What's Coming From the MM For KVM

Red Hat, Inc.

Andrea Arcangeli
aarcange at redhat.com

KVM Forum 2011
Vancouver, CA

15 Aug 2011

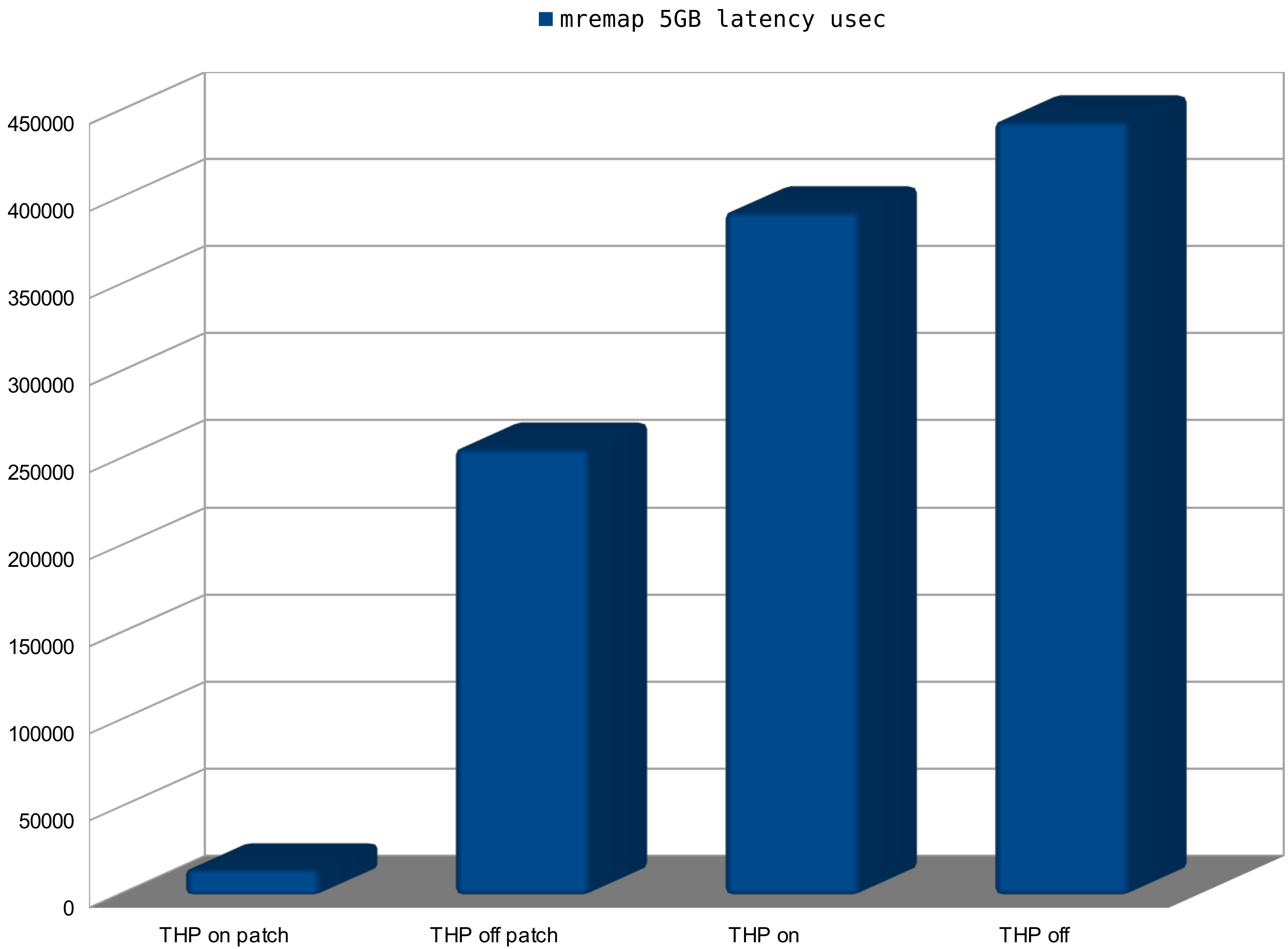
THP pending optimizations

- QEMU support for the 2/4MB mmap alignments is still missing
  - Mandatory to optimize for KVM (not as important without KVM except for the first and last 2/4MB)
    - Use qemu_memalign instead of qemu_vmalloc
- mremap() optimization (posted to linux-mm)
  - Boost THP and non-THP
    - As usual with THP the guest speedup is more significant than on the bare metal

QEMU THP alignment

```
@@ -2902,9 +2914,15 @@ ram_addr_t qemu_ram_alloc_from_ptr(DeviceState *dev, const char
*name,
                                    PROT_EXEC|PROT_READ|PROT_WRITE,
                                    MAP_SHARED | MAP_ANONYMOUS, -1, 0);
 #else
-            new_block->host = qemu_vmalloc(size);
+#ifdef PREFERRED_RAM_ALIGN
+           if (size >= PREFERRED_RAM_ALIGN)
+                   new_block->host = qemu_memalign(PREFERRED_RAM_ALIGN, size);
+           else
+#endif
+                   new_block->host = qemu_vmalloc(size);
 #endif
             qemu_madvise(new_block->host, size, QEMU_MADV_MERGEABLE);
+            qemu_madvise(new_block->host, size, QEMU_MADV_DONTFORK);
         }
     }
```

mremap(5GB) latency usec



Working set estimation

- Patches posted on linux-mm
- They walk pfn and call get_page_unless_zero() and then it walk the rmap of the page if a reference is obtained to mangle the accessed bit
  - Not safe to call that on THP tail pages
    - Proposed rework for the get_page()/put_page() to get a safe reference on tail pages
      - The rework slowdown get_page() on head pages (common case)
- It should be possible to solve it without slowing down get_page() on the head

