PICO: a Massively Parallel Branch-and-Bound Toolbox

Jonathan Eckstein
Faculty of Management and RUTCOR
Rutgers University

Joint work with
William E. Hart
and
Cynthia A. Phillips
(Sandia National Laboratories)

April 2001

PICO

*P*arallel *I*nteger and *C*ombinatorial *O*ptimizer (also sounds “Southwest”)

Goals

Non-shared memory:

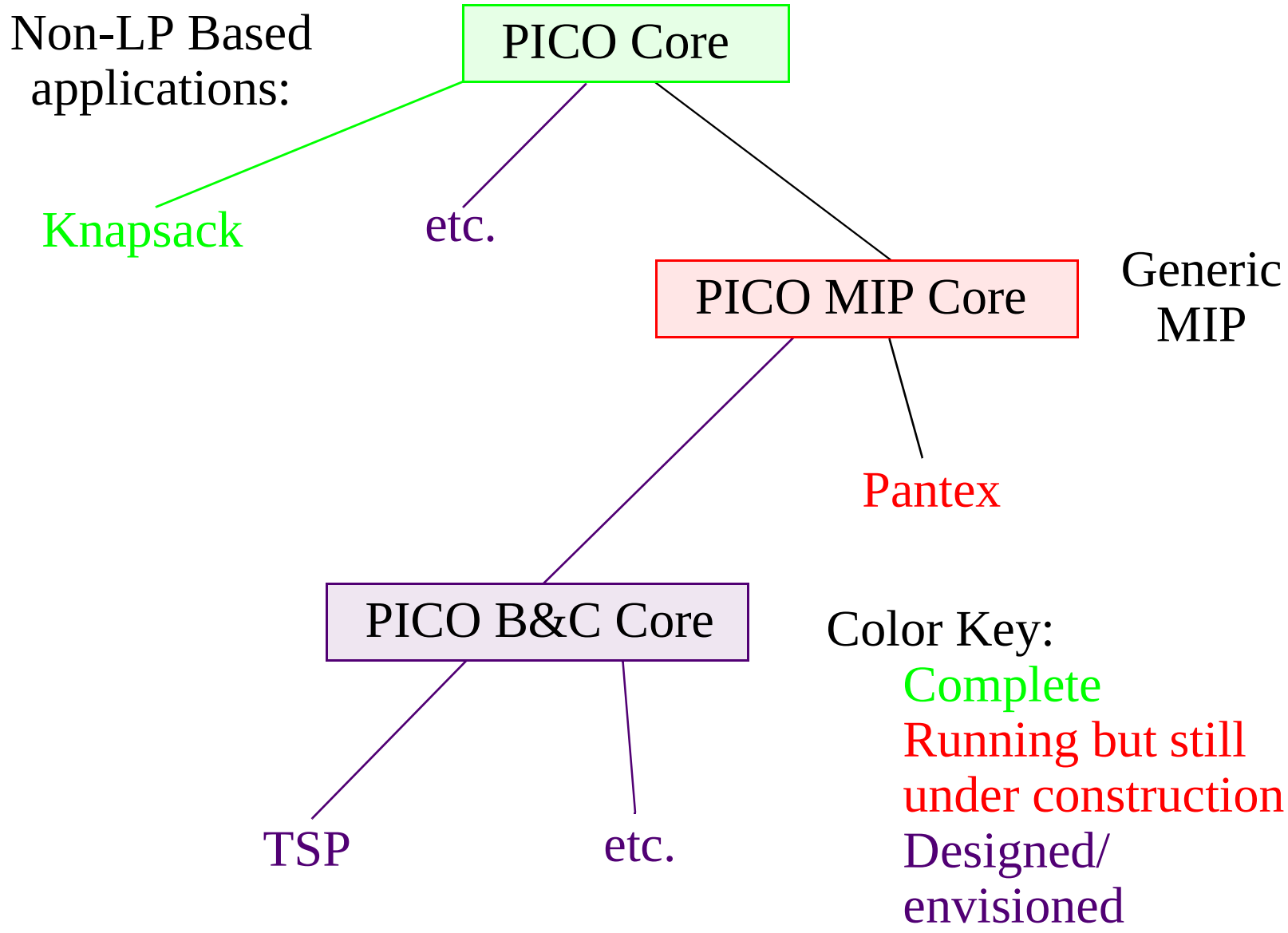
- Processors have separate memory and communicate via *messages*
- Required for scaling to large numbers of processors, can be emulated by other architectures

Build on experience of my earlier CM-5 MIP code (CMMIP)

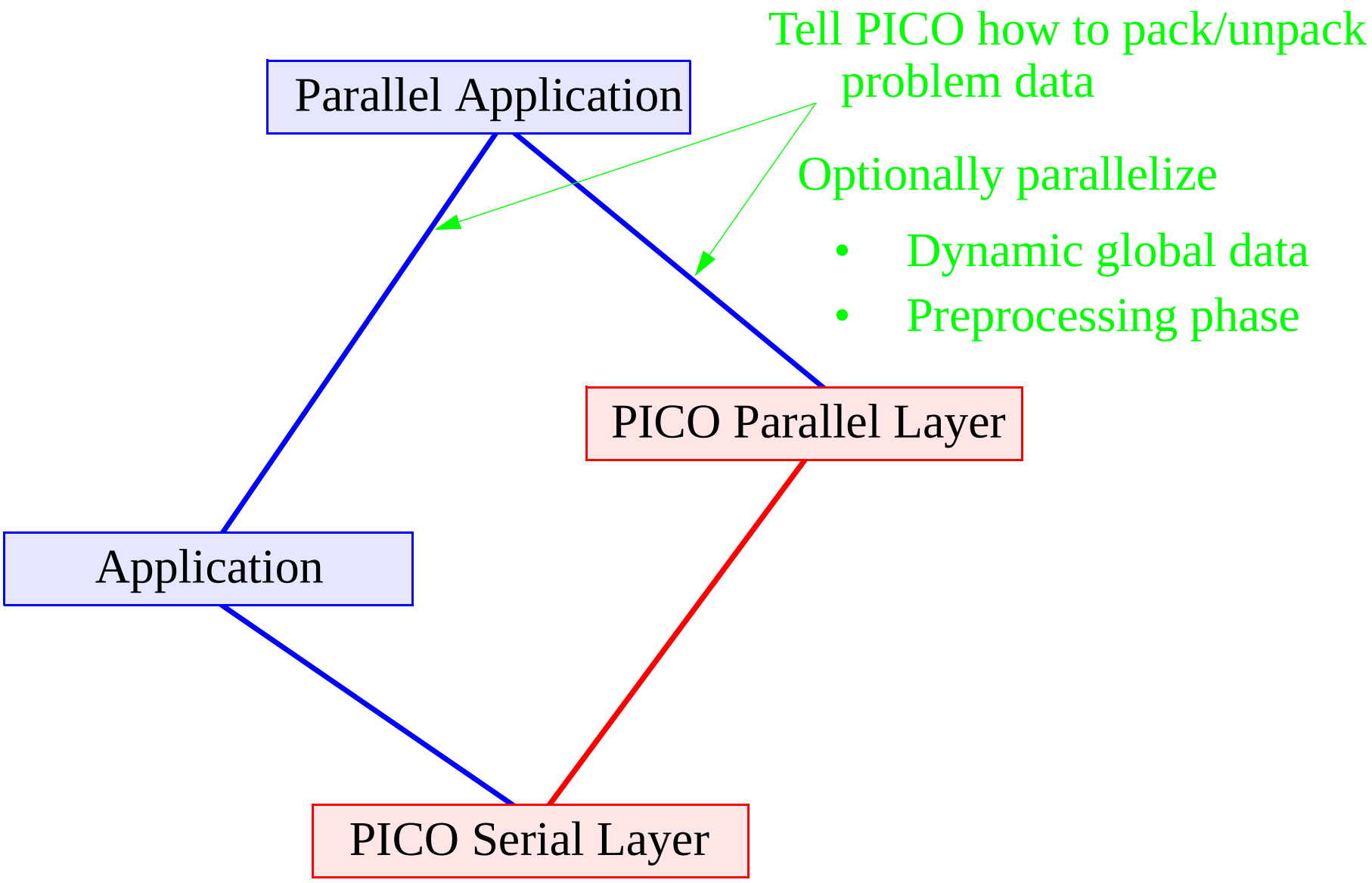
But be more *adaptable* to different

- Hardware/software environments
  - Uses standard MPI message passing library
  - Smoothly variable amount of communication
  - LP Solver (if any) “encapsulated”
- Applications
  - “Decouple” parallel search aspects from application
  - Use C++ multiple inheritance techniques

