Region-Based Memory Management for Task Dataflow Models

Dimitrios S. Nikolopoulos

Joint work with: Spyros Lyberis and Polyvios Pratikakis
Computer Architecture and VLSI Systems Laboratory (CARV)
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)

EPoPPEA, January 24, 2012

Outline

The Problem

Manycore Processors and Shared Memory

- Shared memory can be hard to scale to many cores
  - Cache coherence becomes expensive
  - Causes excessive communication, memory contention
- Manycore processors that relax shared memory abstraction
  - AMD Opteron: 12 cores, NUMA, MMU per core
  - Cell Broadband Engine: 1 + 8 cores, no shared memory
  - GPUs: Thousands of cores, small shared memories between few cores
  - Intel Single Chip Cloud: 48 cores, no shared memory
- Increased demand for parallel software development
  - Threads and shared memory: accessible, error-prone
  - Message-passing: tedious, experts-only
  - Both are non-deterministic: difficult to debug and understand

Task-parallelism for distributed and shared memory

- Available in several contemporary programming models (Sequoia, Cilk, OMPSs)
- Recursive task-parallelism
  - Parallel programs are composed from nested parallel tasks
- Annotate each task with its *memory footprint*
- Runtime system performs dependency analysis, scheduling, all required data transfers
- Support region-based memory management
  - Easier to express irregular task footprints
  - Enables use of dynamic data structures

Benefits

- Shared memory abstraction
  - Resemble shared memory programming
  - Runtime system hides data transfers among core memories
  - Use memory footprints to transfer required data locally before starting a task
  - Alternatively, schedule a task near the data
- Message-passing execution semantics
  - Tasks compute on local data
  - Hierarchical scheduling, easier to scale to more cores
  - Remove shared-memory bottleneck
  - No need for complicated SDSM protocol on every access
- Deterministic
  - Implicit, correct synchronization
  - Repeatable behavior
  - Formal proof

Outline

Motivation for Regions

- Task memory footprints are easy to express if the are composed of few objects, contiguous in memory
- What if the task memory footprint is:
  - A linked list or part of it?
  - A tree or part of it?
  - A graph or part of it?
- Hard to use many common linked data structures
- Hard to express irregular algorithms and applications

Example

Region-based Memory Management

```
region G, L, R;
Object *v0;
init () {
  G = newregion();
  L = newsubregion(G);
  R = newsubregion(G);
  v0 = ralloc(G, sizeof(Object));
  v0->left = ralloc(L, sizeof(Object));
  v0->right = ralloc(R, sizeof(Object));
}
```

Example

Region-based Memory Management

```
region G, L, R;
Object *v0;
init () {
  G = newregion();
  L = newsubregion(G);
  R = newsubregion(G);
  v0 = ralloc(G, sizeof(Object));
  v0->left = ralloc(L, sizeof(Object));
  v0->right = ralloc(R, sizeof(Object));
}
```

Example

Region-based Memory Management

```
region G, L, R;
Object *v0;
init () {
  G = newregion();
  L = newsubregion(G);
  R = newsubregion(G);
  v0 = ralloc(G, sizeof(Object));
  v0->left = ralloc(L, sizeof(Object));
  v0->right = ralloc(R, sizeof(Object));
}
```

Example

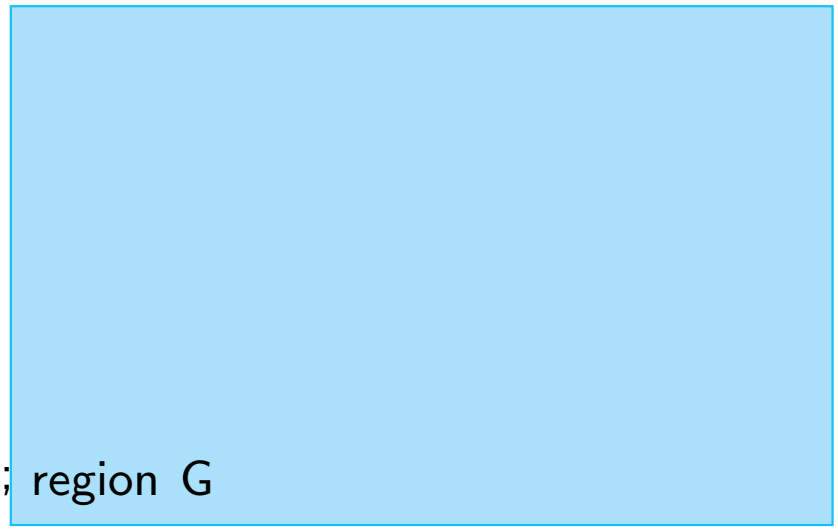
Region-based Memory Management

```
region G, L, R;
Object *v0;
init () {
  G = newregion();
  L = newsubregion(G);
  R = newsubregion(G);
  v0 = ralloc(G, sizeof(Object));
  v0->left = ralloc(L, sizeof(Object));
  v0->right = ralloc(R, sizeof(Object));
}
```

Example

Region-based Memory Management

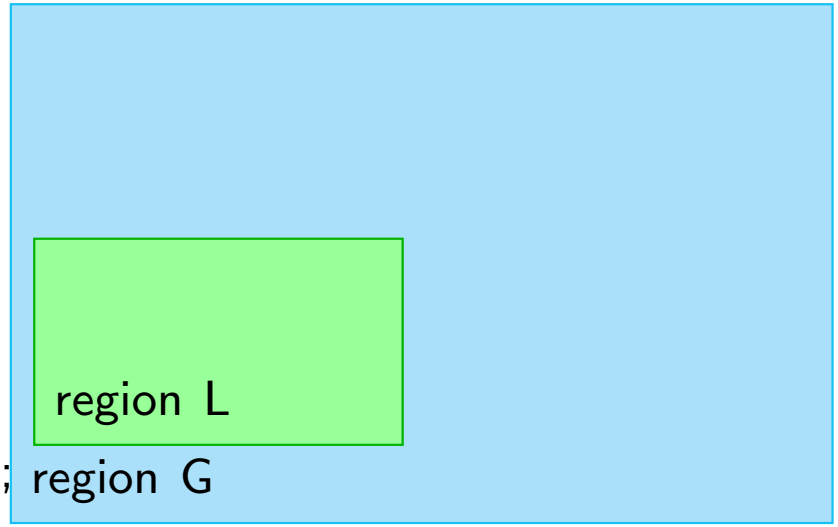
```
region G, L, R;
Object *v0;
init () {
  G = newregion();
  L = newsubregion(G);
  R = newsubregion(G);
  v0 = ralloc(G, sizeof(Object));
  v0->left = ralloc(L, sizeof(Object));
  v0->right = ralloc(R, sizeof(Object));
}
```



Example

Region-based Memory Management

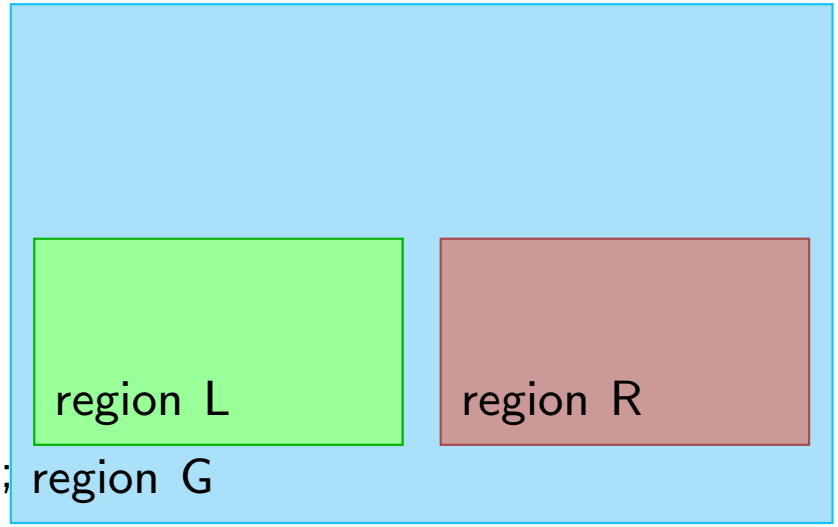
```
region G, L, R;
Object *v0;
init () {
  G = newregion();
  L = newsubregion(G);
  R = newsubregion(G);
  v0 = ralloc(G, sizeof(Object));
  v0->left = ralloc(L, sizeof(Object));
  v0->right = ralloc(R, sizeof(Object));
}
```



Example

Region-based Memory Management

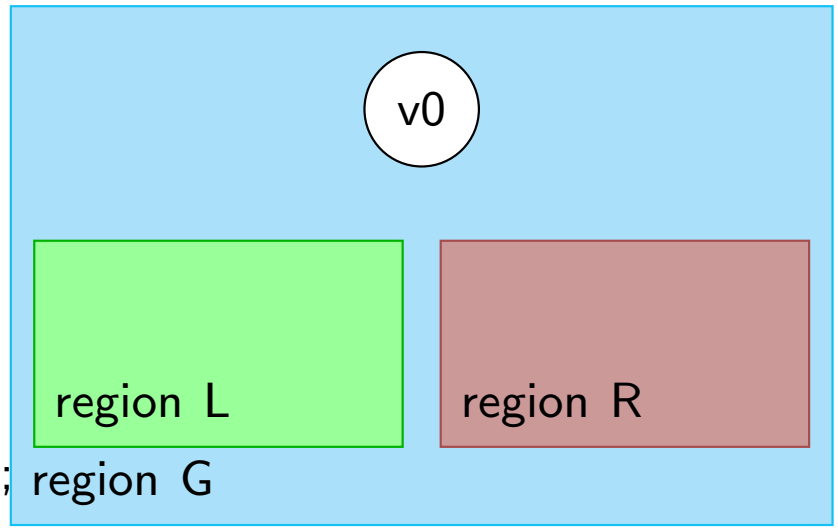
```
region G, L, R;
Object *v0;
init () {
  G = newregion();
  L = newsubregion(G);
  R = newsubregion(G);
  v0 = ralloc(G, sizeof(Object));
  v0->left = ralloc(L, sizeof(Object));
  v0->right = ralloc(R, sizeof(Object));
}
```



Example

Region-based Memory Management

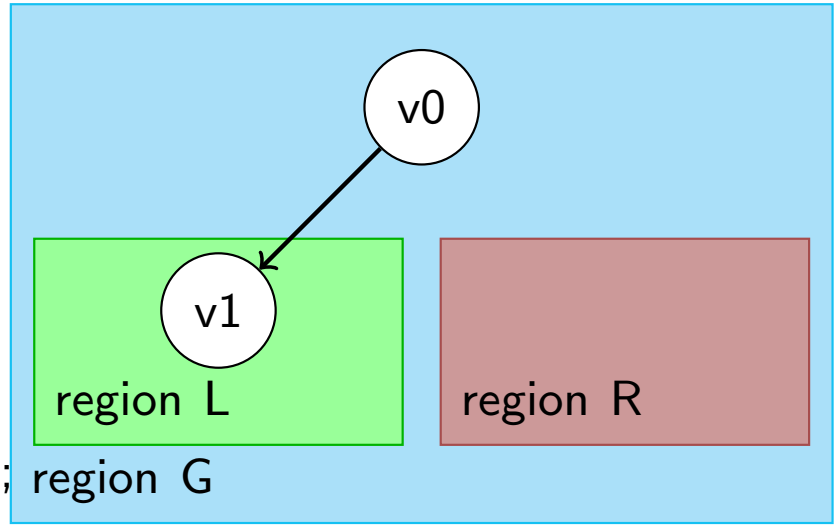
```
region G, L, R;
Object *v0;
init () {
  G = newregion();
  L = newsubregion(G);
  R = newsubregion(G);
  v0 = ralloc(G, sizeof(Object));
  v0->left = ralloc(L, sizeof(Object));
  v0->right = ralloc(R, sizeof(Object));
}
```



