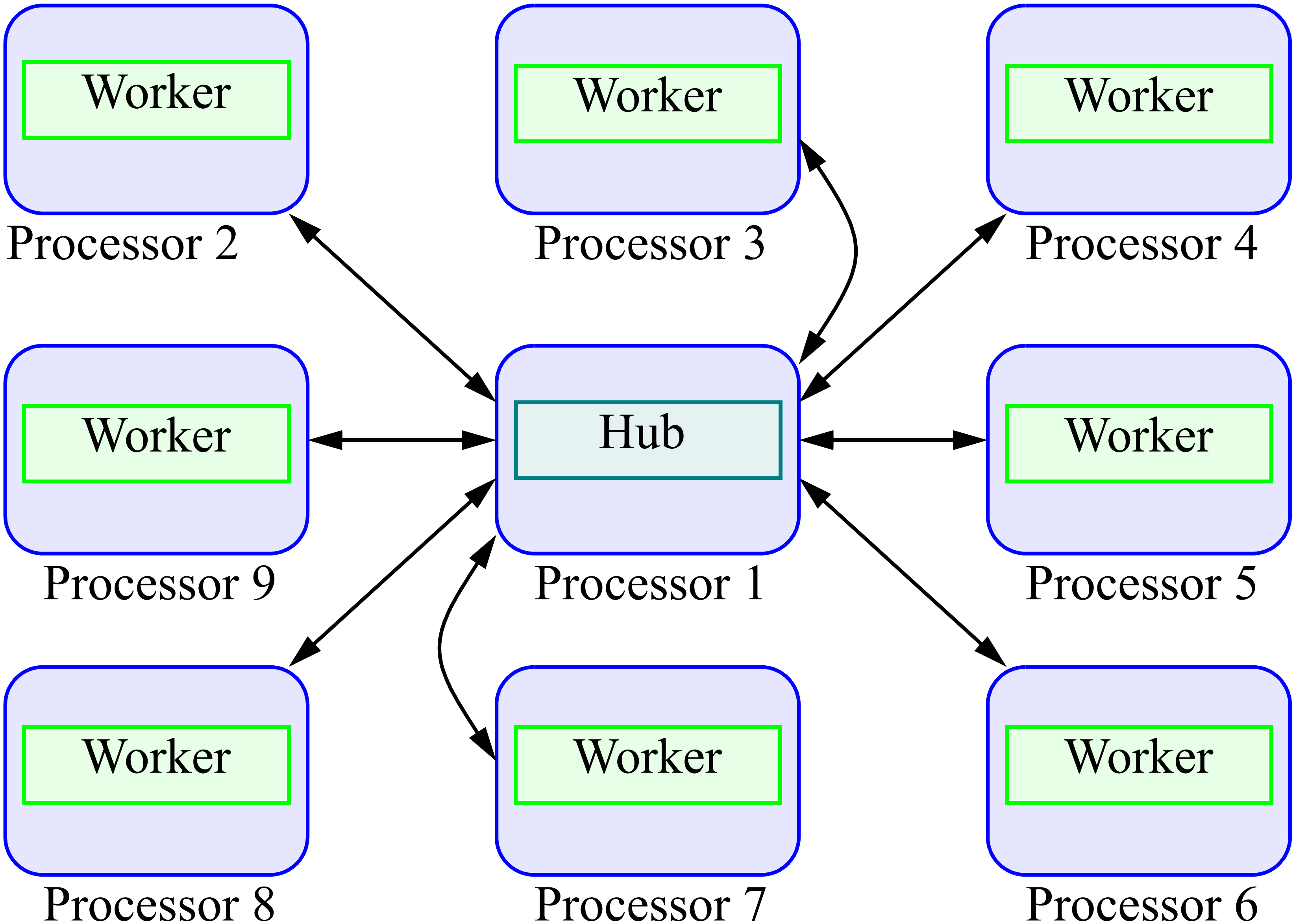
- Other pool implementations are possible
- Other handlers possible