PICO Family Tree: Sketch of C++ Inheritance Structure

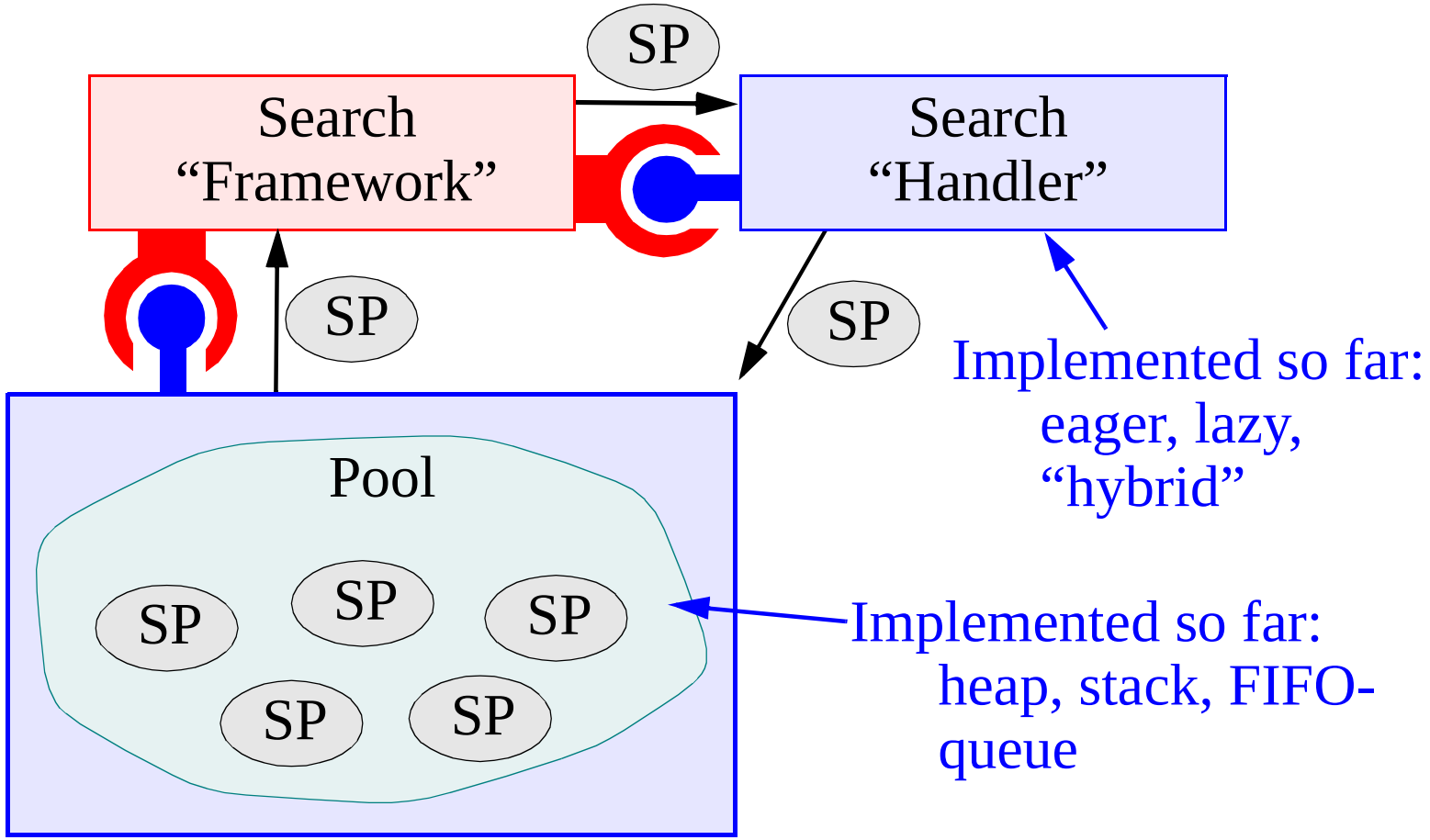


Basic C++ Class Structure: Serial and Parallel Layers



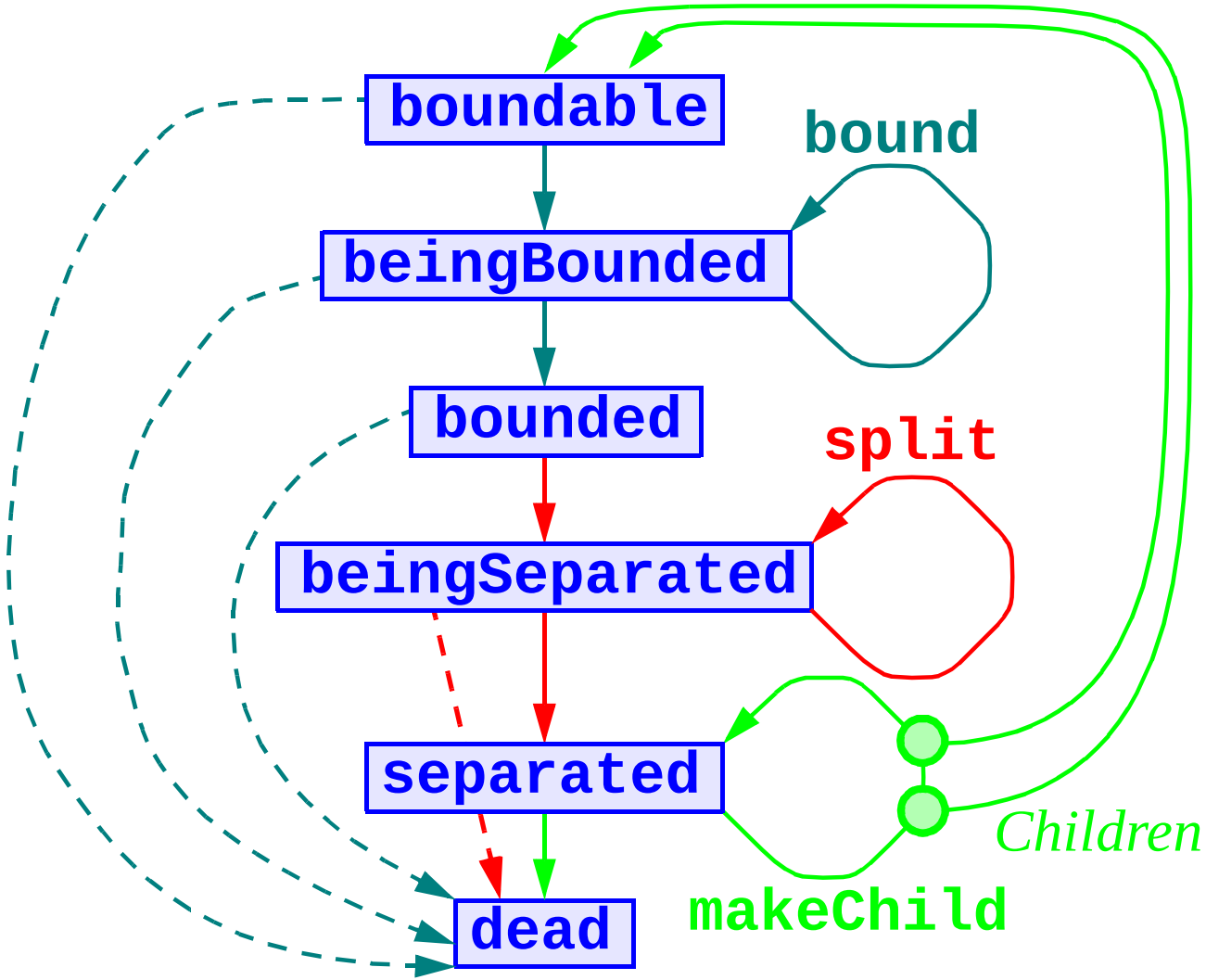
PICO Serial Layer Design

- 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).