Ballooning improvements

- The ballooning guest driver needs to become THP friendly
  - The guest should use compaction to release 2MB (or 4M on 32bit noPAE) of guest-physically-contiguous naturally aligned regions
- The working set estimation algorithm worked on by Google in the host (for soft-limits in cgroups) could drive the balloon driver automatically
  - aka auto-ballooning
  - Page hinting is an alternative to this

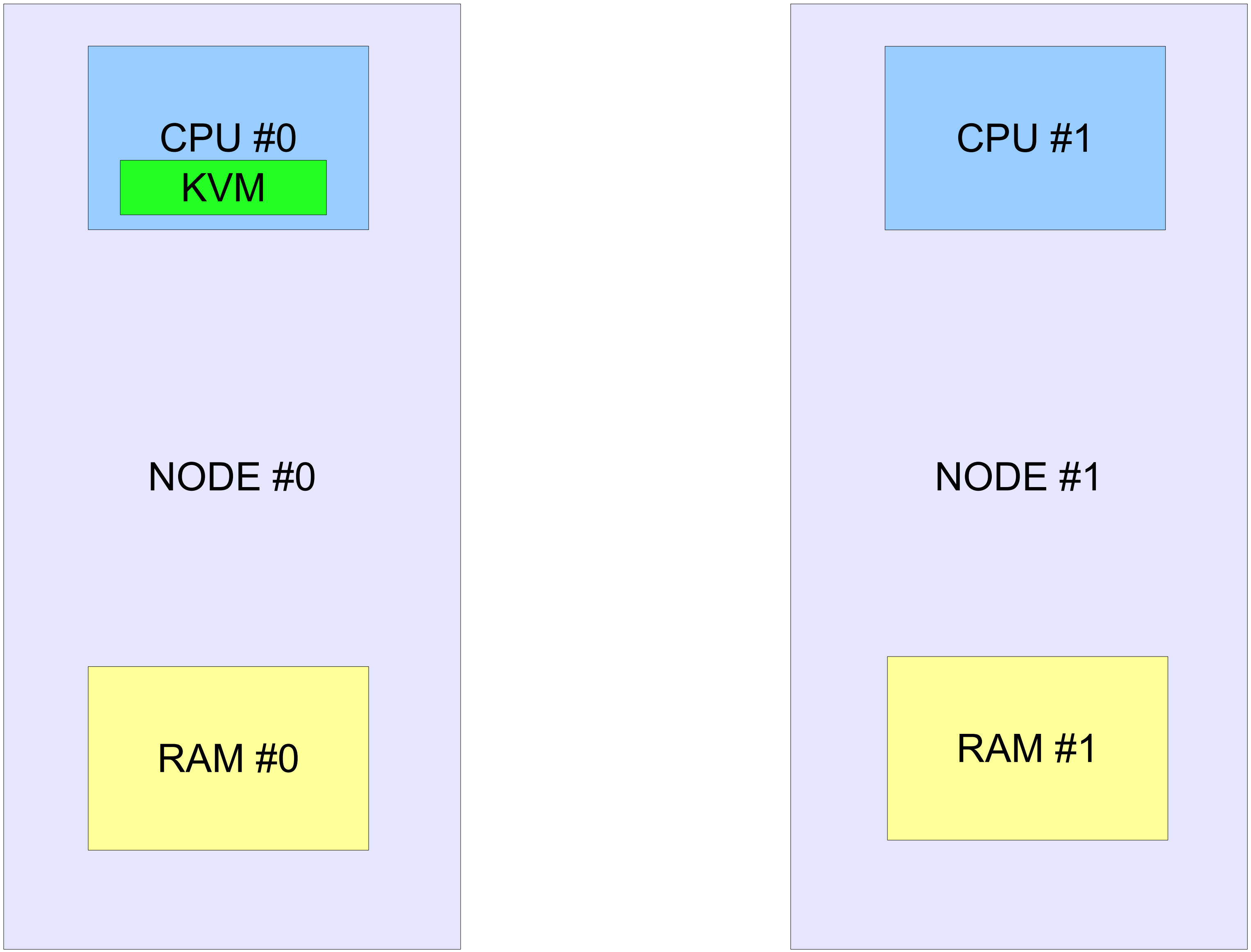
KSM using dirty bit

- A patch is available to make KSM use the dirty bit to detect “frequently changing memory” that is not worth trying to merge
- Detects equal overwrite too
- Problem: no dirty bit in EPT
  - So for the time being it's not very useful for KVM
  - Flushing the dirty bit from the TLB is also not cheap with several vCPUs
  - It reduces the CPU load for the scanning but it may slowdown the guests a bit
- We may consider it in the future