Parallel Layer: User-Adjustable Clustering Strategy

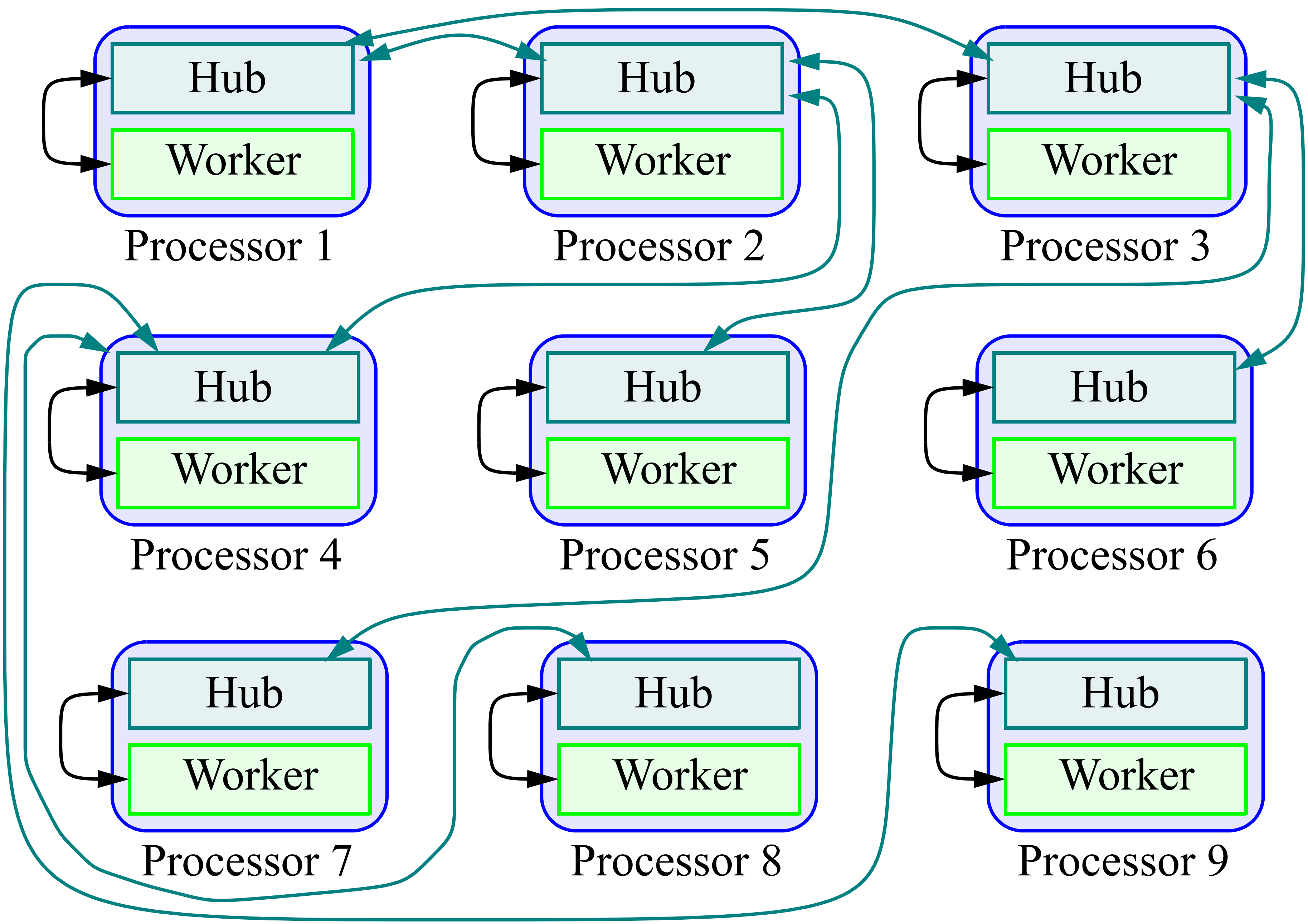
- Processors are collected into *clusters*
- One processor in the cluster is a *hub* (central controller for cluster)
- Other processors are *workers* (process subproblems)
- Optionally, a hub can be a worker too (depends on cluster size)



Extreme Case: Central Control



Extreme Case: Fully Decentralized Control



Work Transmission: Within a Cluster

Hub processes deal with *tokens* only. A token =

- # of creating processor
- Pointer to creating processor's memory
- Serial number
- Bound
- (Any other information needed in work scheduling decisions)