Example

Region-based Memory Management

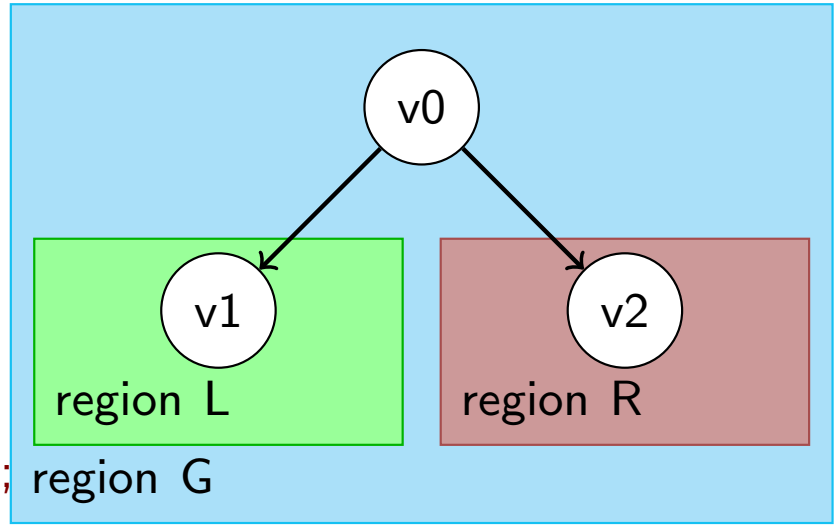
```
region G, L, R;
Object *v0;
init () {
  G = newregion();
  L = newsubregion(G);
  R = newsubregion(G);
  v0 = ralloc(G, sizeof(Object));
  v0->left = ralloc(L, sizeof(Object));
  v0->right = ralloc(R, sizeof(Object));
}
```



Example

Region-based Memory Management

```
region G, L, R;
Object *v0;
init () {
  G = newregion();
  L = newsubregion(G);
  R = newsubregion(G);
  v0 = ralloc(G, sizeof(Object));
  v0->left = ralloc(L, sizeof(Object));
  v0->right = ralloc(R, sizeof(Object));
}
```

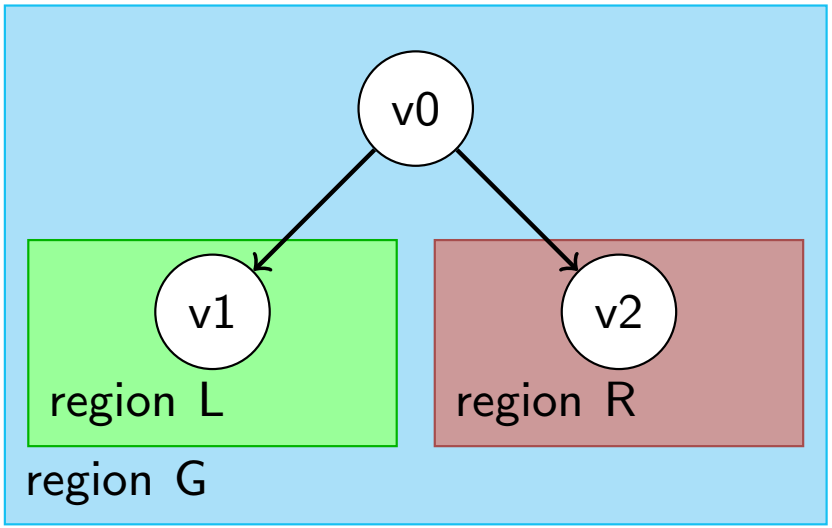


Outline

Example

Tasks and Regions

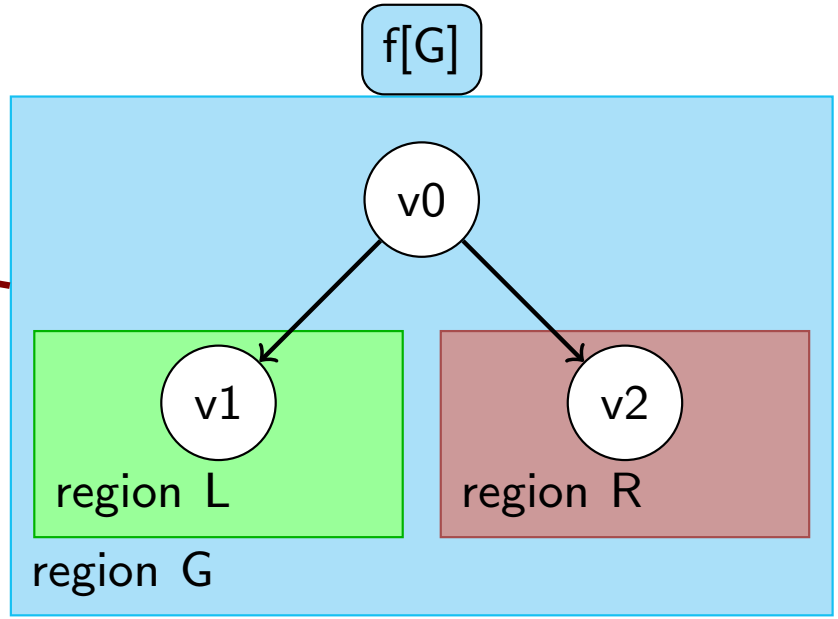
```
f(Object *v0) {
  if (done) return;
  spawn f(v0->left) [inout region L];
  spawn f(v0->right) [inout region R];
  spawn h(v0->left) [inout v0->left];
  spawn h(v0->right) [inout v0->right];
}
```



Example

Tasks and Regions

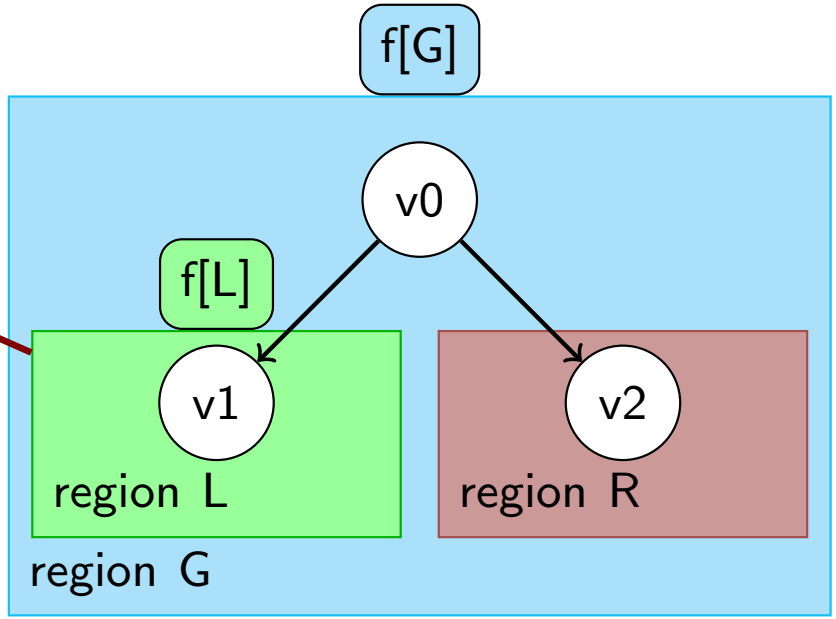
```
f(Object *v0) {
  if (done) return;
  spawn f(v0->left) [inout region L];
  spawn f(v0->right) [inout region R];
  spawn h(v0->left) [inout v0->left];
  spawn h(v0->right) [inout v0->right];
}
```



Example

Tasks and Regions

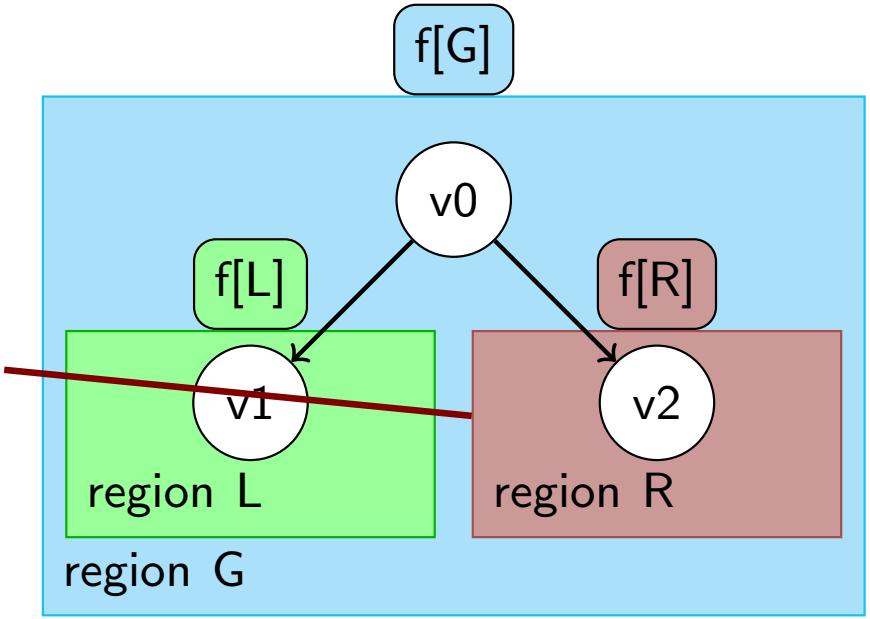
```
f(Object *v0) {
  if (done) return;
  spawn f(v0->left) [inout region L];
  spawn f(v0->right) [inout region R];
  spawn h(v0->left) [inout v0->left];
  spawn h(v0->right) [inout v0->right];
}
```



Example

Tasks and Regions

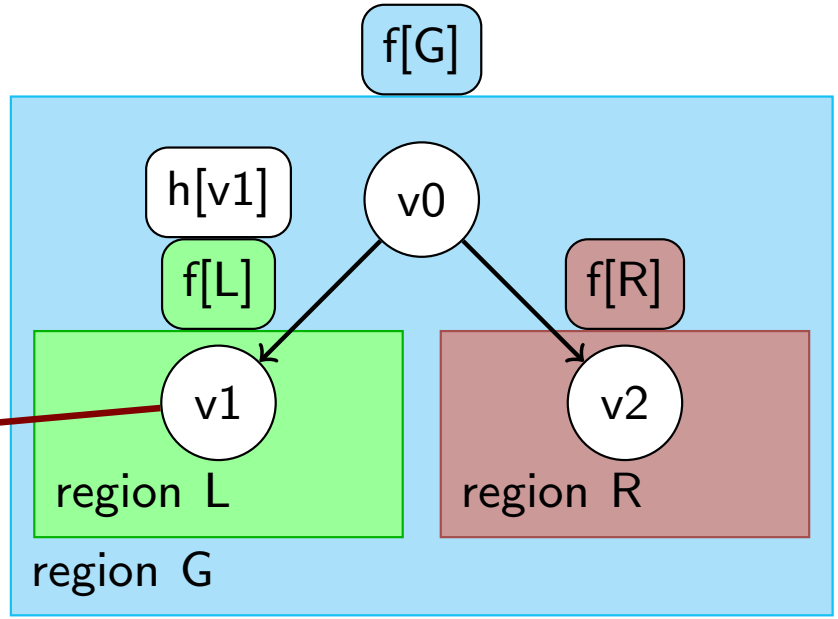
```
f(Object *v0) {
  if (done) return;
  spawn f(v0->left) [inout region L];
  spawn f(v0->right) [inout region R];
  spawn h(v0->left) [inout v0->left];
  spawn h(v0->right) [inout v0->right];
}
```