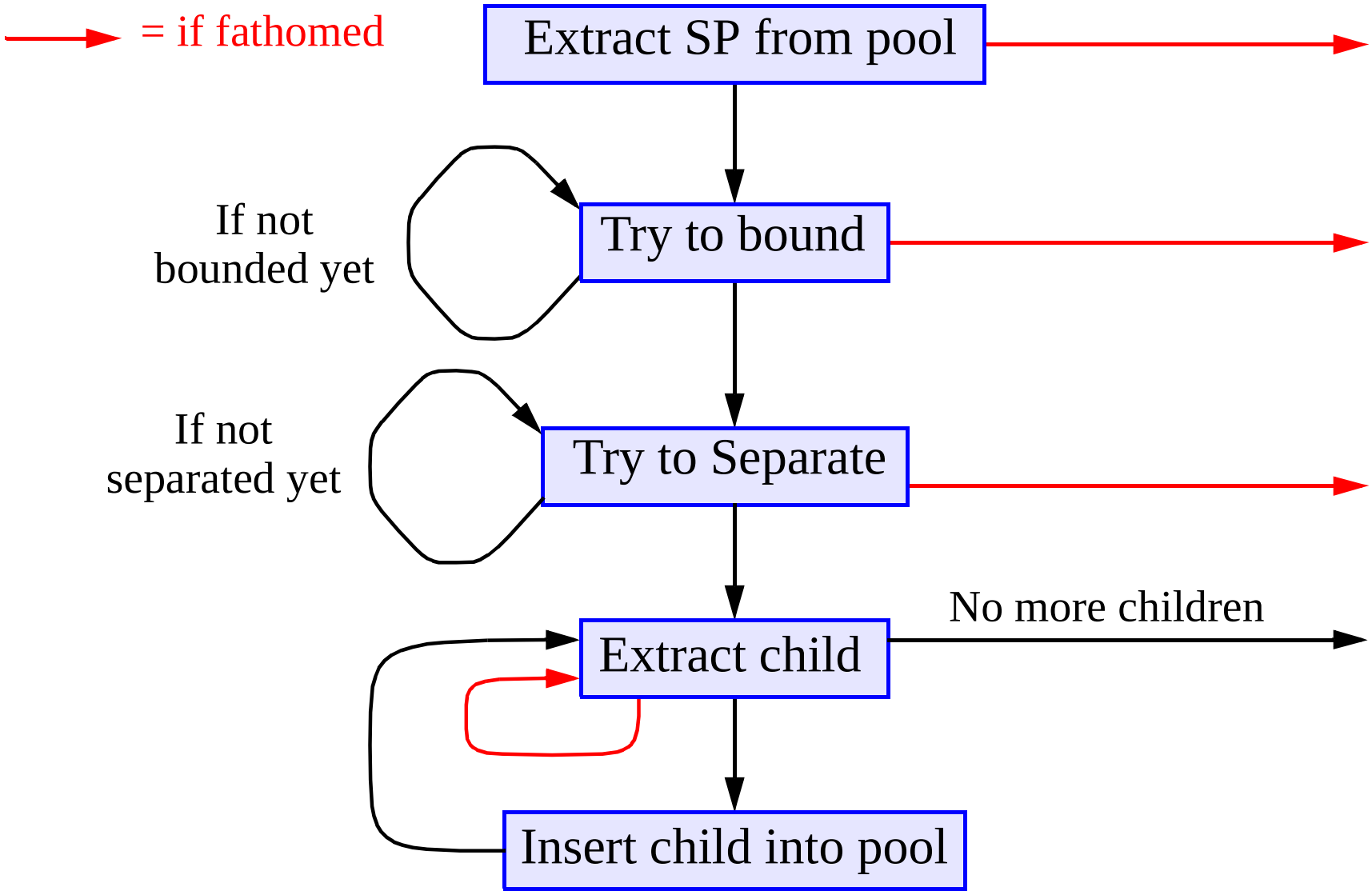
- Key point: problems in the pool can remember their *state*.

Standard Subproblem State Sequence



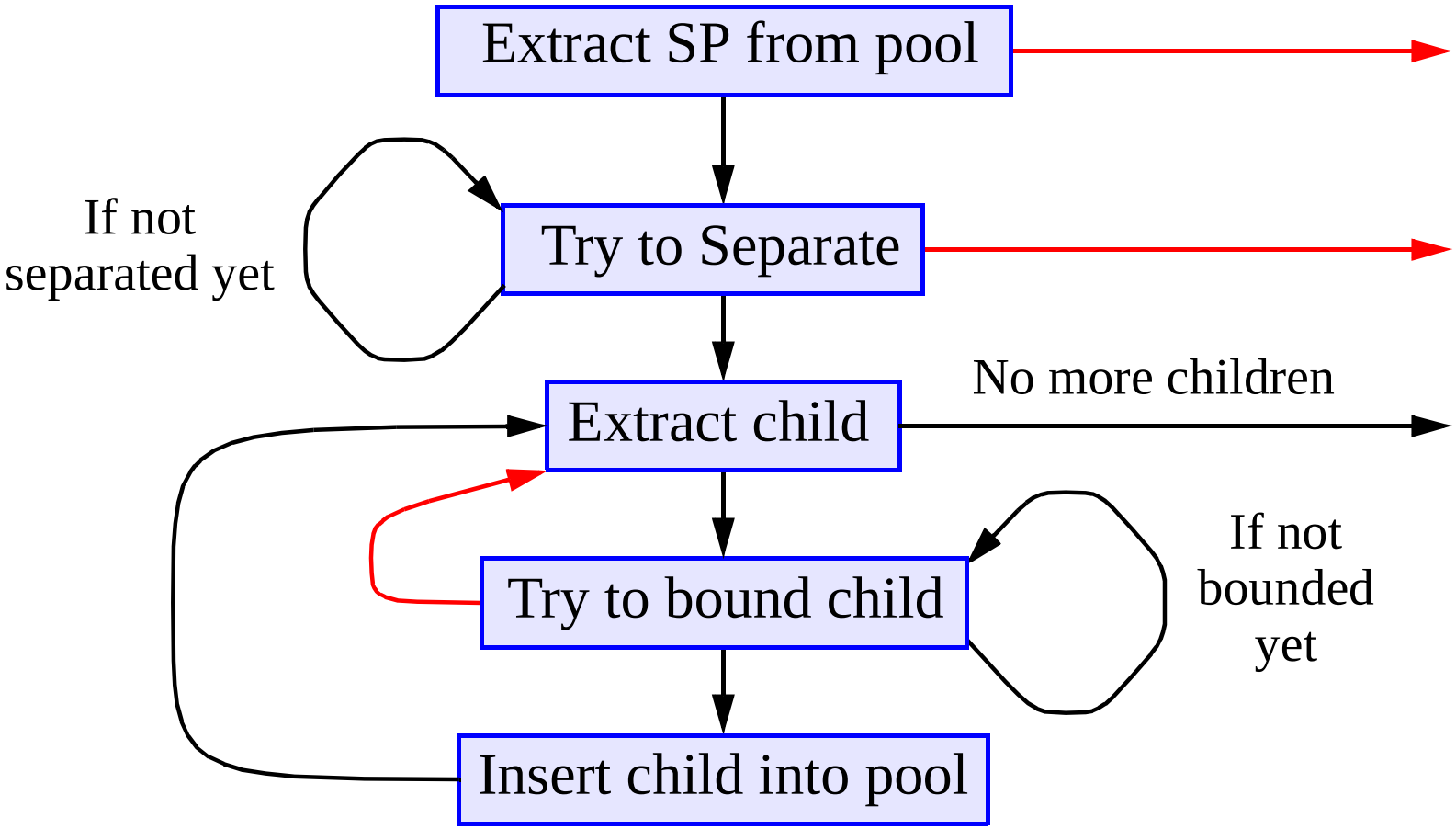
PICO interacts with the application solely through virtual functions that cause state transitions (→ / → / →)