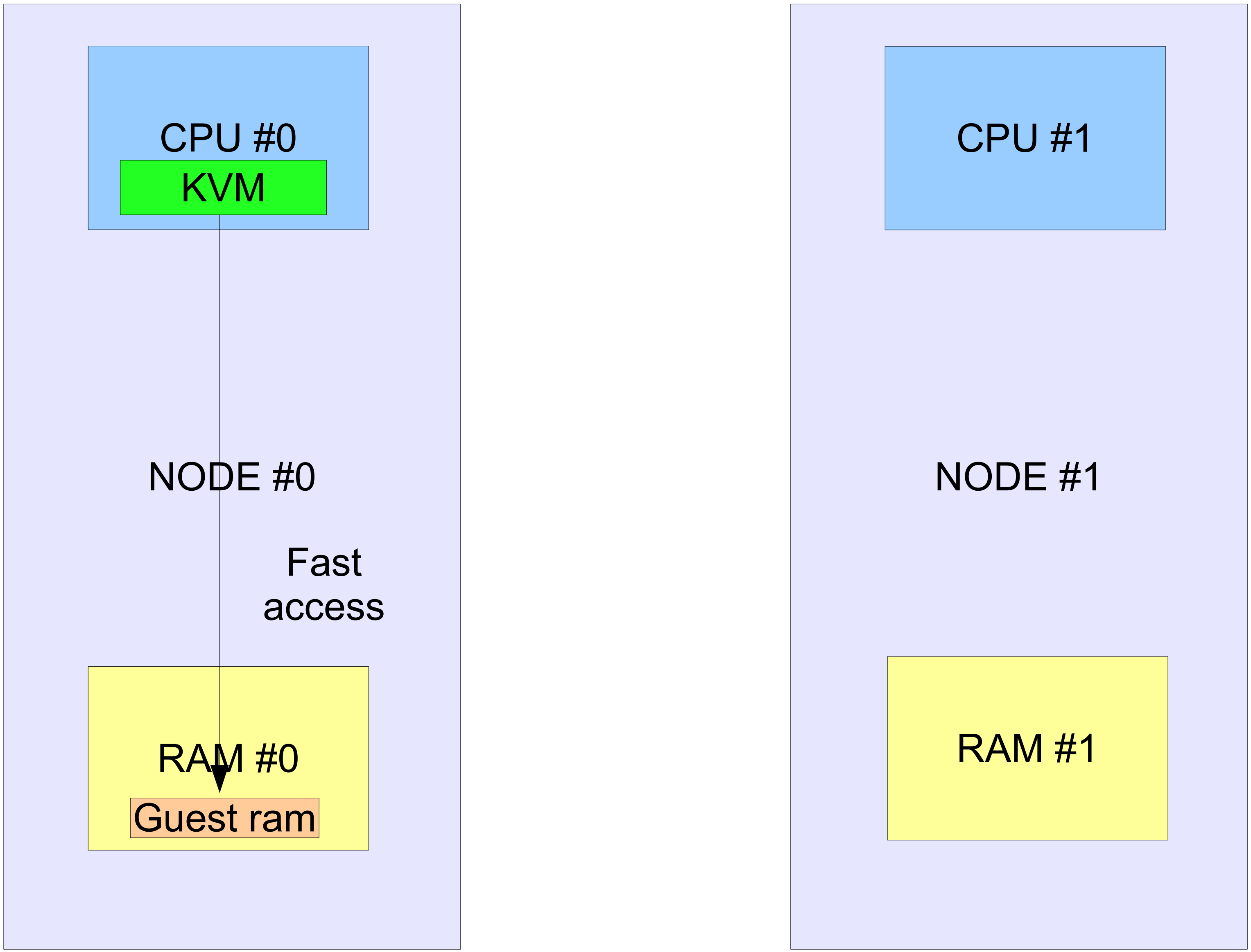
KVM NUMA awareness

- I.e. making Linux NUMA aware
- The Linux Scheduler currently is blind about the memory placement of the process
- MPOL_DEFAULT allocates memory from the local node of the current CPU
- It all works well if the process isn't migrated by the scheduler to a different NUMA node later
  - Or if the memory gets full in the local node and the memory allocation spills on other nodes
- Short lived tasks (like gcc) are handled pretty well