Example

Tasks and Regions

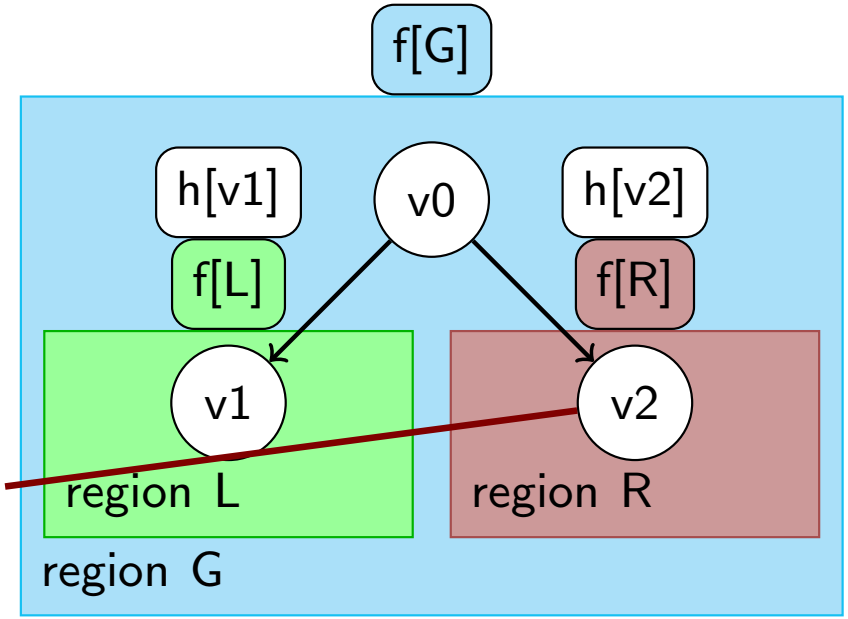
```
f(Object *v0) {
  if (done) return;
  spawn f(v0->left) [inout region L];
  spawn f(v0->right) [inout region R];
  spawn h(v0->left) [inout v0->left];
  spawn h(v0->right) [inout v0->right];
}
```



Example

Tasks and Regions

```
f(Object *v0) {
  if (done) return;
  spawn f(v0->left) [inout region L];
  spawn f(v0->right) [inout region R];
  spawn h(v0->left) [inout v0->left];
  spawn h(v0->right) [inout v0->right];
}
```



Deterministic Scheduler Operational Semantics

- Define small-step operational semantics
  - $\langle T, D, S, R \rangle \rightarrow_p \langle T', D', S', R' \rangle$
- Memory model:
  - Global address space: the *store* S includes all memory addresses
  - Distributed memory implementation: each task only accesses memory locations declared in its footprint
- Scheduling algorithm:
  - Dependency metadata D maintain a *task queue* per memory location
  - Always possible to spawn a task
  - But, it can only run when at the head of queue for all locations in the footprint

Deterministic Proof Technique (Pratikakis et. al, MSPC'11)

- Define sequential operational semantics
  - $\langle S, e \rangle \rightarrow_s \langle S', e' \rangle$
- Prove *sequential equivalence* on execution traces
  - Intuitively: Every parallel execution will produce the same value and memory state as the sequential execution
  - More precisely: Given a program e and a finite (terminating) parallel execution trace for e that produces a value v and a memory state S, we can always construct a sequential execution trace for e that also returns v and produces S
- Proof by induction on the parallel trace
  - We can always reorder steps in the parallel trace to bring the next "sequential" step to the front of the trace
  - Proof does not require scoped parallelism (sync)
  - Similar to a confluence proof

Outline

Distributed Memory Management API

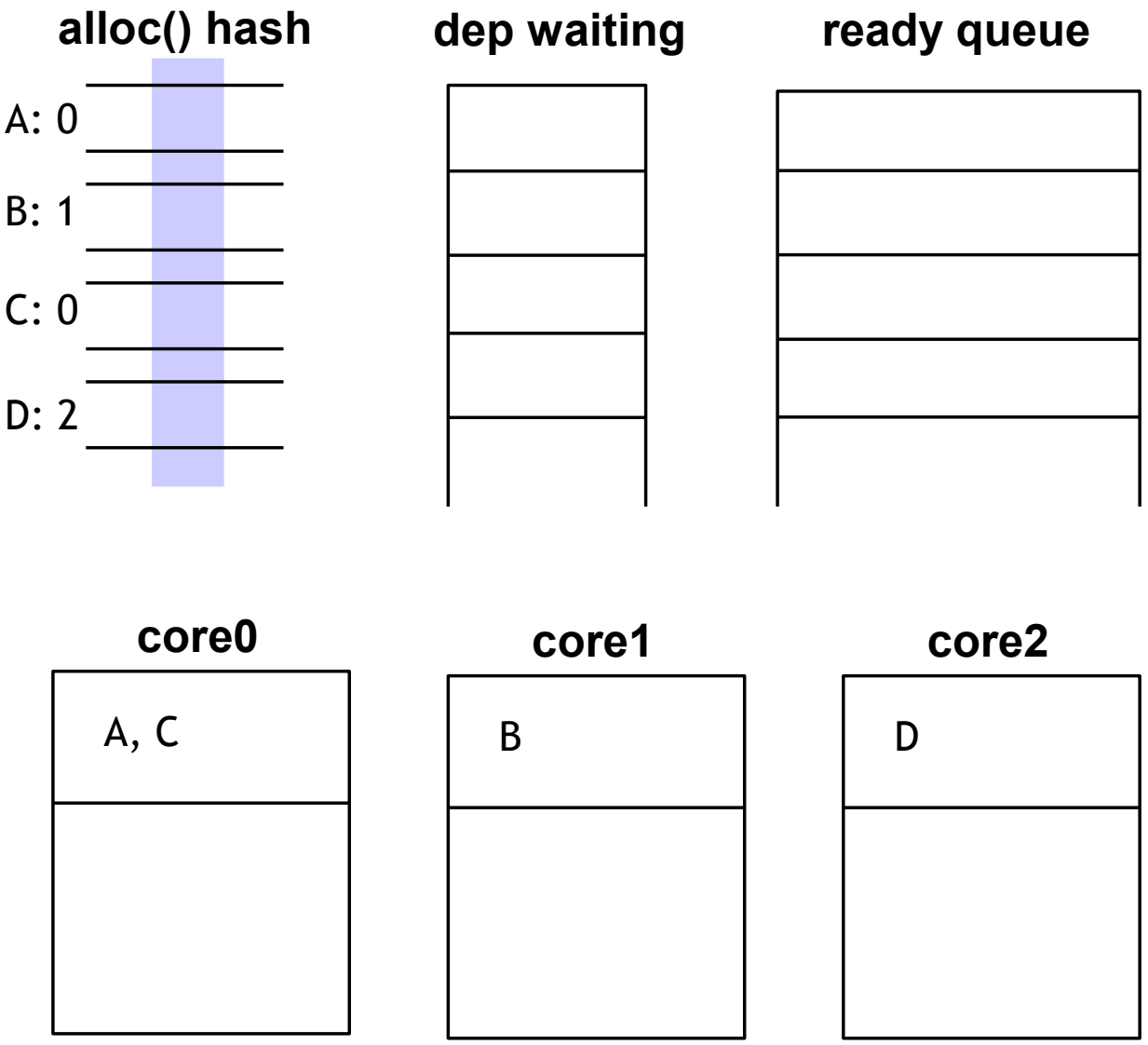
- Object allocation:

```
void *sys_alloc(size_t size, rid_t region);

void sys_balloc(size_t size, rid_t region,
                int num_objects, void **objects);

void sys_free(void *ptr);

void *sys_realloc(void *ptr, size_t size, rid_t rgn);
```

- Region allocation:

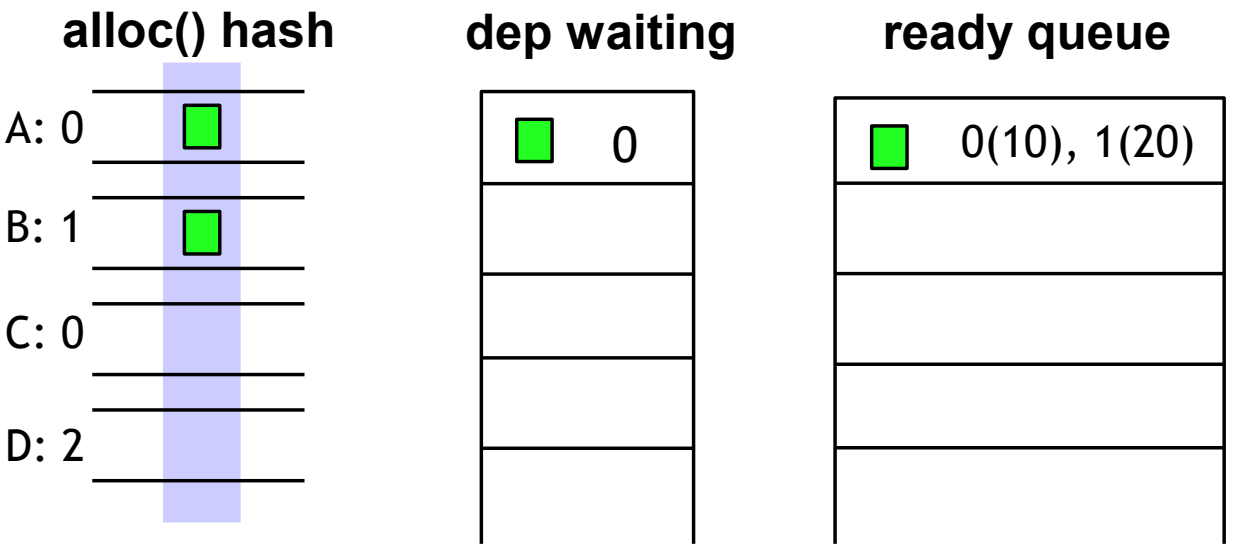
```
rid_t sys_ralloc(rid_t parent, char level_hint);

void sys_rfree(rid_t region);
```