Search Handler: Lazy



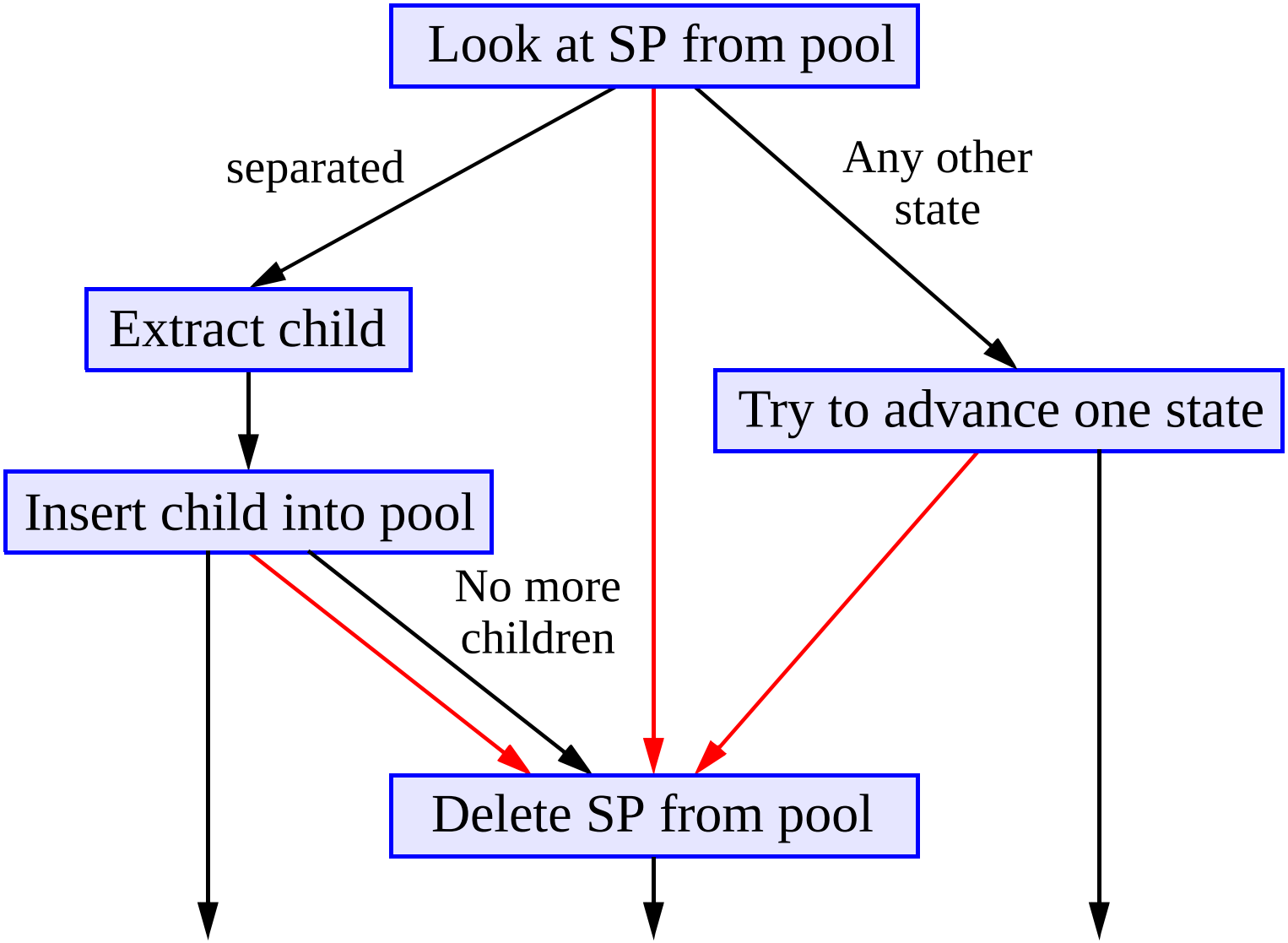
Pool consists of **`boundable`** subproblems

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 enormous range of branch-and-bound algorithm variations

Most known variations are possible by combining

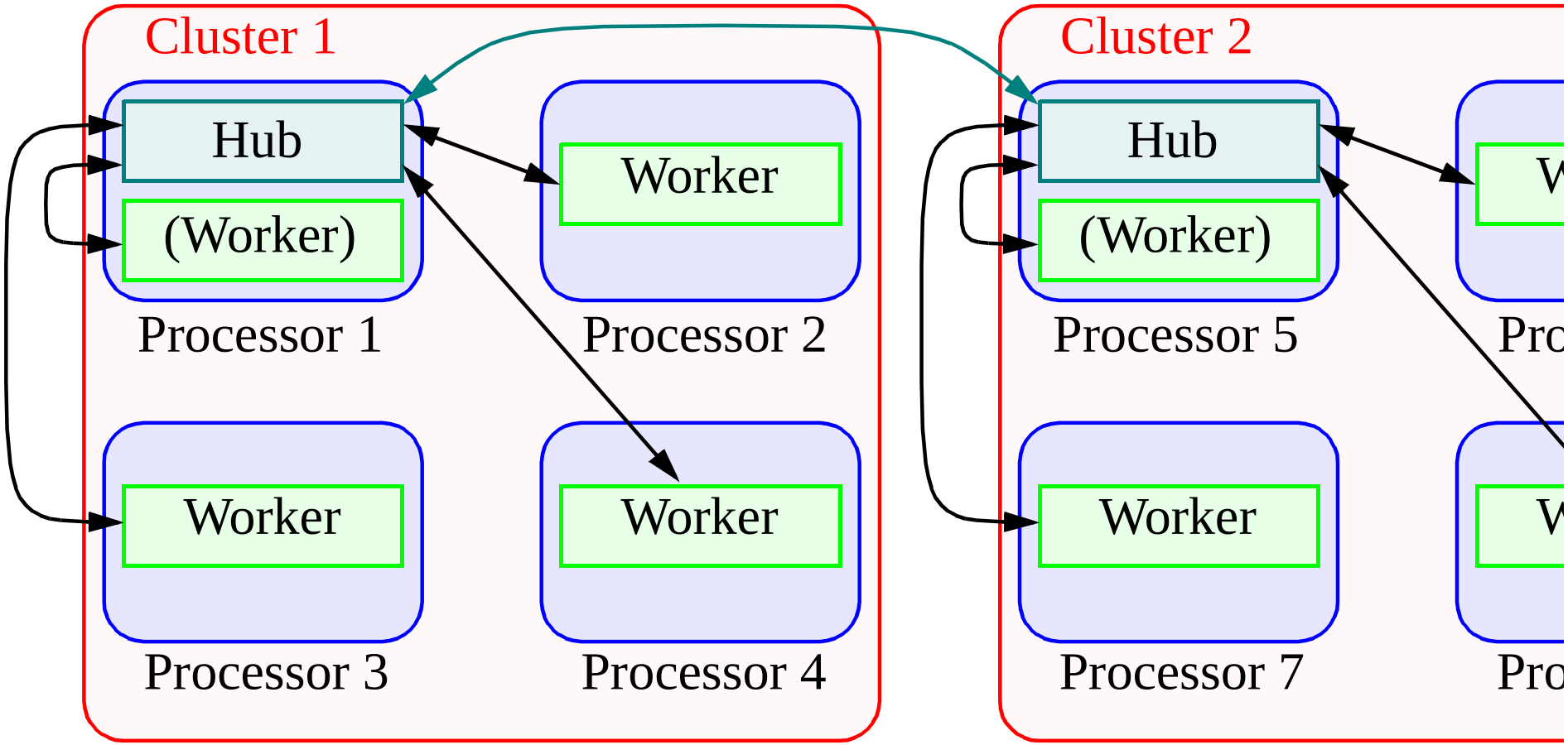
- Three existing handlers
- Stack and heap pools
- Proper implementation of virtual functions for application

Also:

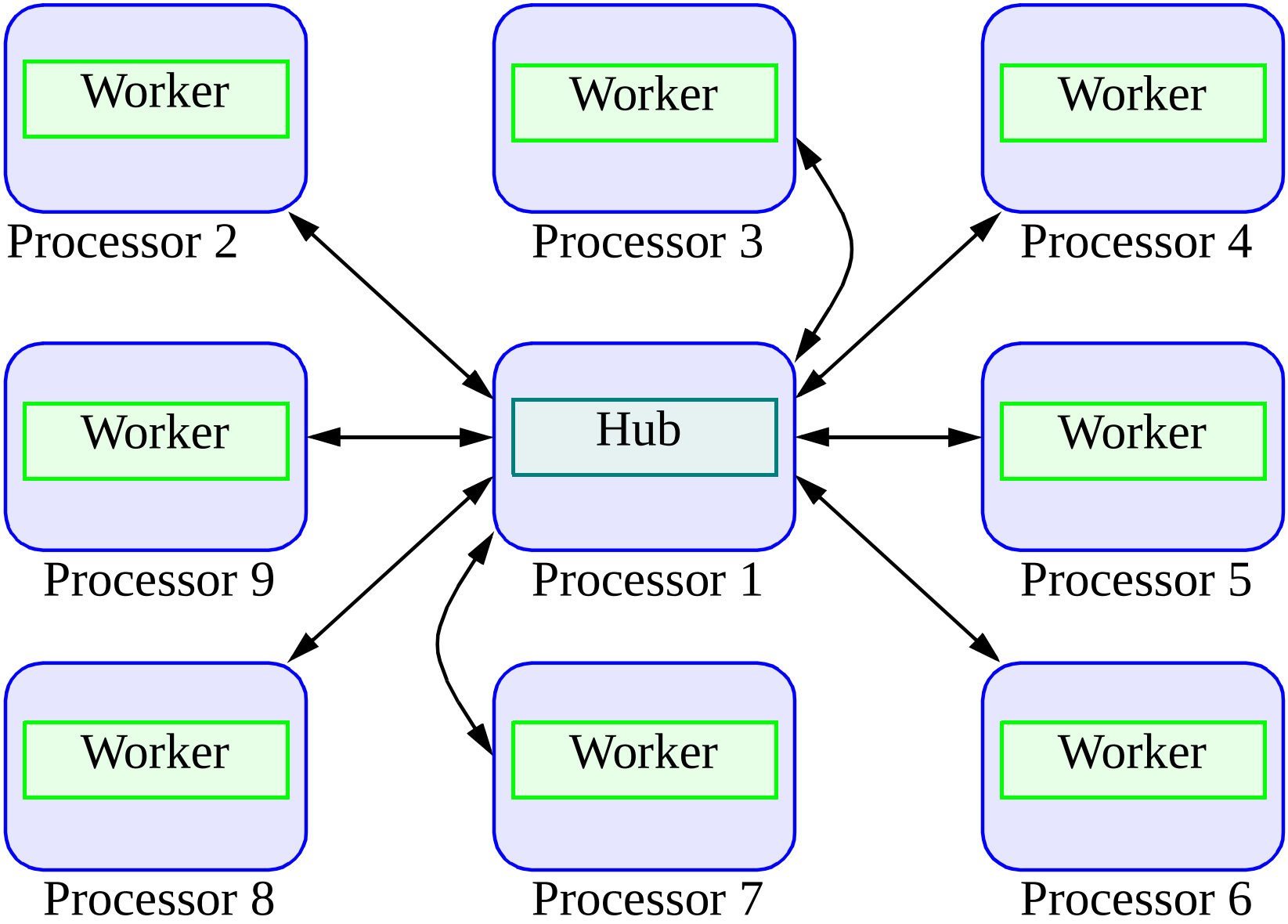
- Many other pool implementations are possible
- Many other valid handlers possible

Parallel Layer: User-Adjustable Clustering Strategy

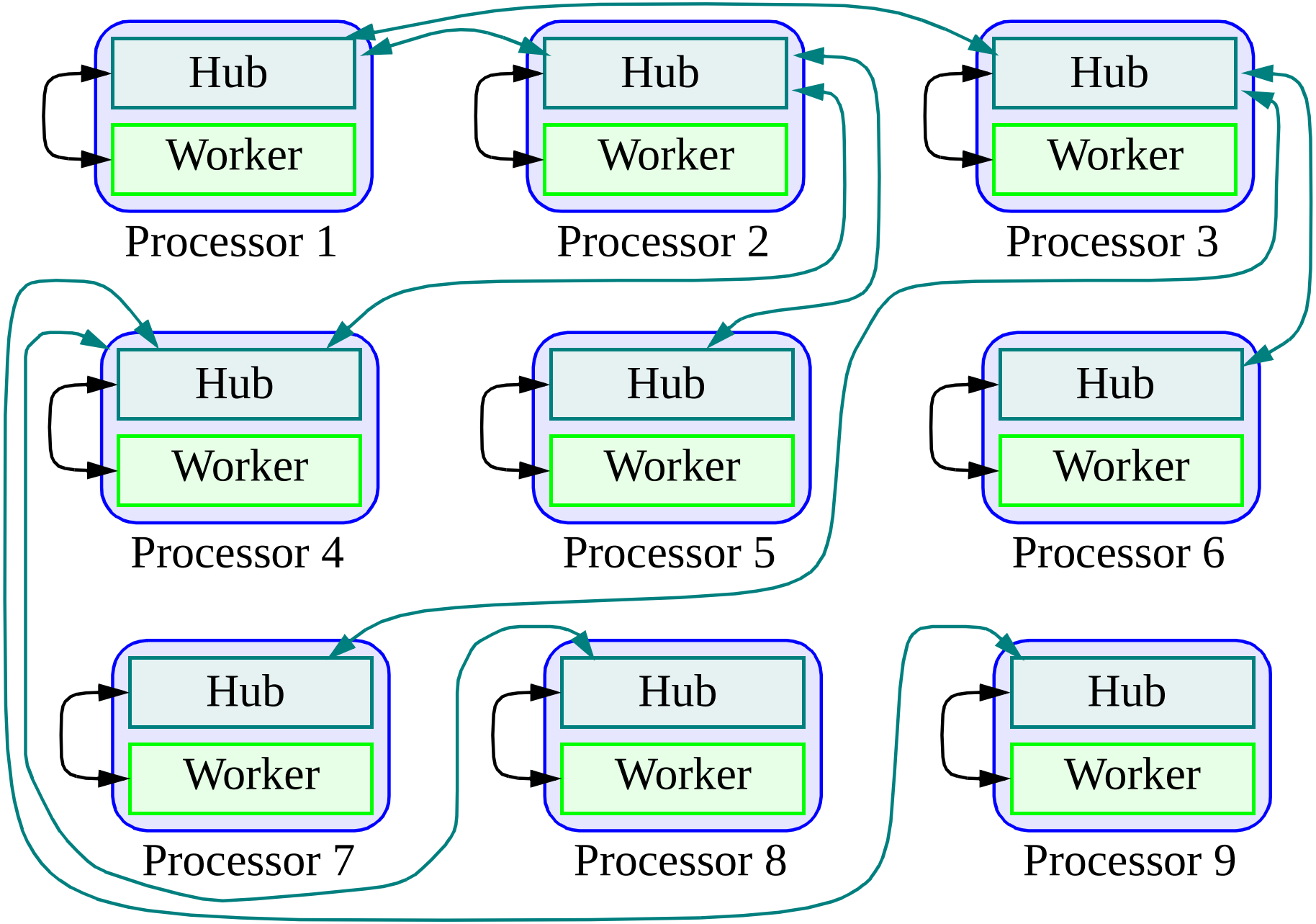
- Processors are collected into *clusters*
- One processor in the cluster is a *hub* (central controller for cluster)
- Other processors are *workers* (process subproblems)
- Optionally, a hub can be a worker too (depends on cluster size)