KVM startup on CPU #0

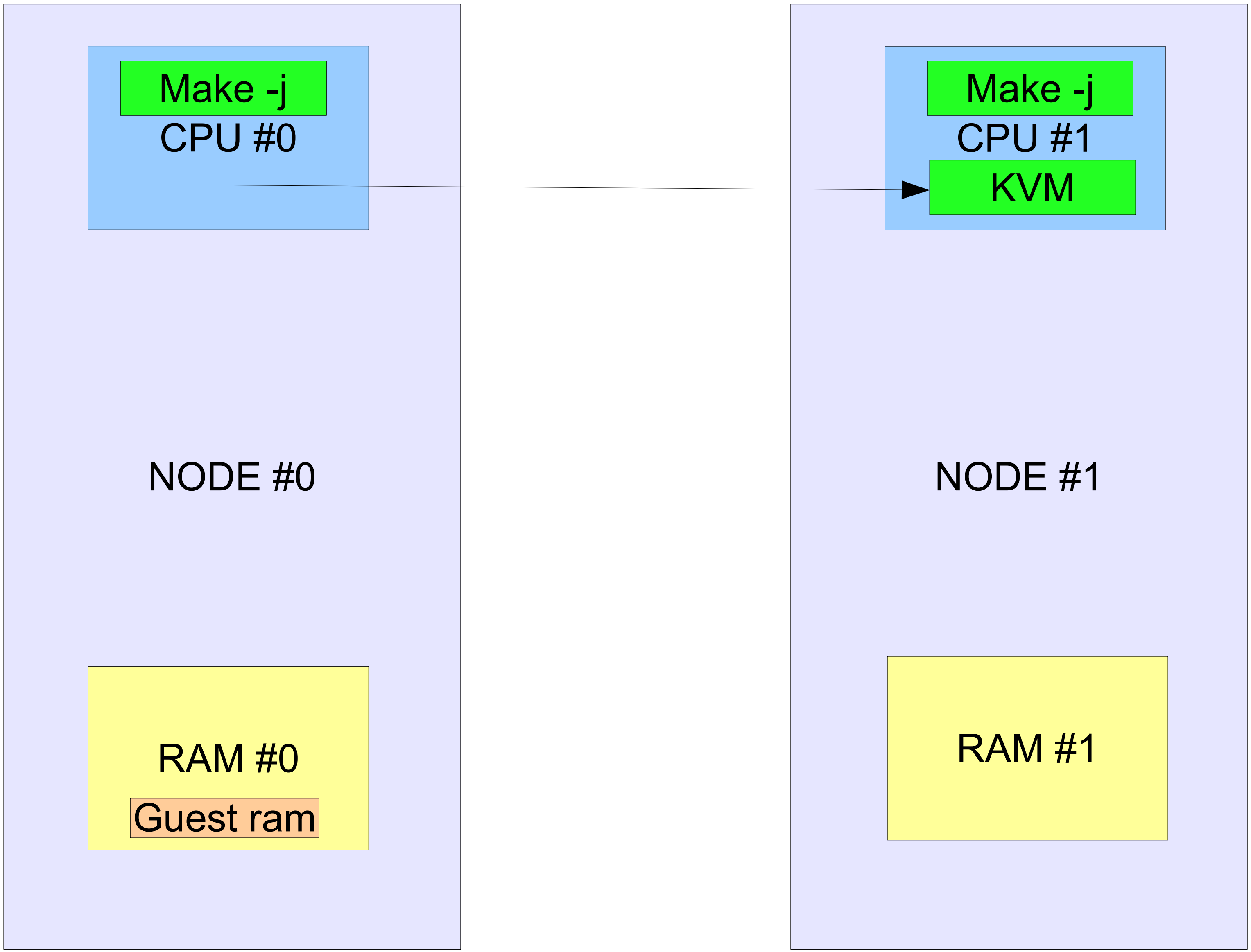


KVM allocates from RAM #0

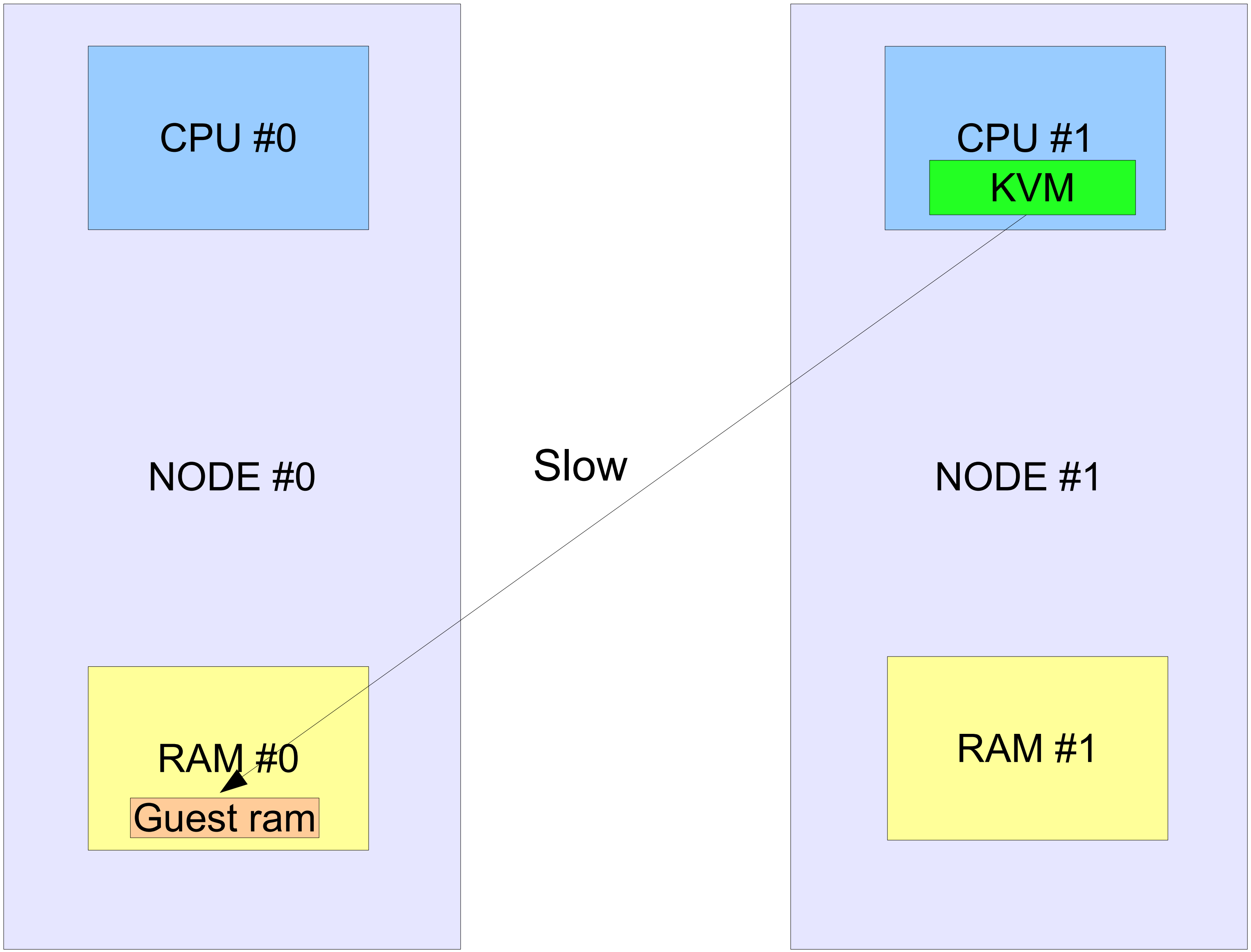


No NUMA hard bindings and MPOL_DEFAULT policy

Scheduler CPU migration



“make -j” load goes away



The Linux Scheduler is blind at this point: **KVM may stay in CPU #1 forever**

The scheduler is memory blind

- Short lived tasks are ok
- Long lived tasks like KVM can suffer badly from using remote memory for extended periods of times
  - Because they live longer, they're more likely to be migrated if there's some CPU overcommit