Prevents irrelevant information from

- Overloading memory at hubs
- Wasting communication bandwidth in and out of hubs

Remaining subproblem information sent directly between workers when necessary

Within a Cluster: Adjustable Behavior

Worker has its own local pool (buffer) of subproblems

Chance of returning a processed subproblem (or child) into the worker pool:

- 0% $\Rightarrow$ pure master-slave, hub makes all decision (fine for tightly-coupled hardware and time-consuming bounds).
- 100% $\Rightarrow$ hub "monitors" workers but doesn't make low-level decisions (better for workstation farms).
- Continuum of choices in between...

Backup "rebalancing" mechanism to make sure that hub controls enough subproblems

- Otherwise hub might be "powerless" in some situations
- Rebalancing uncommon for standard parameter settings

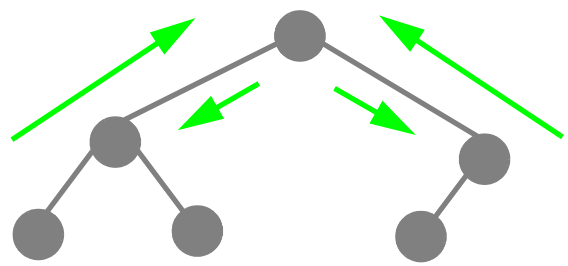
Work Transmission: Between Clusters

Load balancing between clusters via

- Random scattering upon subproblem creation, supplemented by...

Rendezvous load balancing:

- Non-hierarchical: there is no “hub-of-hubs” or “master-of-masters”
- Hubs are organized into a tree



- Periodic message sweeps up and down tree summarize overall load balance situation
- Efficient method for matching underloaded and overloaded clusters, followed by pairwise work exchange
- *Not* “work stealing” (receiver initiated)
- *Not* “work sharing” (sender initiated)

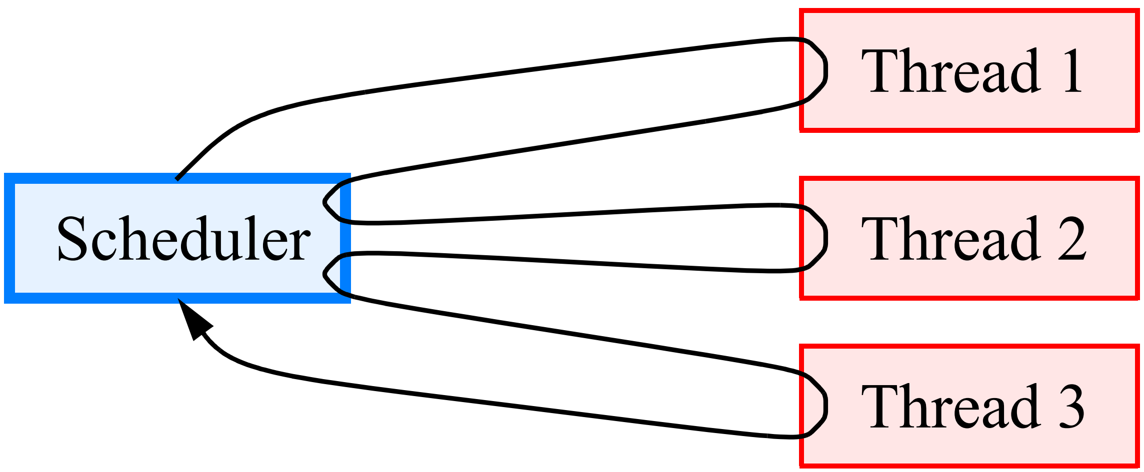
Non-Preemptive Threads on Each Processor