Extreme Case: Central Control



Extreme Case: Fully Decentralized Control



Work Transmission: Within a Cluster

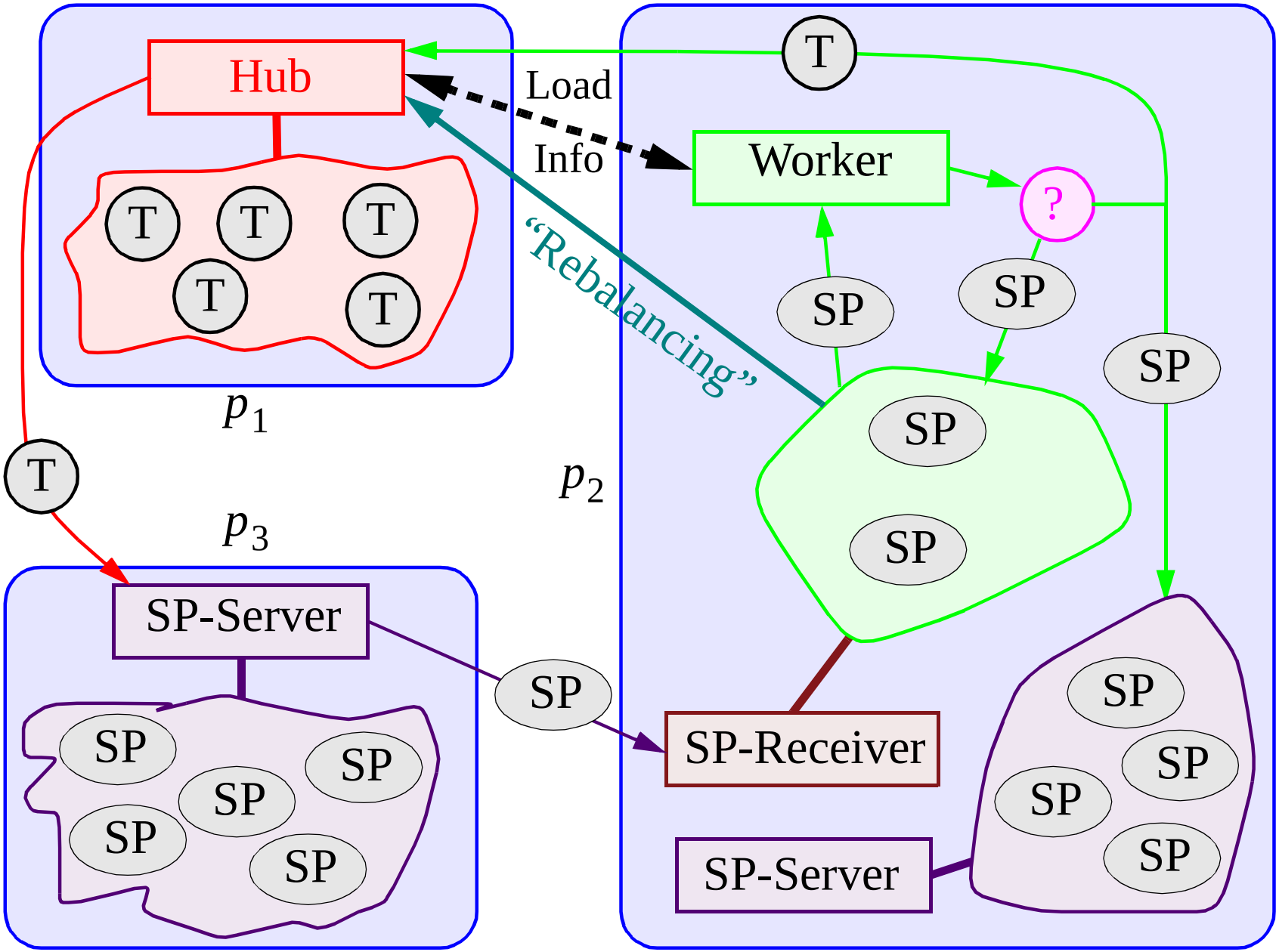
As in CMMIP, hub processes deal with *tokens* only. A token =

- # 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

Hub-Worker Interaction: Subproblem/Token Flow



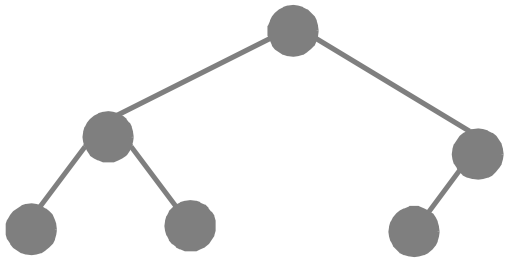
Work Transmission: Between Clusters

Load balancing between clusters via

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

Rendezvous:

- Previously considered a “SIMD” algorithm. “Global perspective...”
- Organize the H cluster hubs into a binary tree



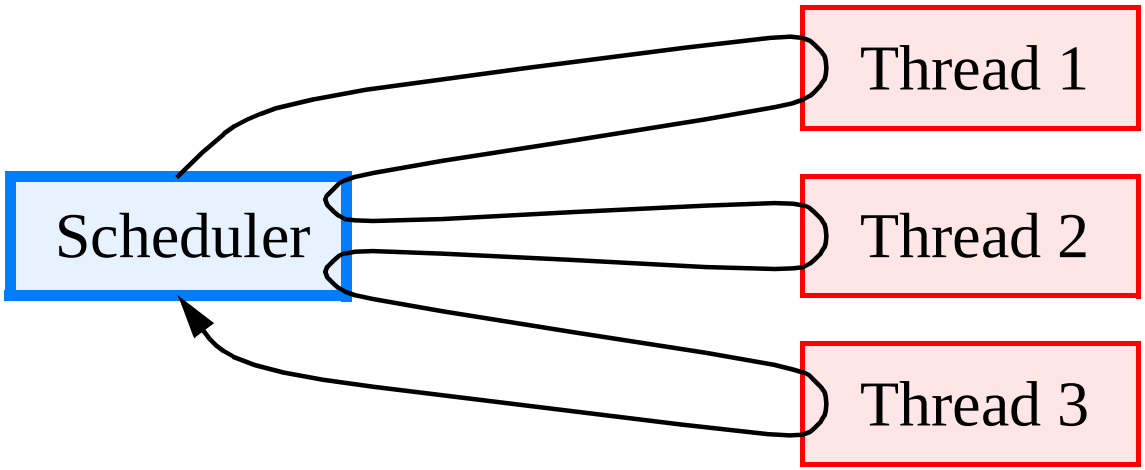
- At time t, hub h has a pool of problems $U(h, t)$
- Somehow measure its *load* $L(h, t)$
- Asynchronously with other computations, do a series of “cycles:”
  - Sense total load
  - Identify potential load donors and requesters
  - Match up pairs
  - Transfer work

Managing Tasks Within a Processor: Non-Preemptive Threads

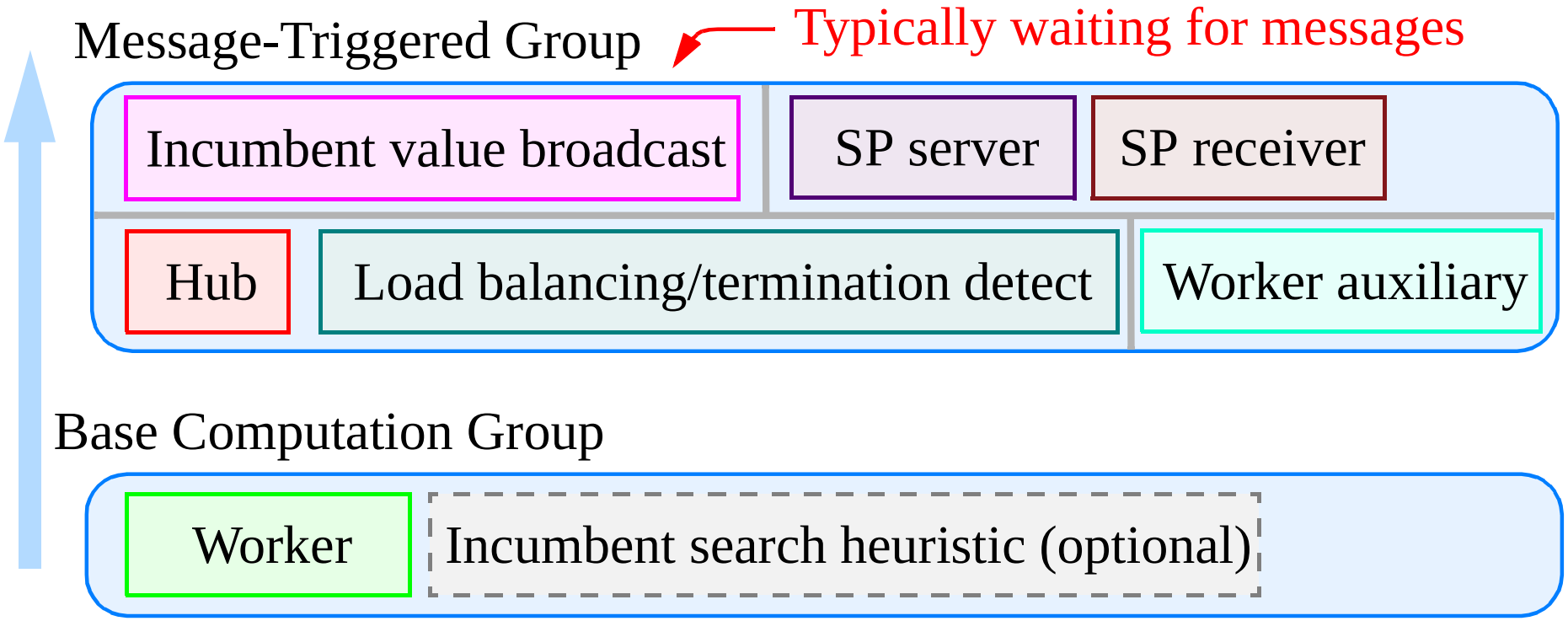
Schedule multiple *threads* of control within each processor