- It's fairly cheap for the CPU to follow the memory
- We would like the CPU to follow the memory
  - CPU placement based on memory placement
- We would like to achieve the same performance of the NUMA bindings without having to use them

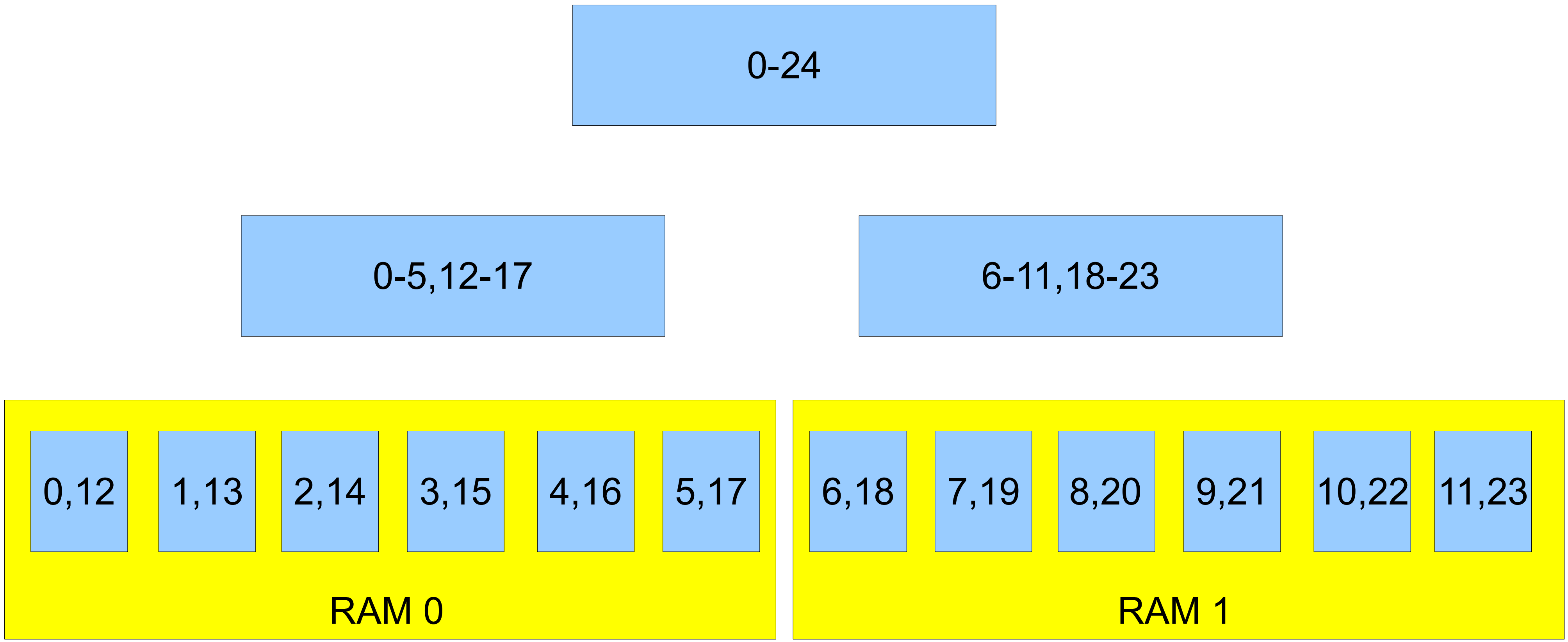
redhat

What we have today

- Hard NUMA bindings
  - sys_mempolicy
  - sys_mbind
  - sys_sched_setaffinity
  - sys_move_pages
  - /dev/cpuset
    - Job manager can monitor memory pressure and act accordingly
- All depends on numbers taken for example from the next slide to split the machine resources
- Full topology available in /sys