Each processor must do a certain amount of multi-tasking

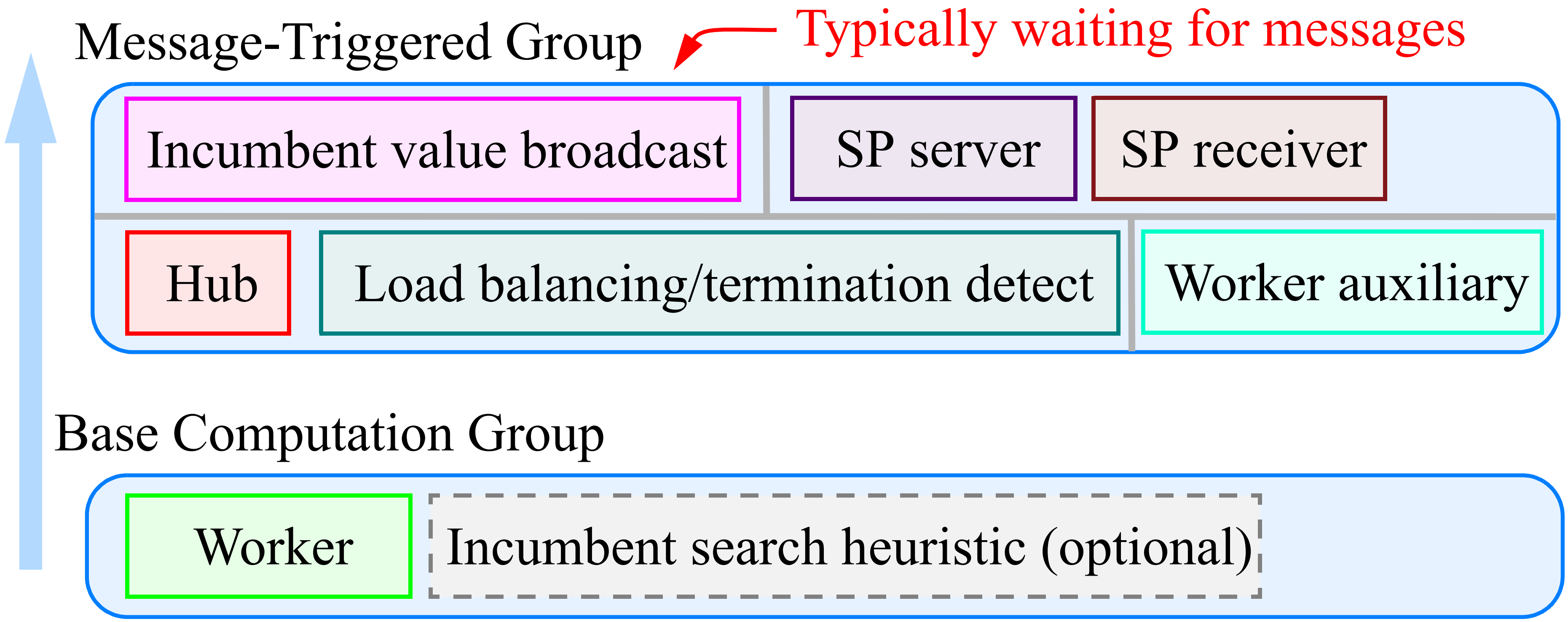
Schedule multiple *threads* of control within each processor

- Each task gets a thread.
- Threads can share memory.
- We use a *scheduler* to allocate CPU time to threads.

Scheduler uses non-preemptive multitasking approach (*à la* old Macs, Win 3.*x*):



Base Scheduler Setup



- Upper group: each thread waits for a specific kind of message
  - Wakes up; processes message; posts another receive request; sleeps again
- Base group: usually ready to run
  - ***Worker*** does work usually handled by serial layer
  - Continuously adjusts amount of work at each invocation to try to match a target ***time slice***
  - CPU time allocated in specifiable proportion via ***stride scheduling***

Incumbent Search Thread

Implements application-specific search heuristic; could be:

- Tabu
- GA
- etc...