Distributed Memory Implementation by Example

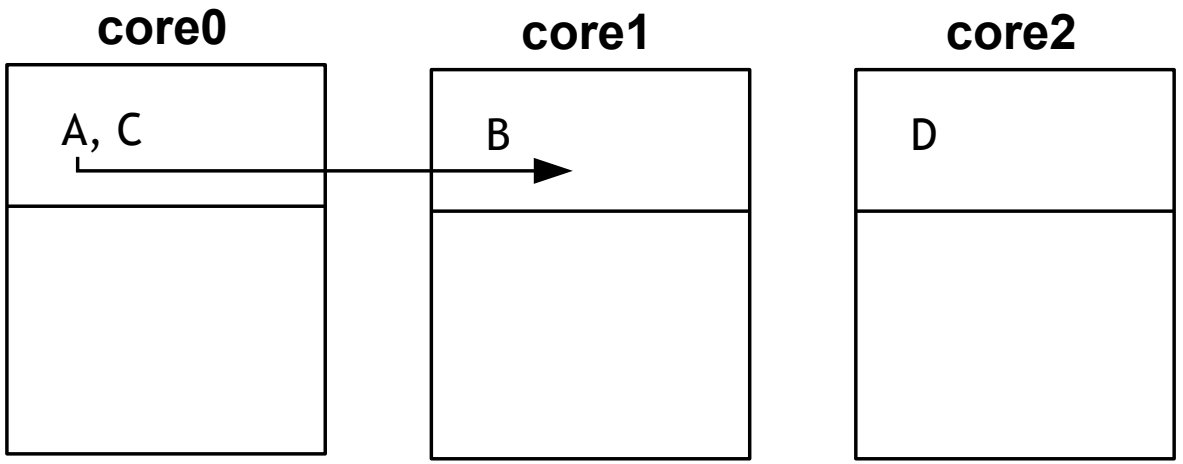



```
A = malloc(10);
B = malloc(20);
C = malloc(5);
D = malloc(80);
```

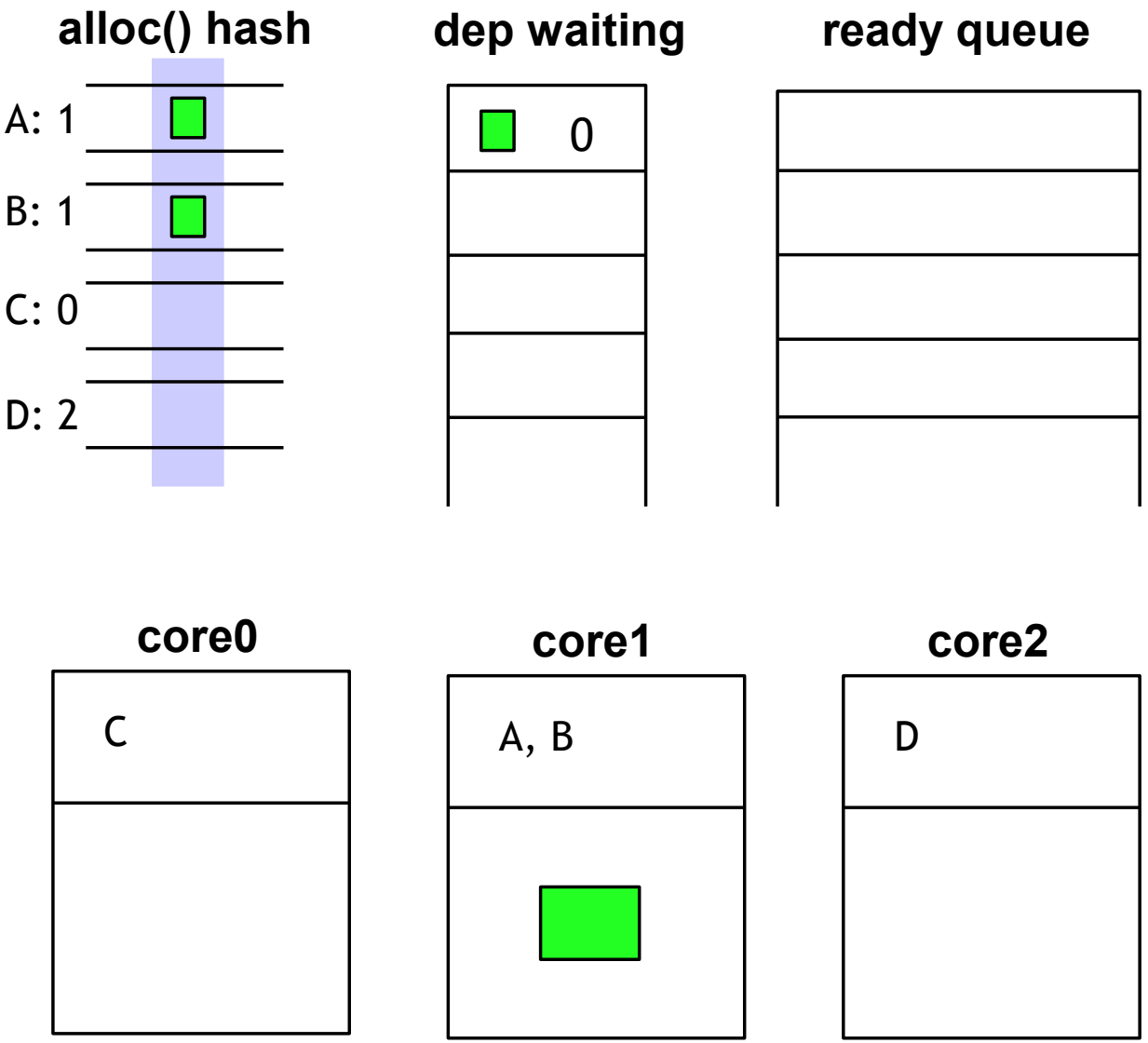
Distributed Memory Implementation by Example




```
A = malloc(10);
B = malloc(20);
C = malloc(5);
D = malloc(80);

# task in A, out B
task1(A, B);
```

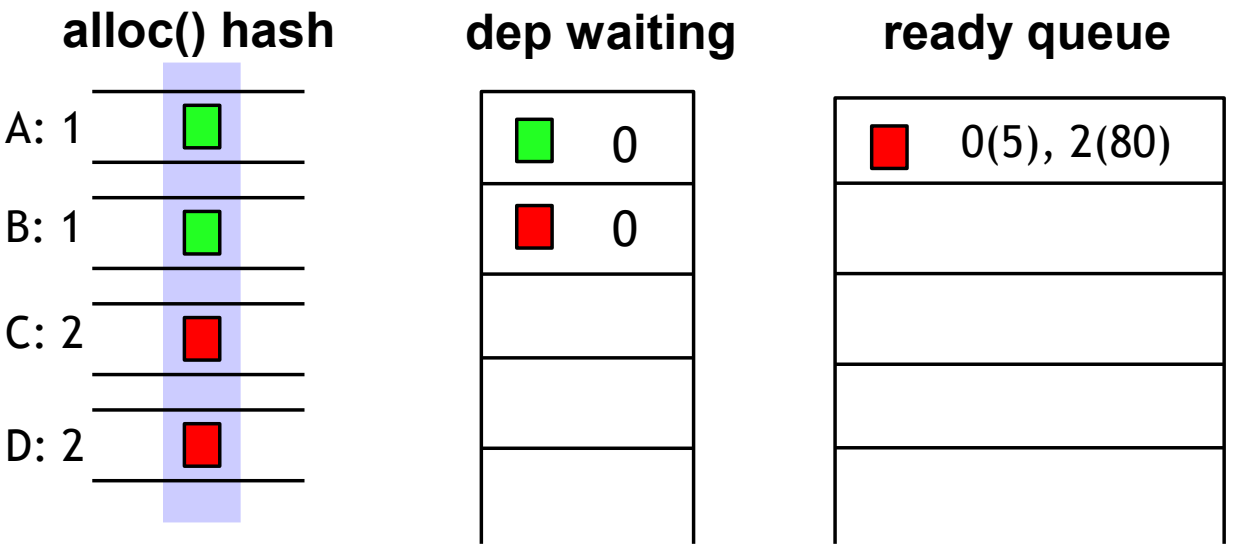


Distributed Memory Implementation by Example

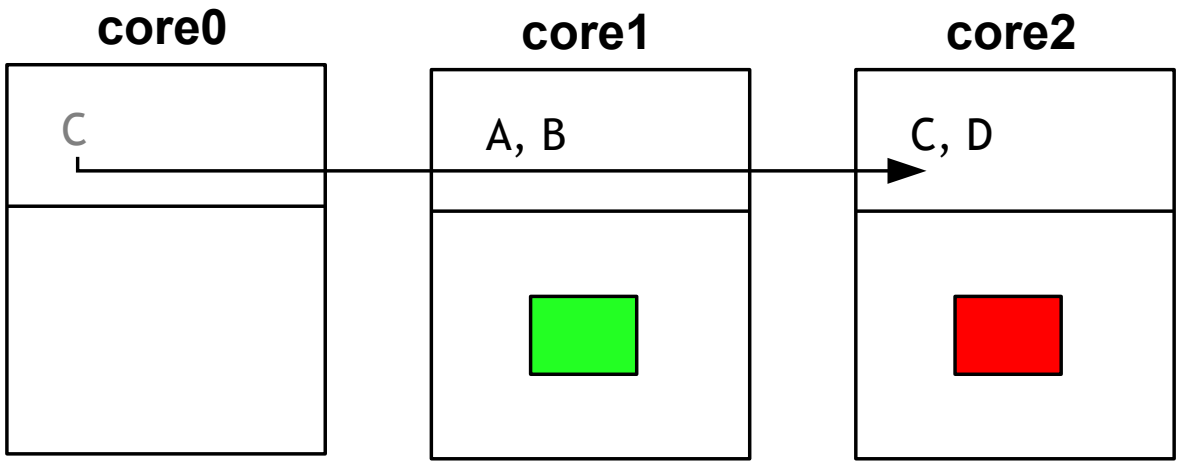



```
A = malloc(10);
B = malloc(20);
C = malloc(5);
D = malloc(80);

# task in A, out B
task1(A, B);
```