redhat

Scheduler domains



Example of a common 2 nodes, 2 sockets, 12 cores, 24 threads system

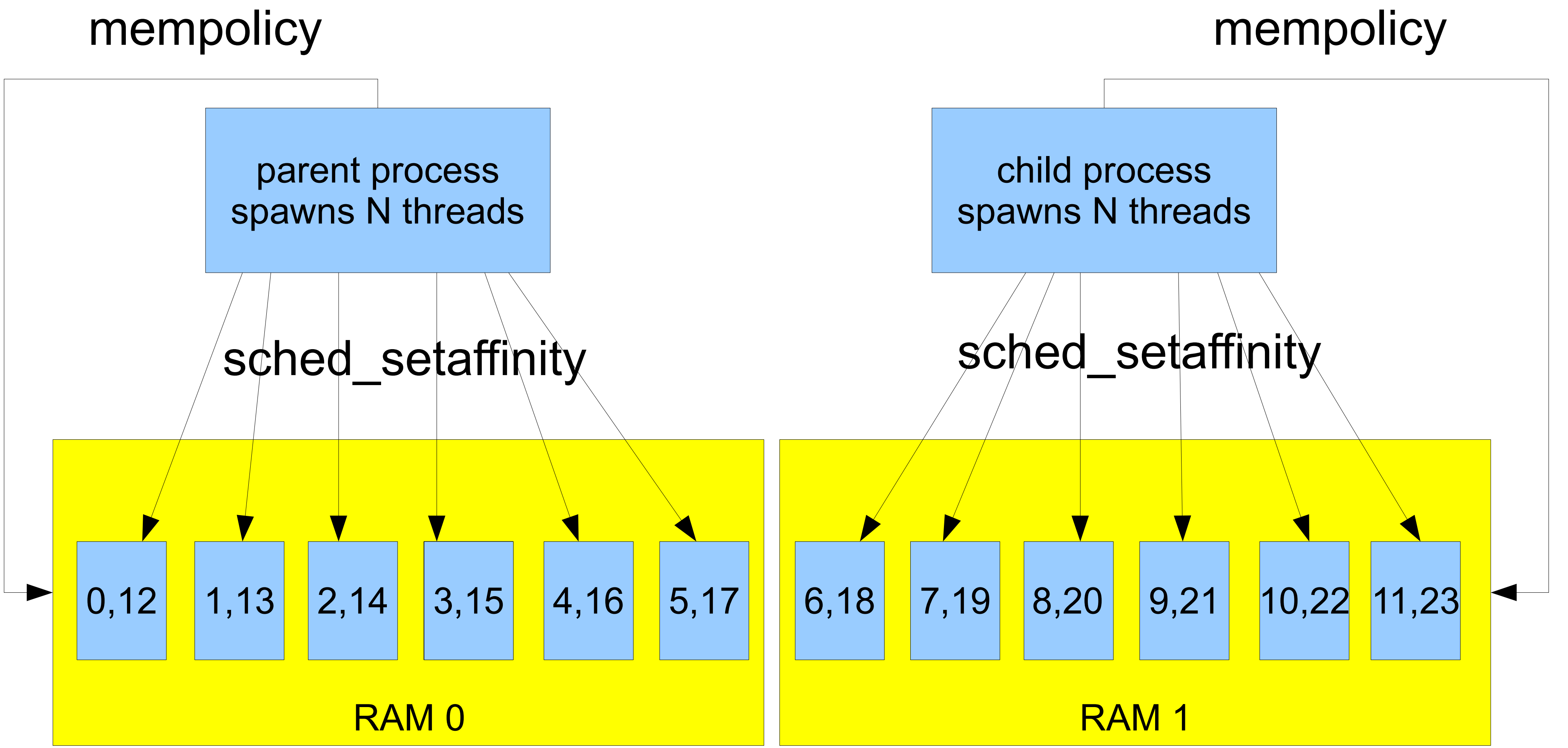
Hard bindings and hypervisors

- Cloud nodes powered by virtualization hypervisors
  - Dynamic load
    - VM started/shutdown/migrated
    - Variable amount of vRAM and vCPUs
  - A job manager can do a static placement
    - But not as efficient to tell which vCPUs are idle and which memory is important for each process/thread at any given time
  - The host kernel probably can do better at optimizing a dynamic workload

How bad is remote RAM? (bench)