- 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* Mac, Win 3.*x*):



Base Scheduler Setup for PICO



- 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 (not yet implemented for general MIP)

Implements application-specific search heuristic:

- 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

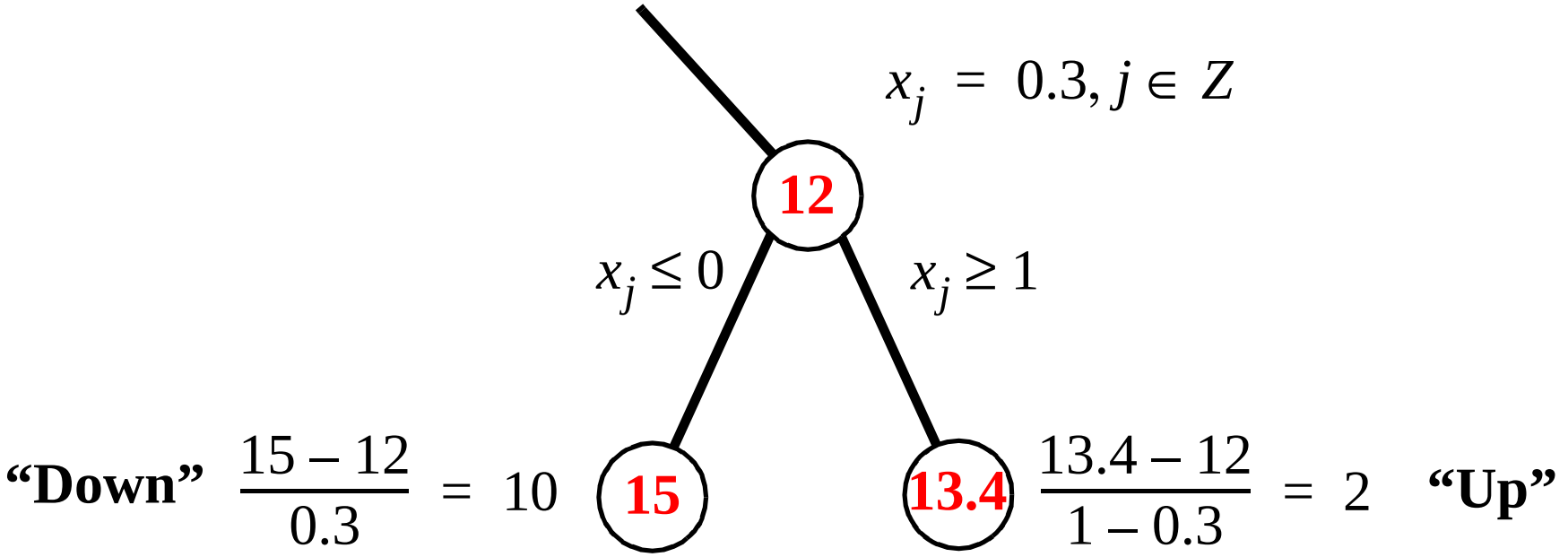
Can adjust priority as run proceeds

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

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

Implementation of the PICO MIP / Pseudocosts

Currently, a fairly standard "bare-bones" MIP algorithm with *pseudocosts* to choose branching variables: average over all uses of x_j as a branching variable:



- When faced with a branching decision, choose the variable which pseudocosts predict will maximize the minimum bound of your children
- Pseudocost tables accumulate over the exploration of the tree; constitute supplementary global dynamic state

Parallel Pseudocost Implementation

Use general approach suggested by Linderoth:

Solve root problem (redundantly on all processors) in "preprocess" phase:

- Probe all integrality-violating variables and compute one set of up and down pseudocosts for each
- Spread this work evenly among all processors
- Do all-to-all gather operation to broadcast results

Once searching tree, just accumulate pseudocost data locally, except...

- If no prior pseudocost data exists for an integrality-violating variable, probe up and down branches and broadcast resulting pseudocosts to all other workers

Requires one extra thread to receive pseudocost broadcasts.

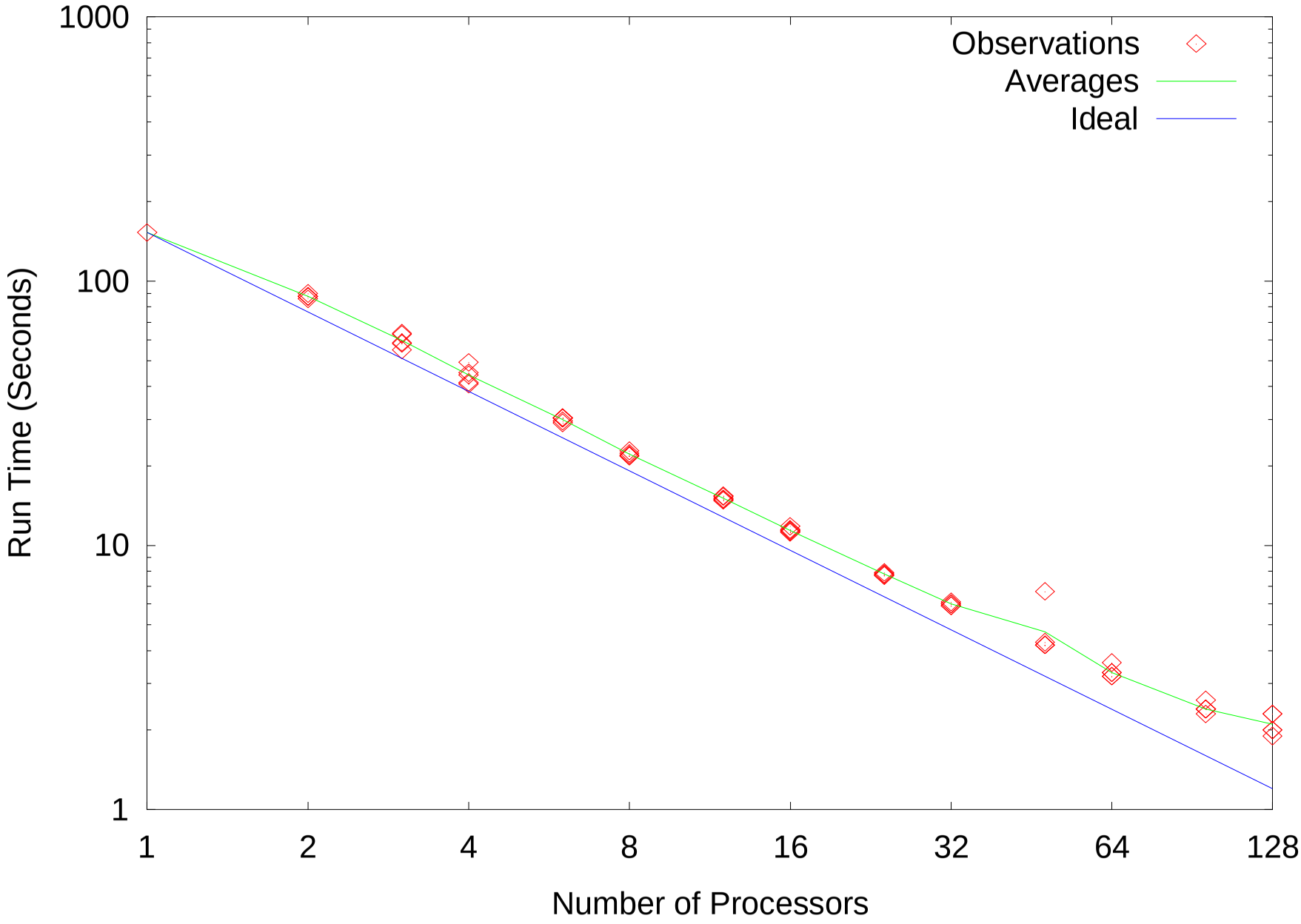
Preliminary Computational Testing on "Janus" at Sandia

- Full configuration is 4,536 nodes (we've only tried up to 128 so far)
- Each node has two Pentium-II/333, 256MB RAM
  - Can use first Pentium for user code, second Pentium as communication/ OS co-processor, *or* split each node into two 128MB "virtual" nodes