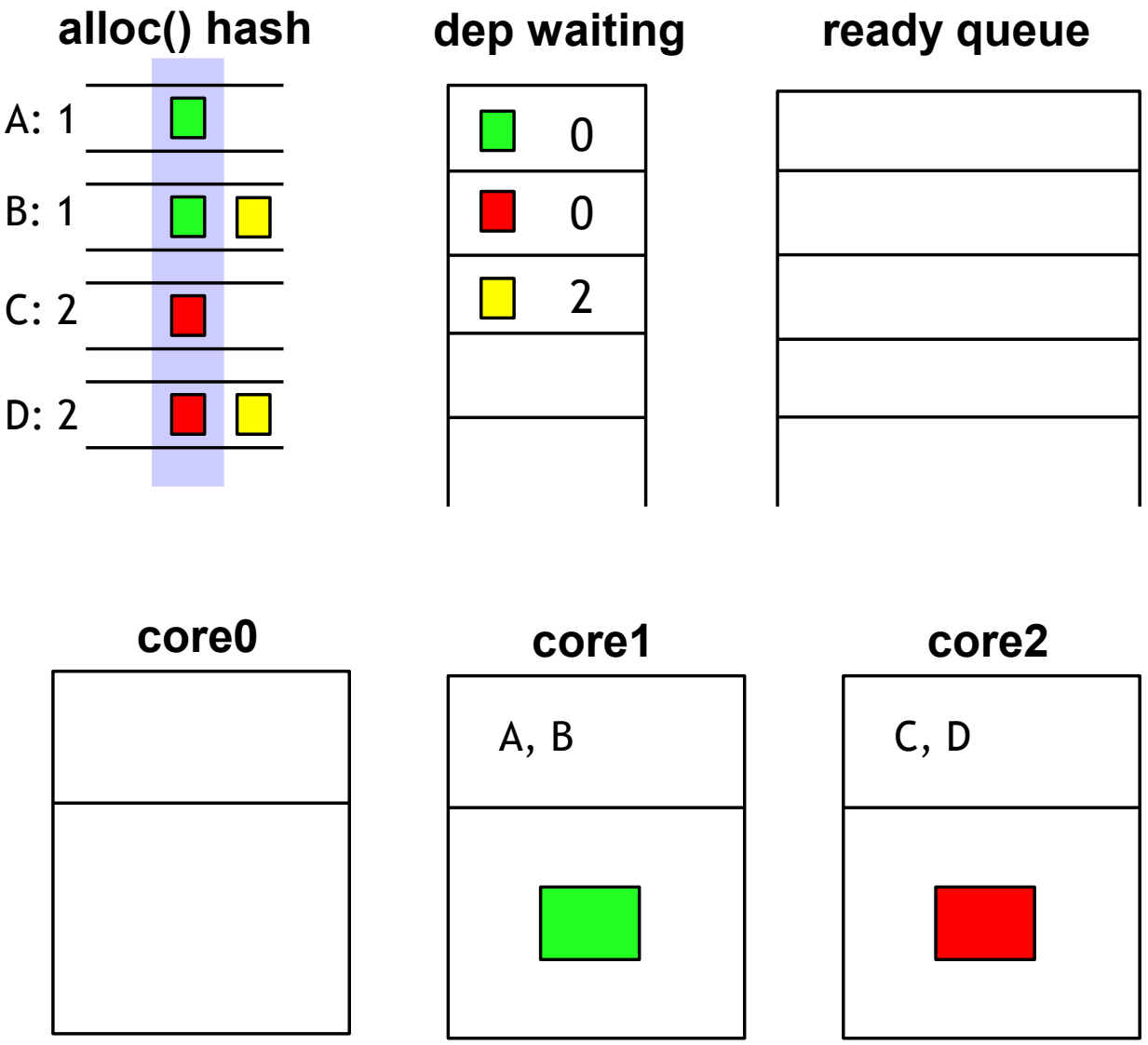
Distributed Memory Implementation by Example




```
A = malloc(10);
B = malloc(20);
C = malloc(5);
D = malloc(80);

# task in A, out B
task1(A, B);

# task in C, out D
task2(C, D);
```

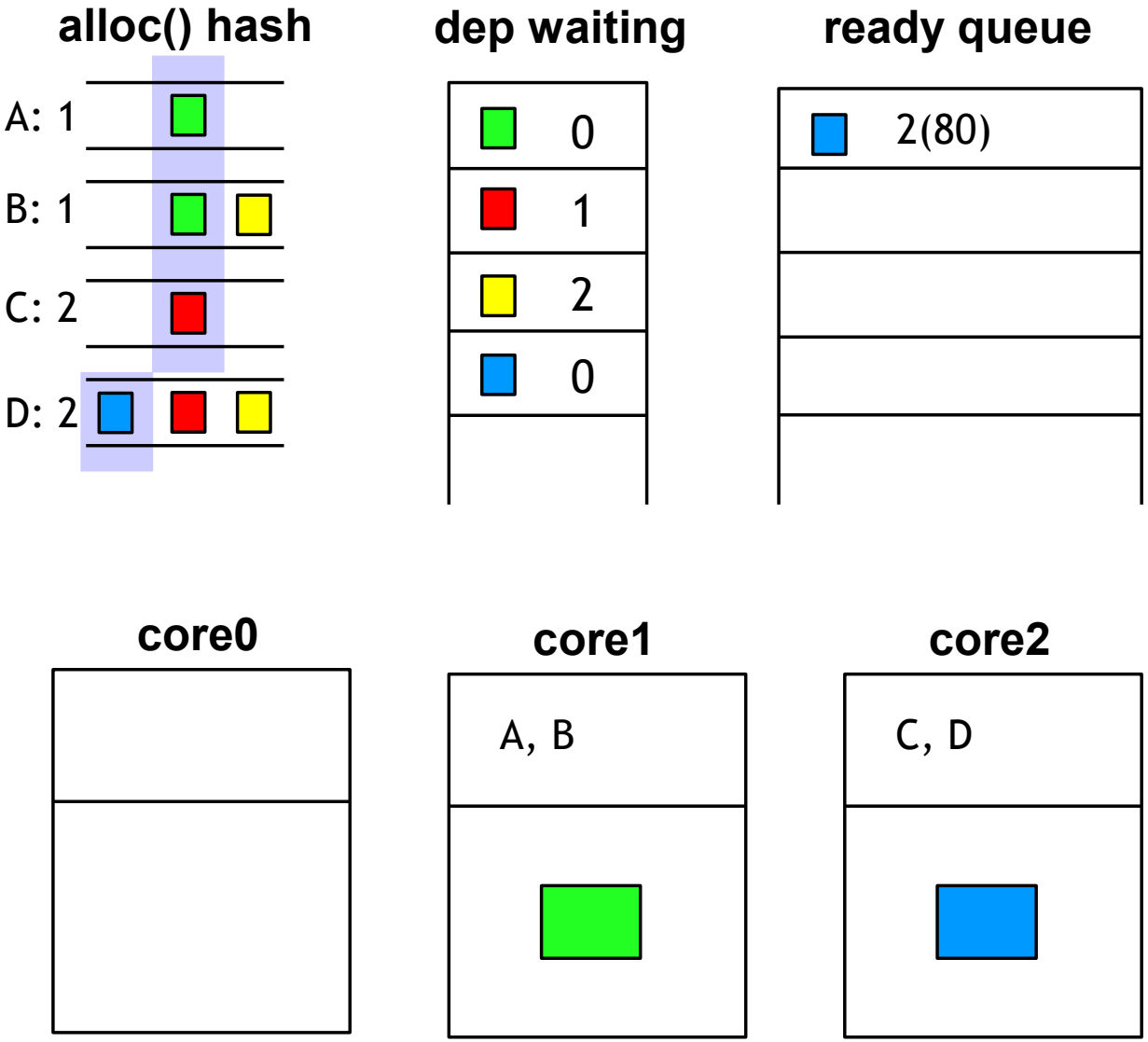


Distributed Memory Implementation by Example



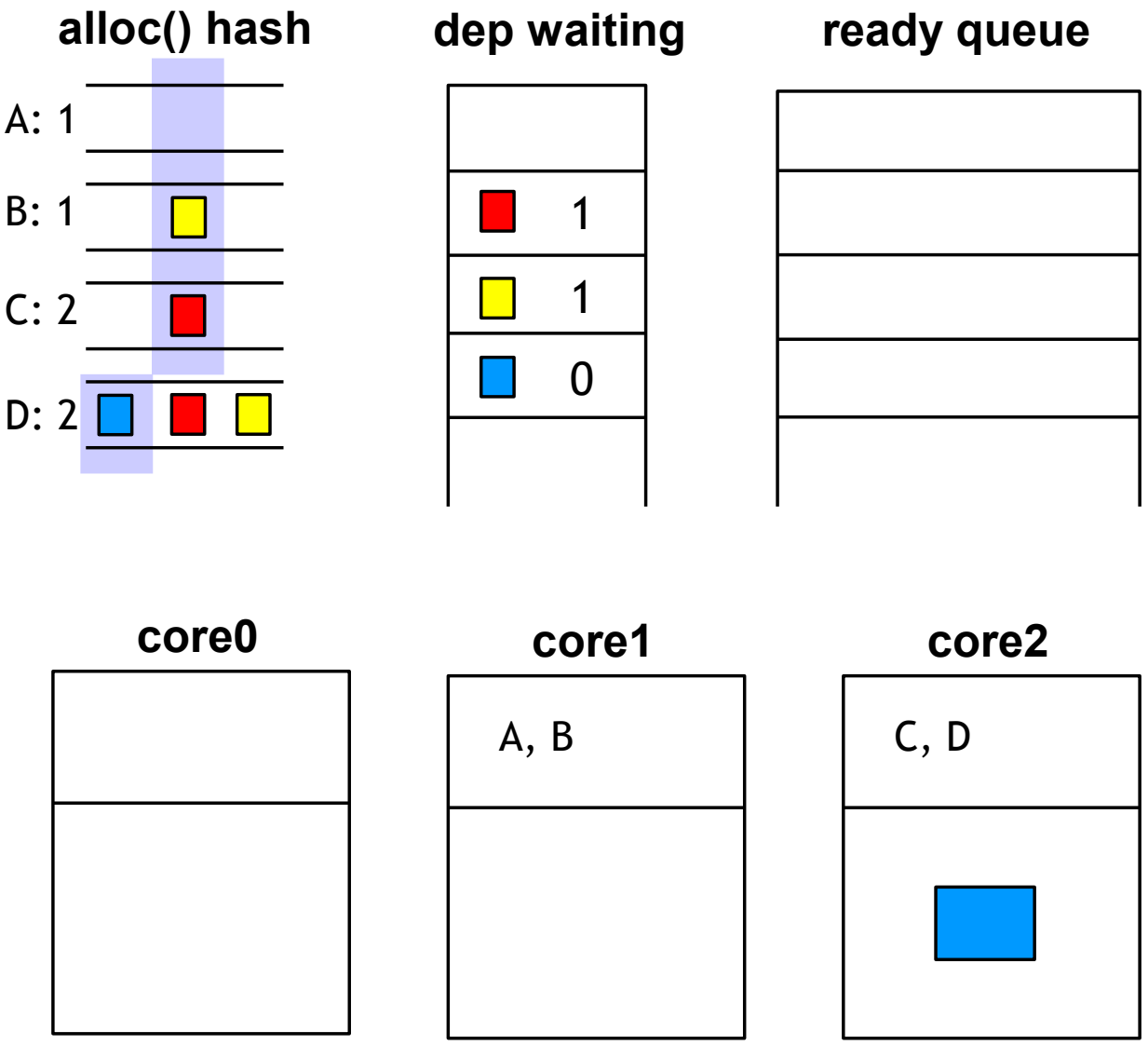

```
A = malloc(10);
B = malloc(20);
C = malloc(5);
D = malloc(80);

# task in A, out B
task1(A, B);

# task in C, out D
task2(C, D);

# task in B, inout D
task3(B, D);
```