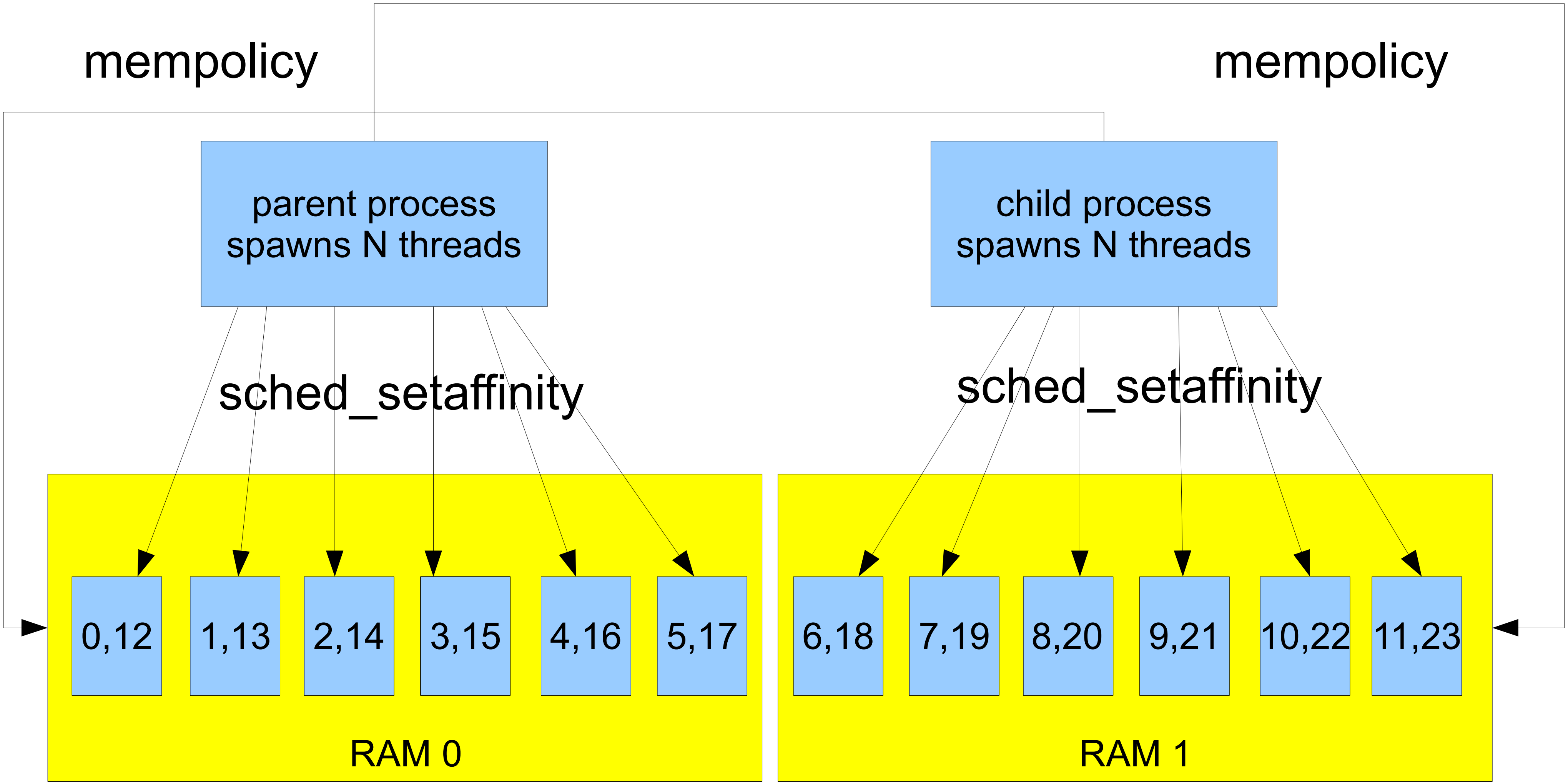
```
#define SIZE (6UL*1024*1024*1024)
#define THREADS 24

void *thread(void * arg)
{
	char *p = arg;
	int i;
	for (i = 0; i < 3; i++) {
		if (memcmp(p, p+SIZE/2, SIZE/2))
			printf("error\n"), exit(1);
	}
	return NULL;
}
[..]
	if ((pid = fork()) < 0)
		perror("fork"), exit(1);
[..]
#ifdef 1
	if (sched_setaffinity(0, sizeof(cpumask), &cpumask) < 0)
		perror("sched_setaffinity"), exit(1);
#endif
	if (set_mempolicy(MPOL_BIND, &nodemask, 3) < 0)
		perror("set_mempolicy"), printf("%lu\n", nodemask), exit(1);
	bzero(p, SIZE);
	for (i = 0; i < THREADS; i++)
		if (pthread_create(&pthread[i], NULL, thread, p) != 0)
			perror("pthread_create"), exit(1);
	for (i = 0; i < THREADS; i++)
		if (pthread_join(pthread[i], NULL) != 0)
			perror("pthread_join"), exit(1);
```

mempolicy + setaffinity local



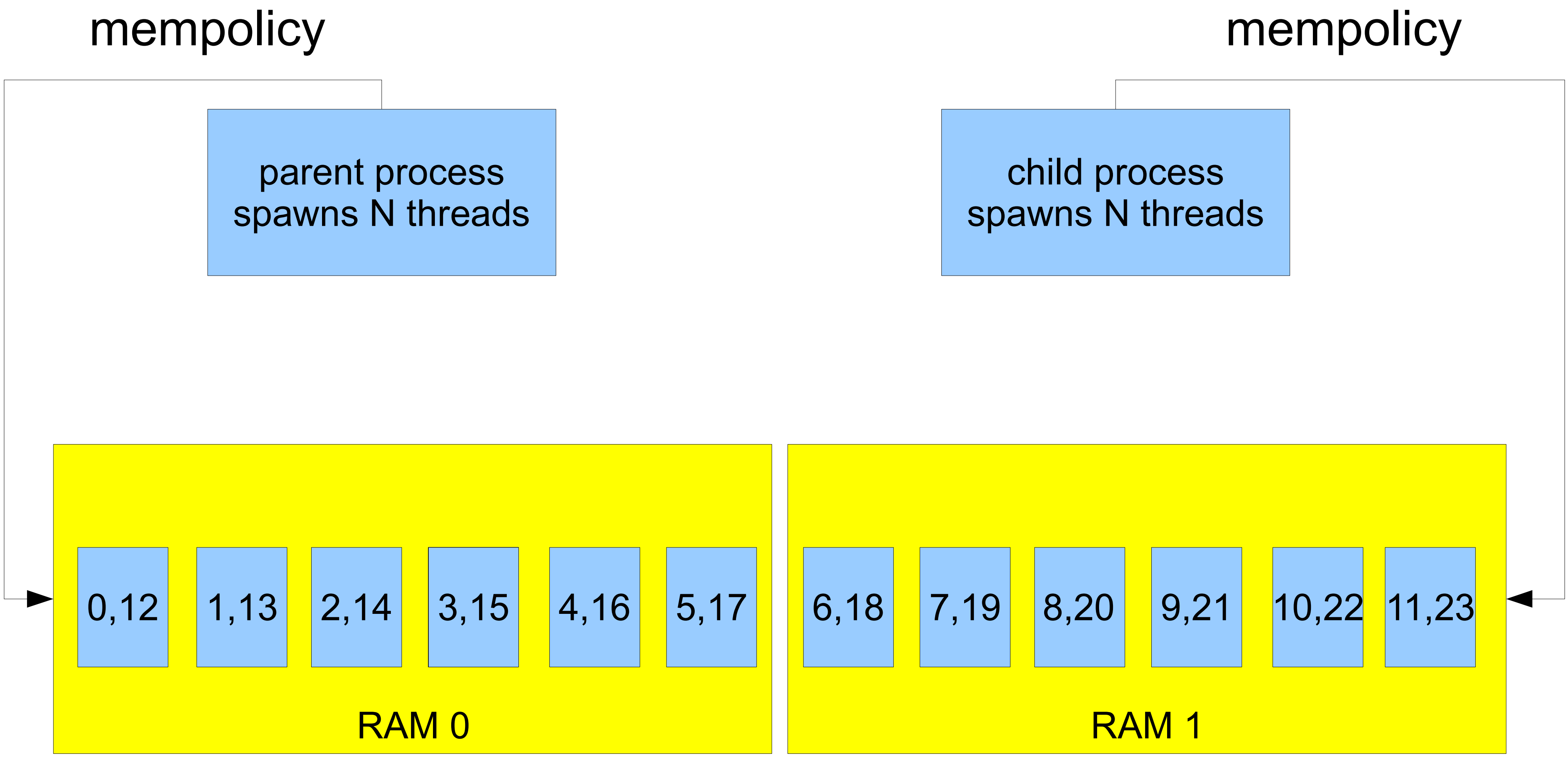
Best possible CPU/RAM NUMA placement
All CPUs only work on local RAM