Can send messages to other processors

- *e.g.* a parallel GA

Has small quantum for easy interruption

Soaks up cycles when worker thread is blocked or waiting

Can adjust priority as run proceeds

- High early on
- Lower later when we’re probably just proving (near) optimality of current incumbent

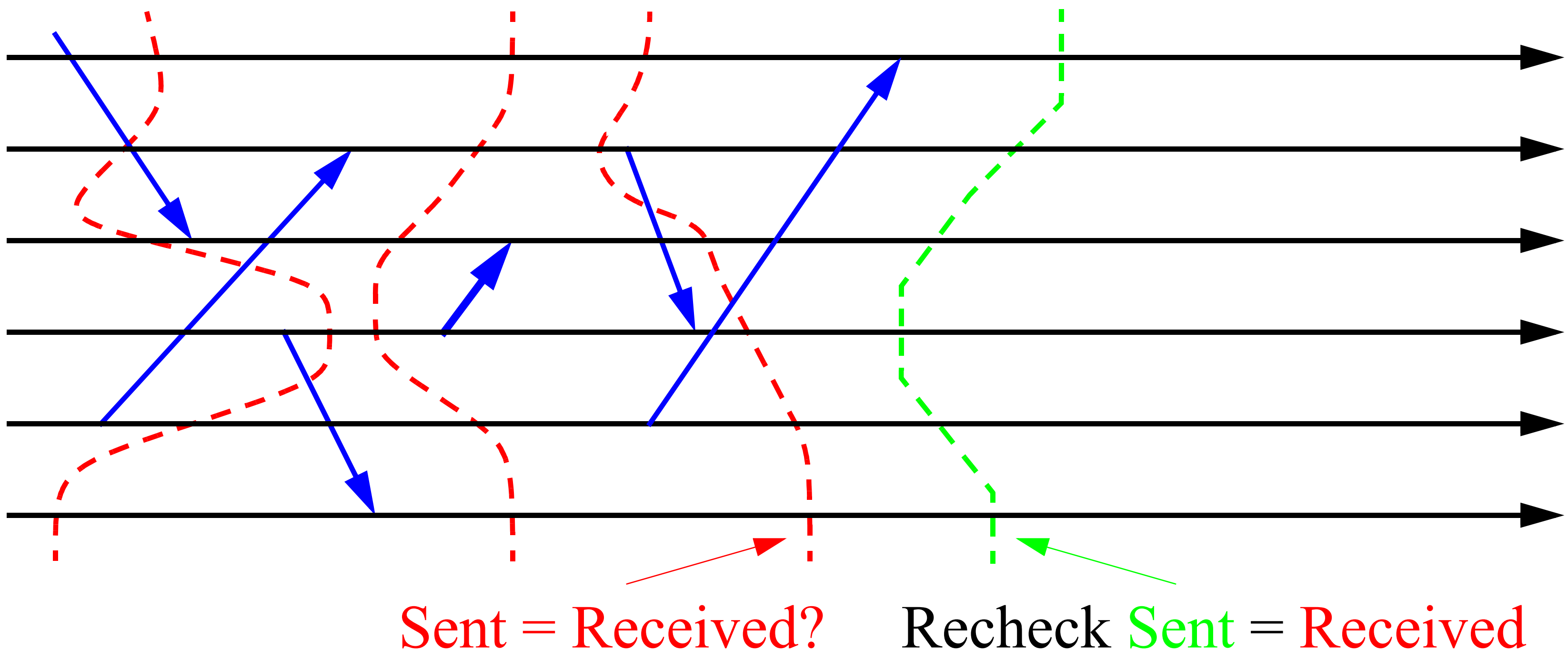
Framework allows smooth blending of parallel search heuristics with branch-and-bound.

General issue with asynchronous message-passing programs.

Make sure:

- All the work is really gone
- There are no stray unreceived messages floating around

PICO uses the “four counters” method of Mattern *et. al*



Handled by load balancing thread

Checkpointing (Relatively New)

- Systems crash
- Jobs exceed time quotas, ...

Don't want to lose all your work when that happens!

- Periodically save state of computation
- Later, you can restart from the saved state

Implementation in PEBBL:

- Load balancer message sweep signals it's time to checkpoint
- Workers and hubs turn "quiet": don't start new communication
- Use standard termination check logic to sense when all messages have arrived
- Each processor writes a (possibly local) checkpoint file

Restart options

- Normal: each processor reads its own file (possibly in parallel)
- Read serially, redistribute -- allows different number of processors

Ramp-Up: Starting the Search

There may be multiple sources of parallelism in *any* branch and bound application (not just MIP):