Distributed Memory Implementation by Example



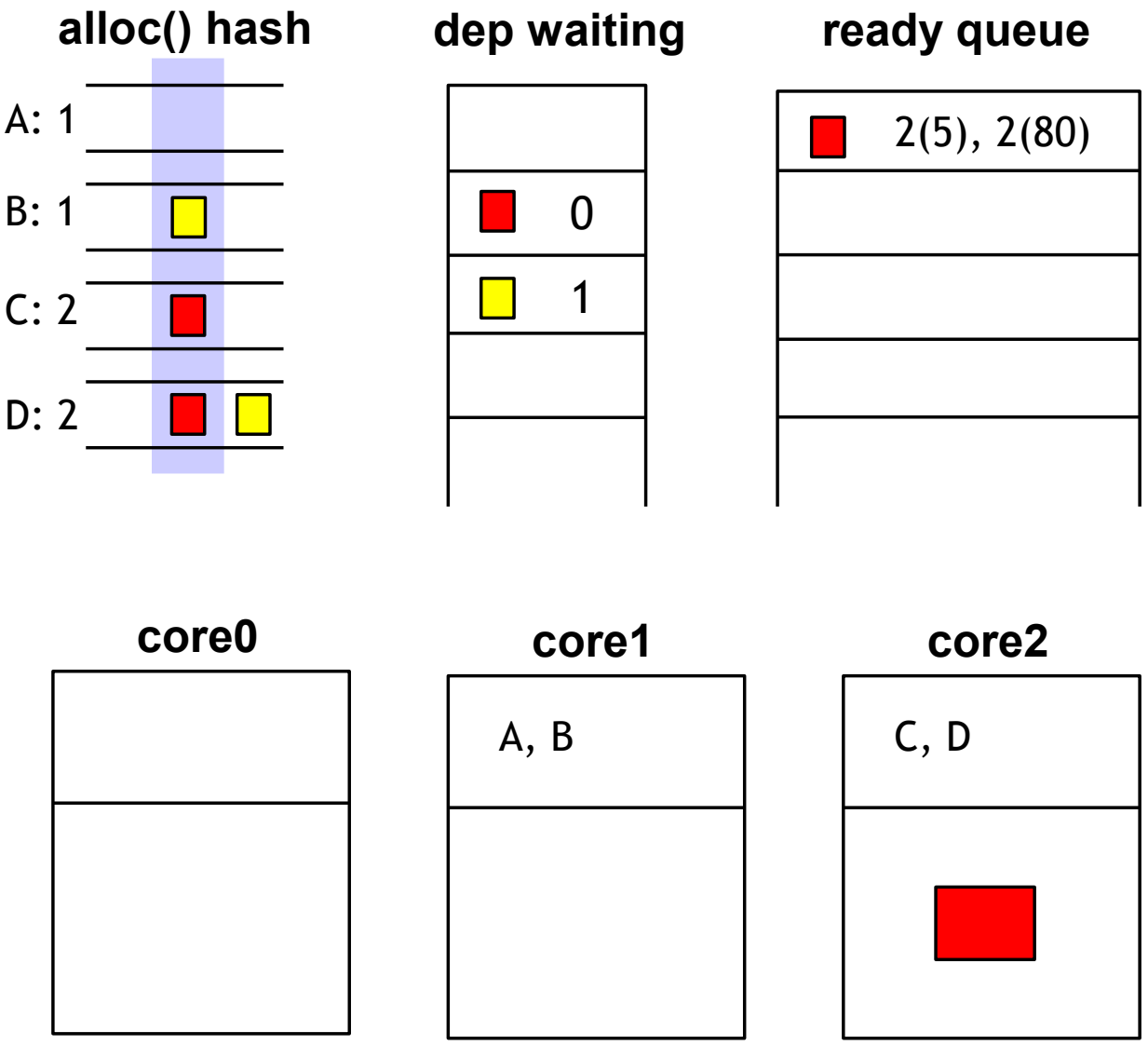

```
A = malloc(10);
B = malloc(20);
C = malloc(5);
D = malloc(80);

# task in A, out B
task1(A, B);

# task in C, out D
task2(C, D);

# task in B, inout D
task3(B, D);


task2 (C, D) {
  ...
  # task inout D
  task4 (D);
  # wait on D
}
```

Distributed Memory Implementation by Example



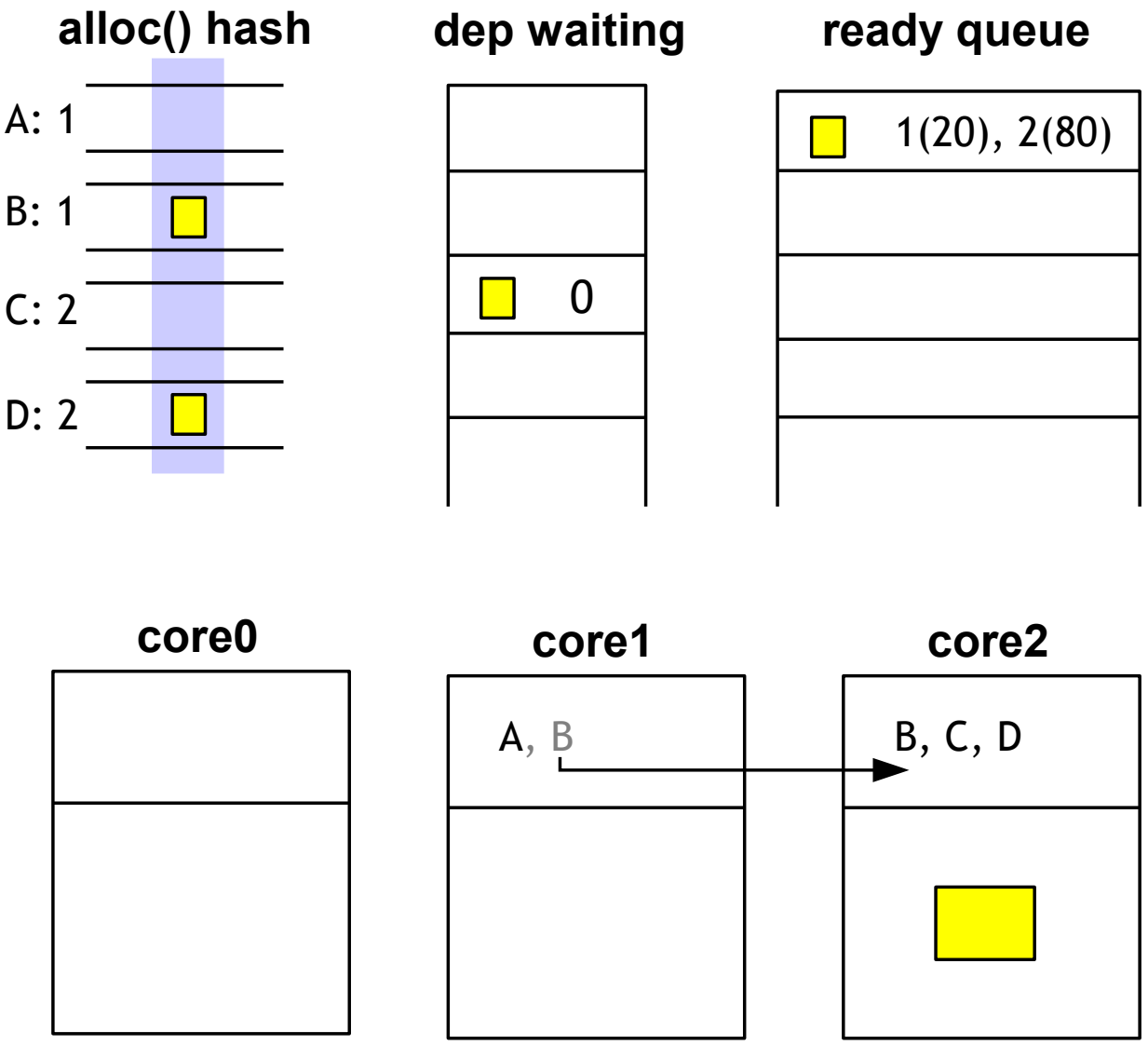

```
A = malloc(10);
B = malloc(20);
C = malloc(5);
D = malloc(80);

# task in A, out B
task1(A, B);

# task in C, out D
task2(C, D);

# task in B, inout D
task3(B, D);


task2 (C, D) {
   ...
   # task inout D
   task4 (D);
   # wait on D
}
```

Distributed Memory Implementation by Example



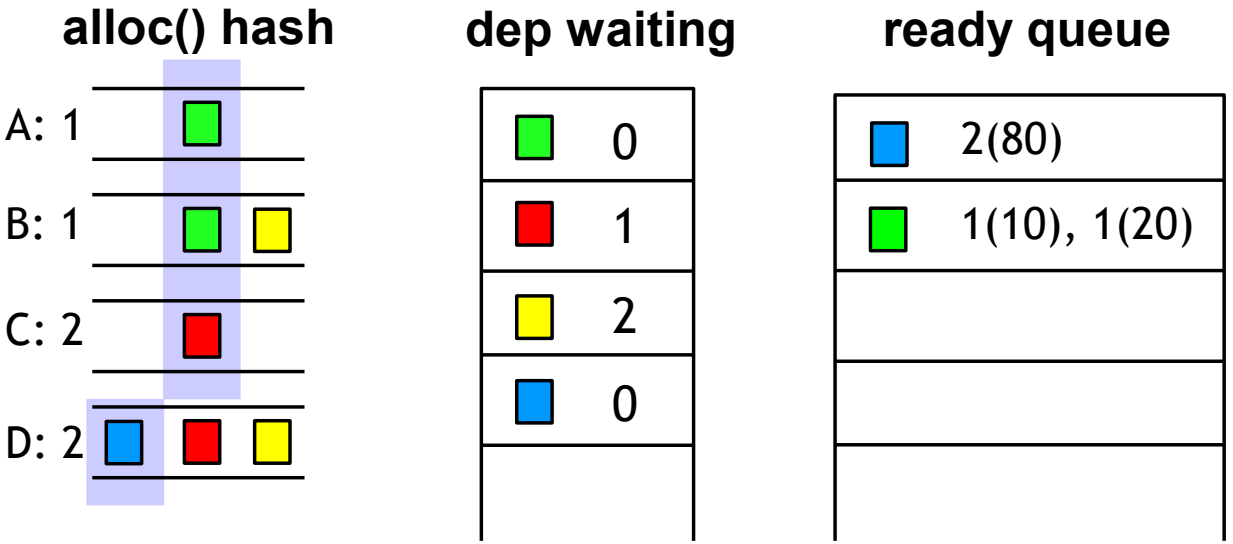

```
A = malloc(10);
B = malloc(20);
C = malloc(5);
D = malloc(80);

# task in A, out B
task1(A, B);

# task in C, out D
task2(C, D);

# task in B, inout D
task3(B, D);


task2 (C, D) {
   ...
   # task inout D
   task4 (D);
   # wait on D
}
```

Distributed Memory Implementation by Example




```
A = malloc(10);
B = malloc(20);
C = malloc(5);
D = malloc(80);

# task in A, out B
task1(A, B);

# task in C, out D
task2(C, D);

# task in B, inout D
task3(B, D);


task2 (C, D) {
  ...
  # task inout D
  task4 (D);
  # wait on D
}
```

Distributed Memory Implementation by Example




```
A = malloc(10);
B = malloc(20);
C = malloc(5);
D = malloc(80);

# task in A, out B
task1(A, B);

# task in C, out D
task2(C, D);

# task in B, inout D
task3(B, D);


task2 (C, D) {
  ...
  # task inout D
  task4 (D);
  # wait on D
}
```

Decode: Enqueue task args into alloc() hash from left/right side; Count non-ownership args in dep waiting

Issue: If dep waiting reaches 0, put task in ready queue along with affinity stats