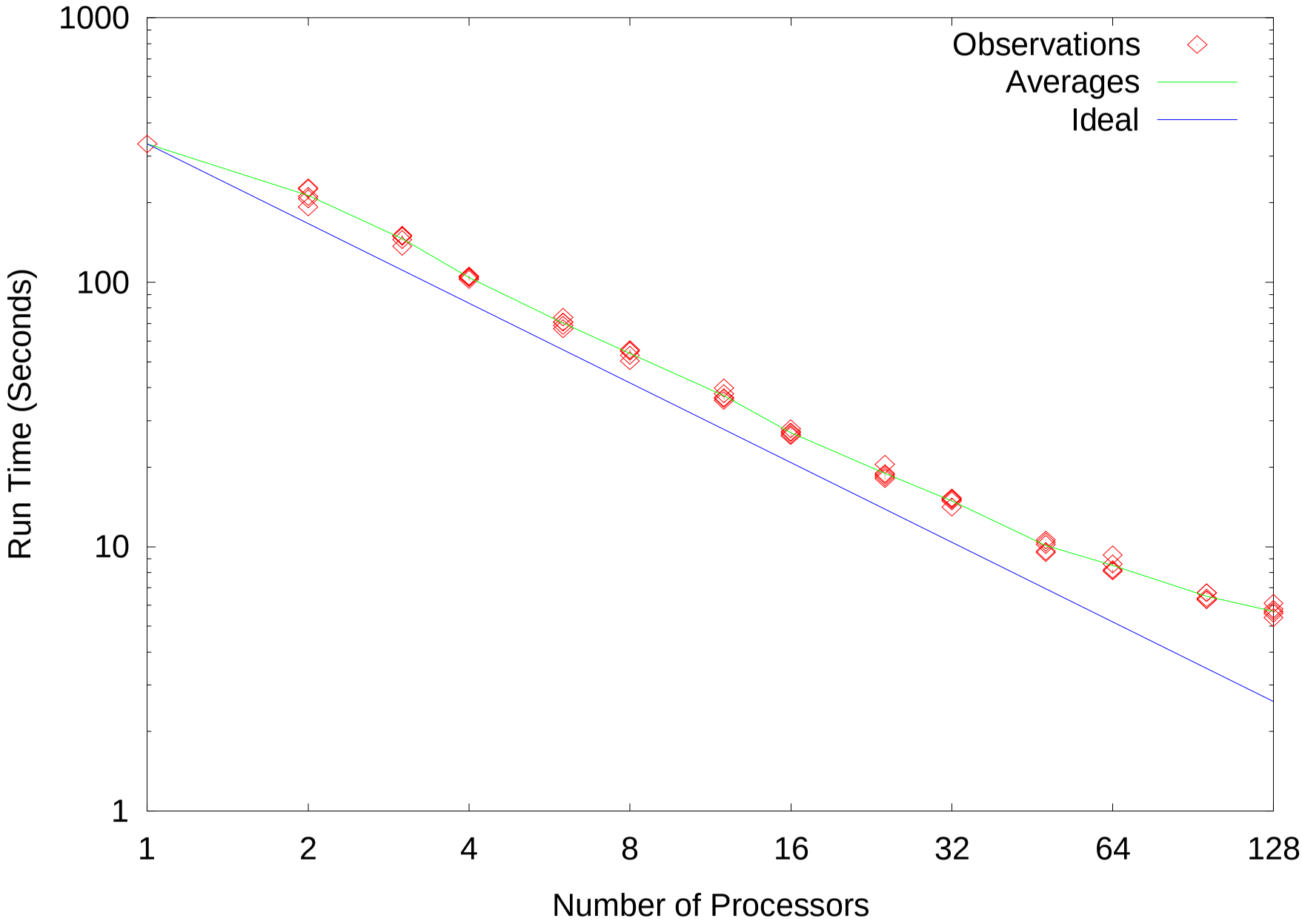
Our tests so far:

- Medium difficulty MIP's from MIPLIB
- LP Solver = CPLEX 6.0
- 1 node (serial layer)
- 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128 nodes (parallel layer)
- Presented below: cluster size = 4

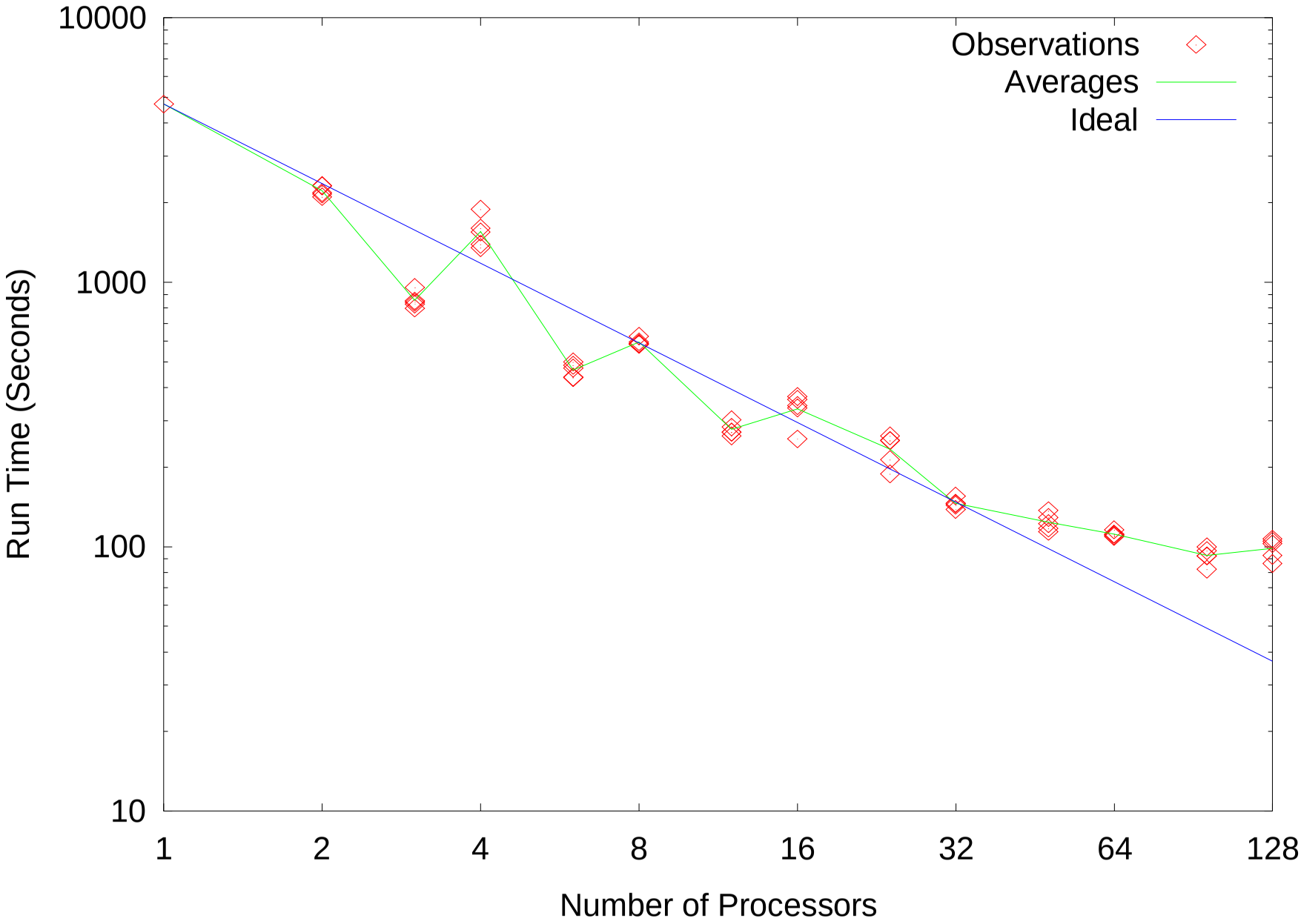
Preliminary Computational Results: bell3a



Preliminary Computational Results: stein45



Preliminary Computational Results: qiu



Future Directions

- Work out more bugs and kinks
- (Parallel) incumbent heuristic for general MIP (Nediak)
- More difficult MIP's on larger processor configurations
- Branch and cut

Peek at some graphical logs of PICO running...