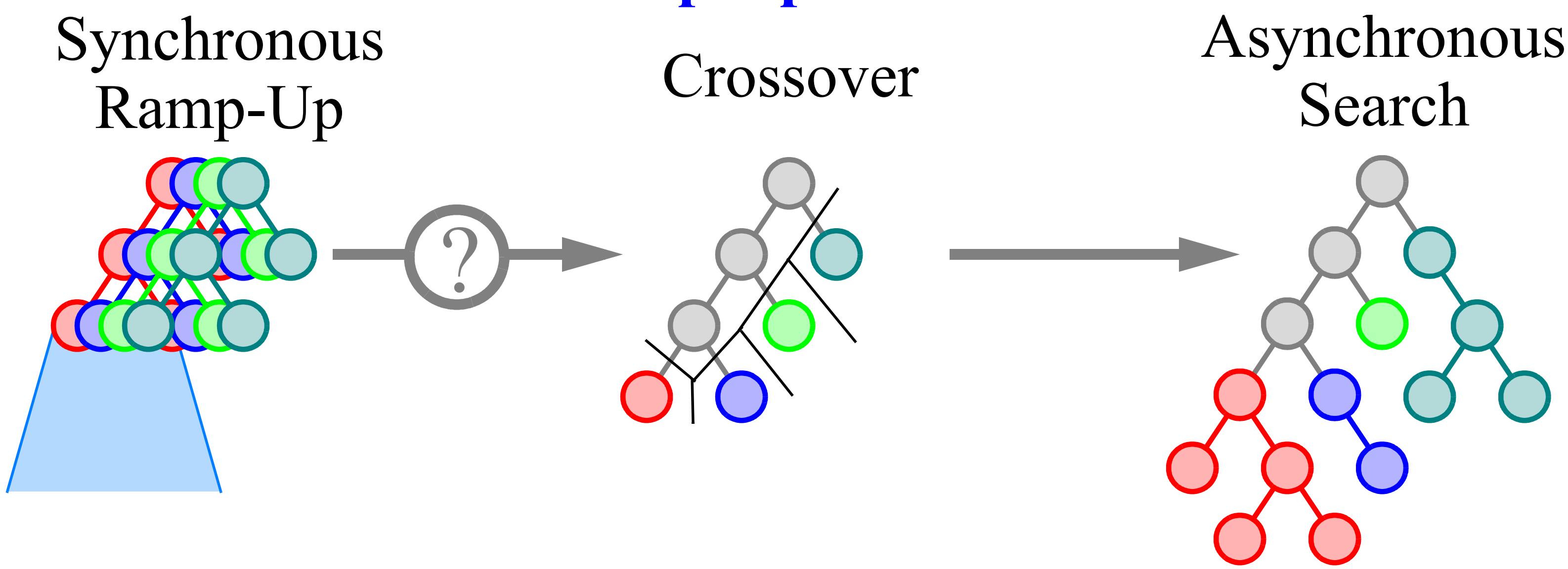
- Parallelism from large search tree (generic)
- Parallelism *within* each subproblem (application-specific)

Early in the search

- Tree is small
- Within-subproblem parallelism may be especially large
- So, there may be more parallelism available within subproblems than from the tree
- You also might not want to exploit tree parallelism too aggressively (likely to work on “non-critical” nodes)

Eventually, tree parallelism will probably dominate (and be safe)

Generic Ramp-Up Mechanism



- *Ramp-Up:* all processors redundantly develop top of tree, synchronously parallelizing some of each subproblem's work
- Virtual function decides when tree parallelism is likely to be better
- *Crossover:* partition tree evenly (no commucation!)
- Then start usual *asynchronous search* (different processors look at different leaves of the tree)
- PICO uses this feature: parallelizes strong-branching-like pseudocost initialization until tree offers more parallism

PEBBL and PICO Availability

ACRO 1.0 available first week of August, 2006

http://software.sandia.gov/acro

Lesser GNU public license

Includes PEBBL 1.0 release:

- Should be stable
- Contains 57-page user guide (will probably grow soon)
- Also, feel free to contact us if interested

PICO -- areas that need more work:

- Cut finders (improve/replace current CGL finders)
- Cut management
- Incumbent heuristic (fairly extensive work done, but more needed)