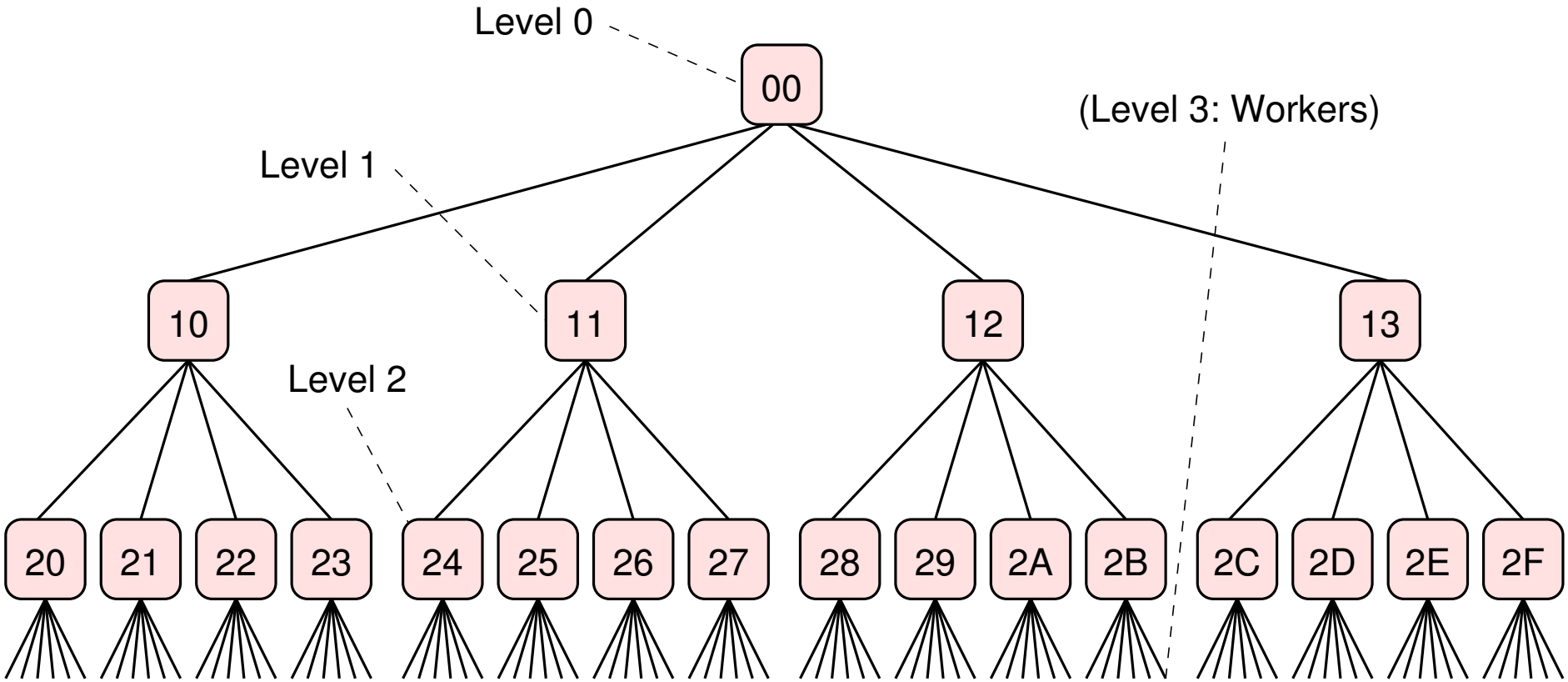
Prepare: Decide where to run, fetch non-local data

Run: Execute code when fetch is finished

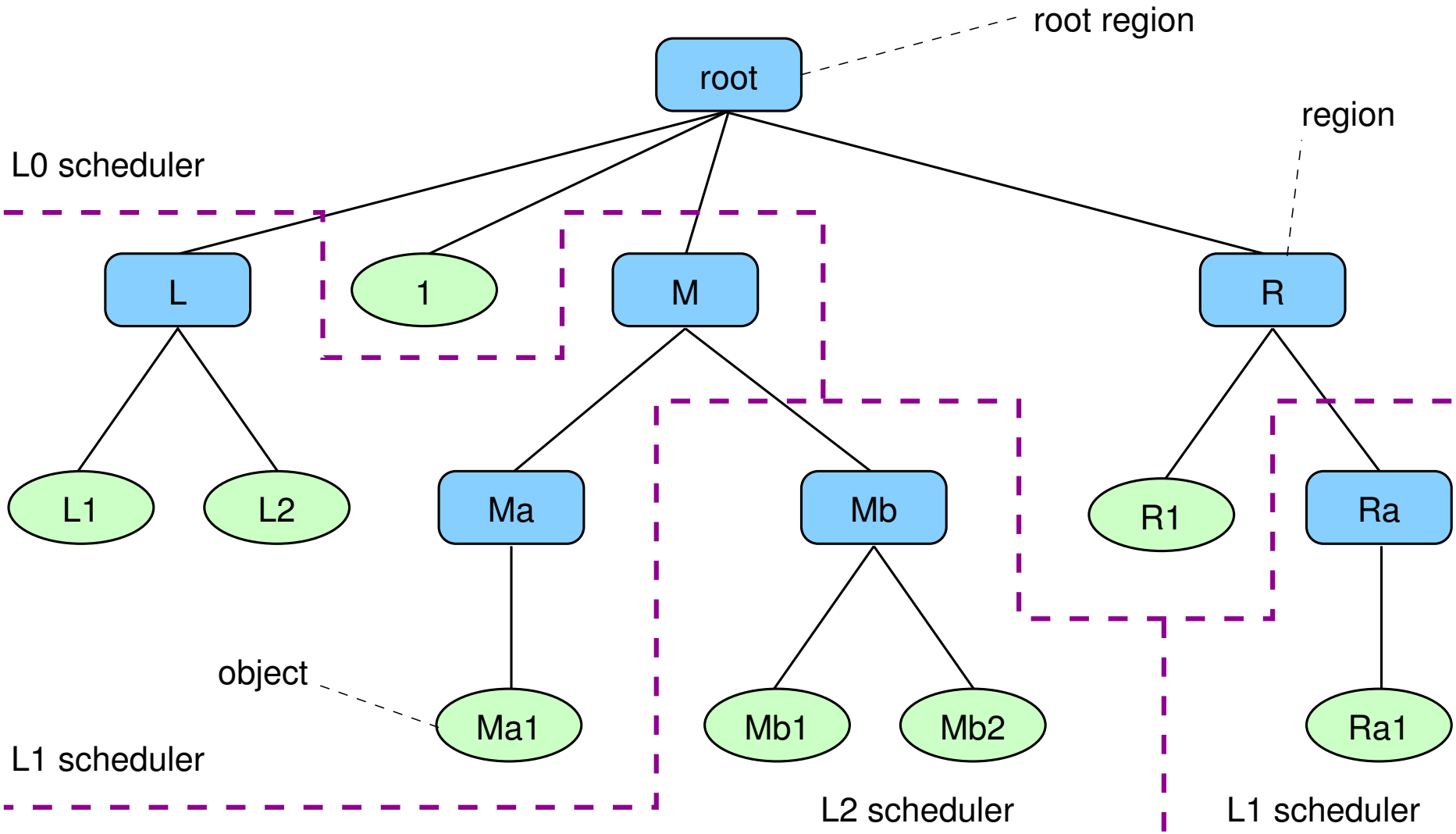
Collect: Update alloc() hash ownership; Update dep waiting for these objects; Maybe goto Issue

Outline

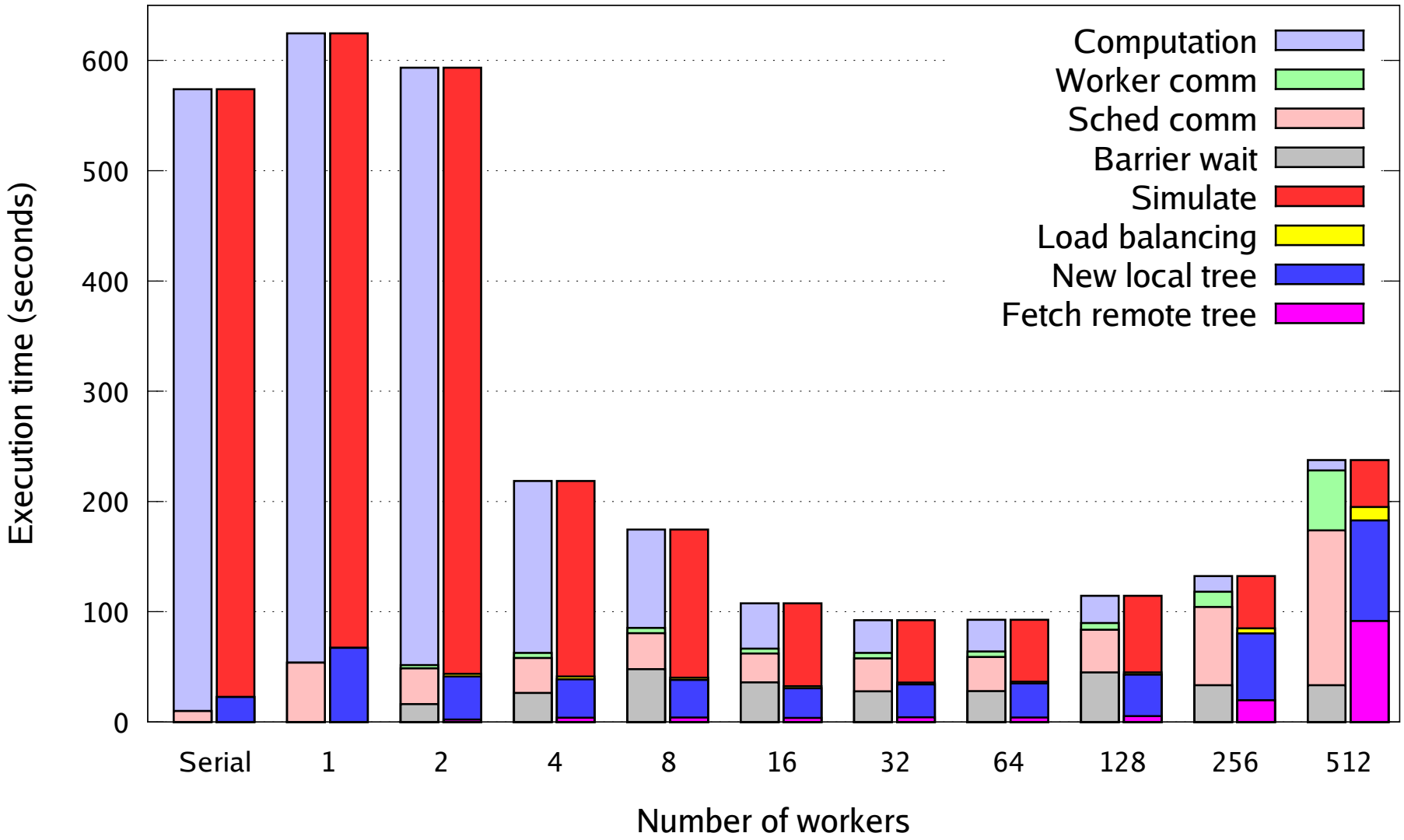
Hierarchical schedulers topology



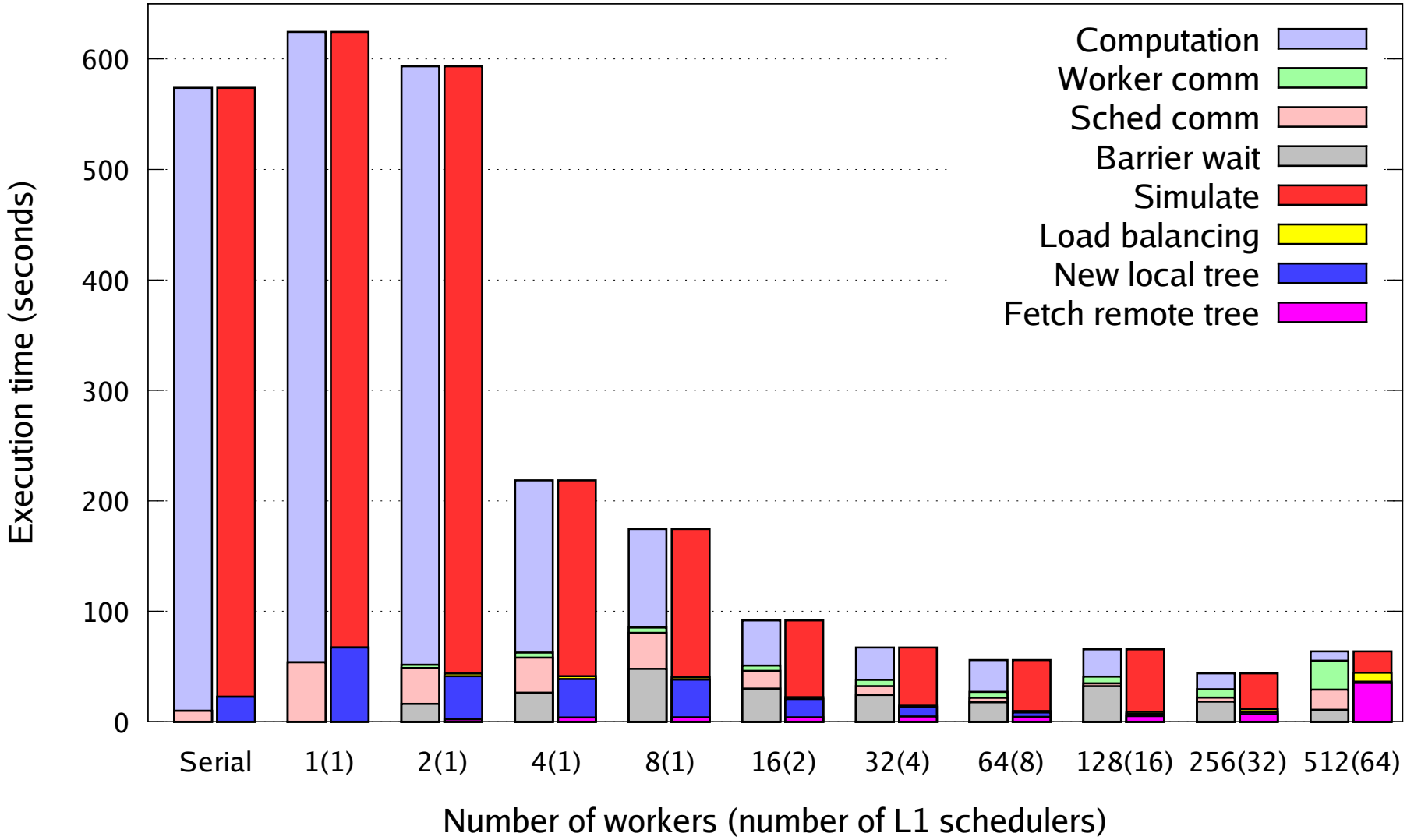
Splitting region trees for hierarchical scheduling



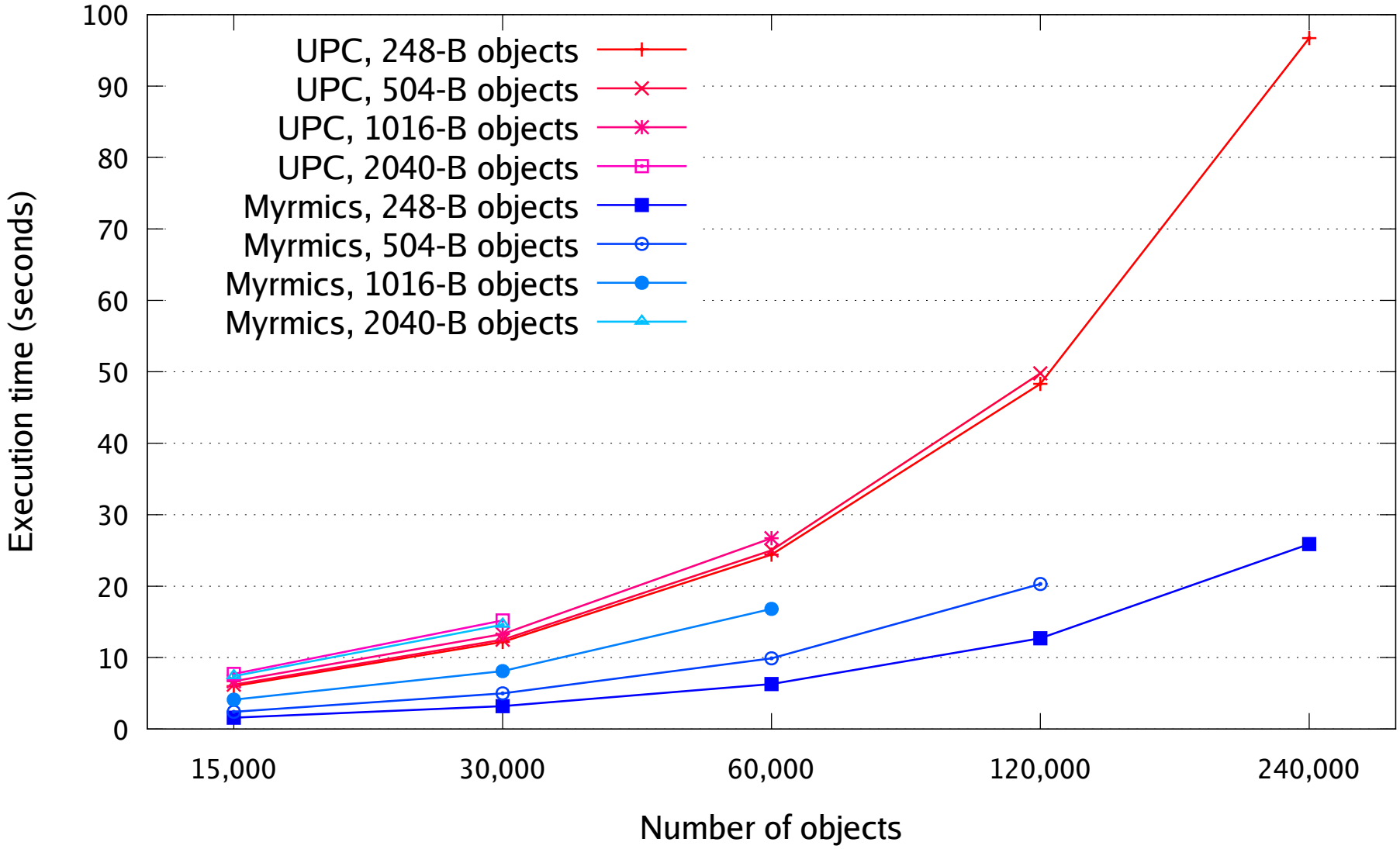
Barnes-Hut, 1.1M bodies, Single scheduler



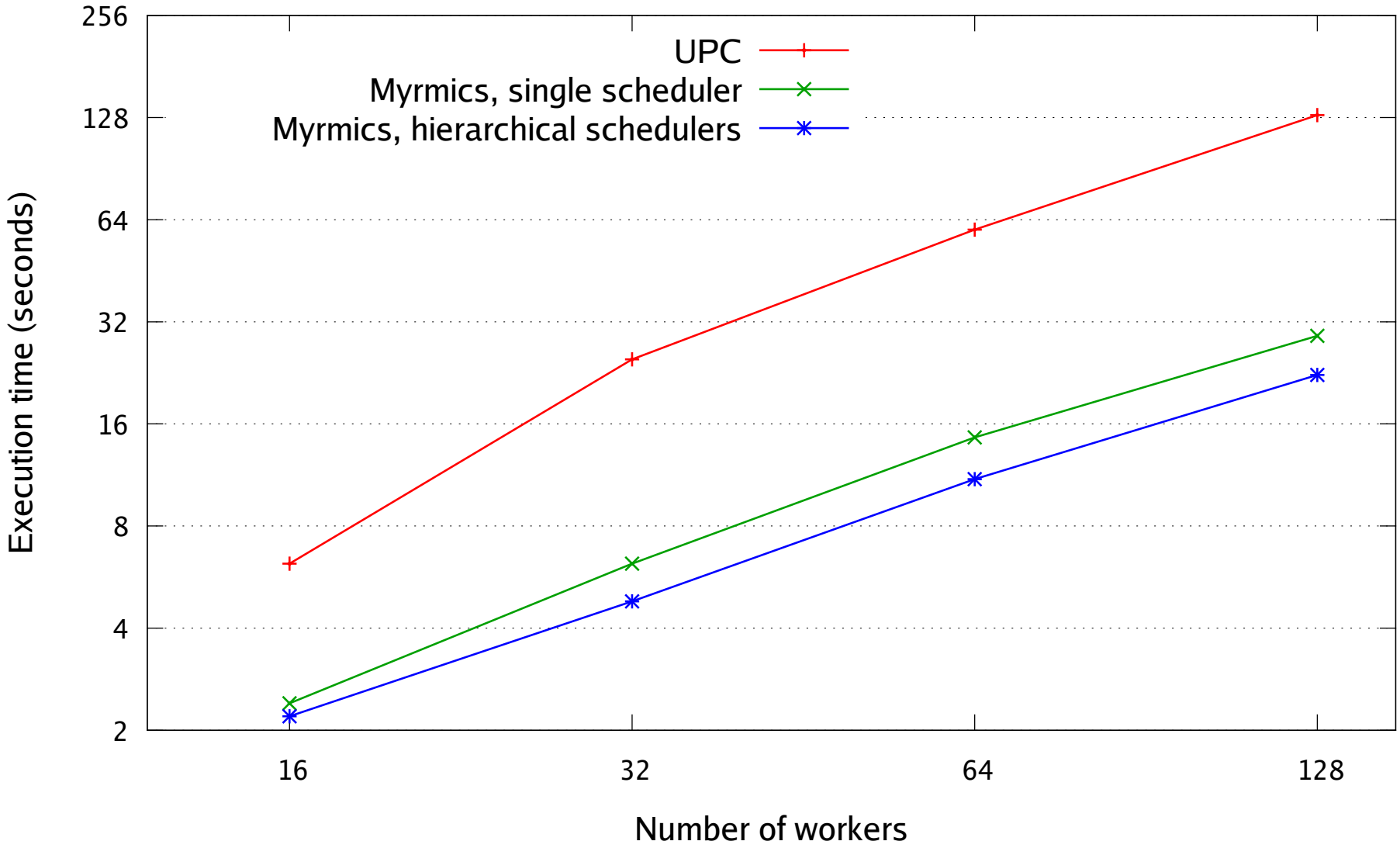
Barnes-Hut, 1.1M bodies, Two-level scheduler hierarchy



UPC comparison, 16 worker cores



UPC comparison, 15,000 504-B objects



Outline

Conclusions

- Regions language construct:
  - arbitrary task memory footprints, better productivity
  - shared memory programming abstractions
  - distributed memory execution semantics
  - scalability
- Work in progress:
  - Distributed memory task-based runtime (Myrmics)
  - Compiler support for regions (SCOOP)
  - Implementation and testing on FPGA prototype (Formic)
- Acknowledgments:

encore:

CARV (FORTH-ICS) Formic 512-core FPGA Prototype