mempolicy + setaffinity remote



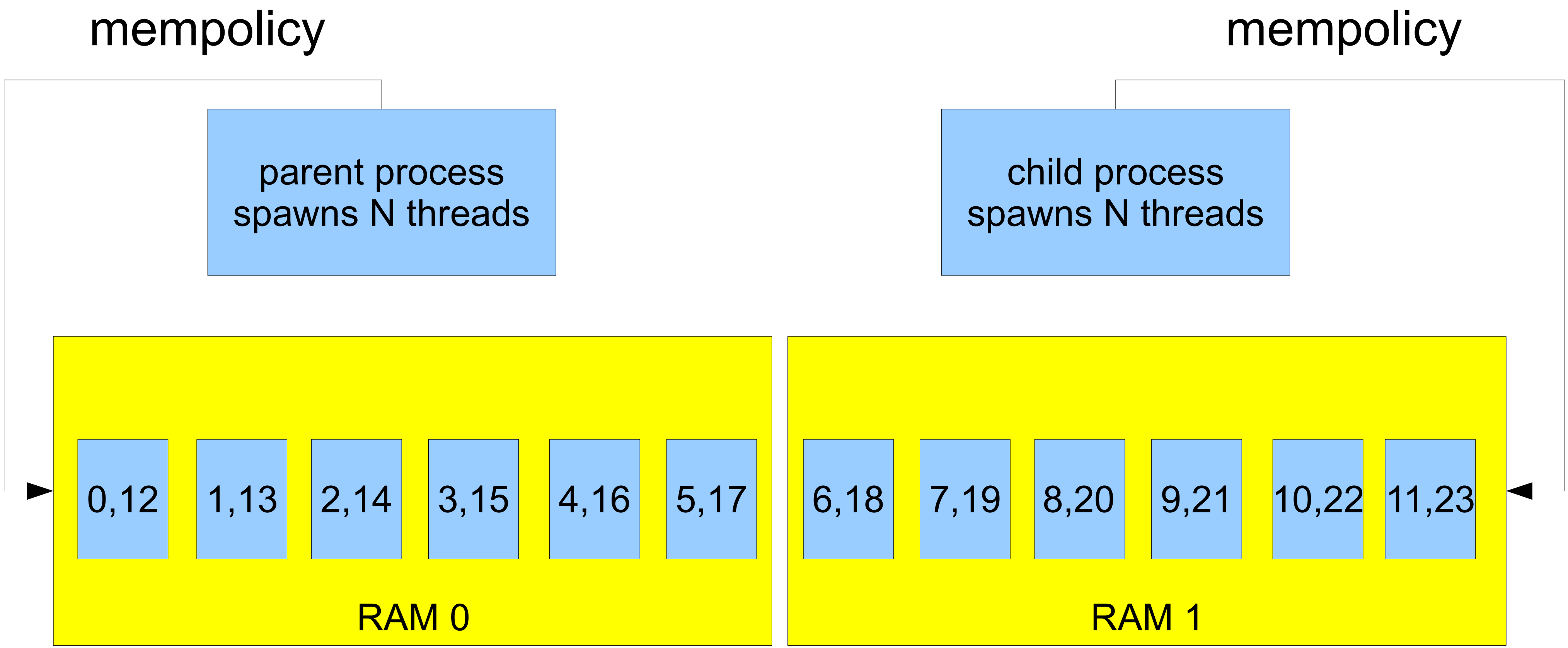
Worst possible CPU/RAM NUMA placement
All CPUs only work on remote RAM

Only mempolicy



Only RAM NUMA binding with mempolicy()
The host CPU scheduler can move all threads anywhere
The CPU scheduler has no memory awareness

Mempolicy + CPU-follow-memory

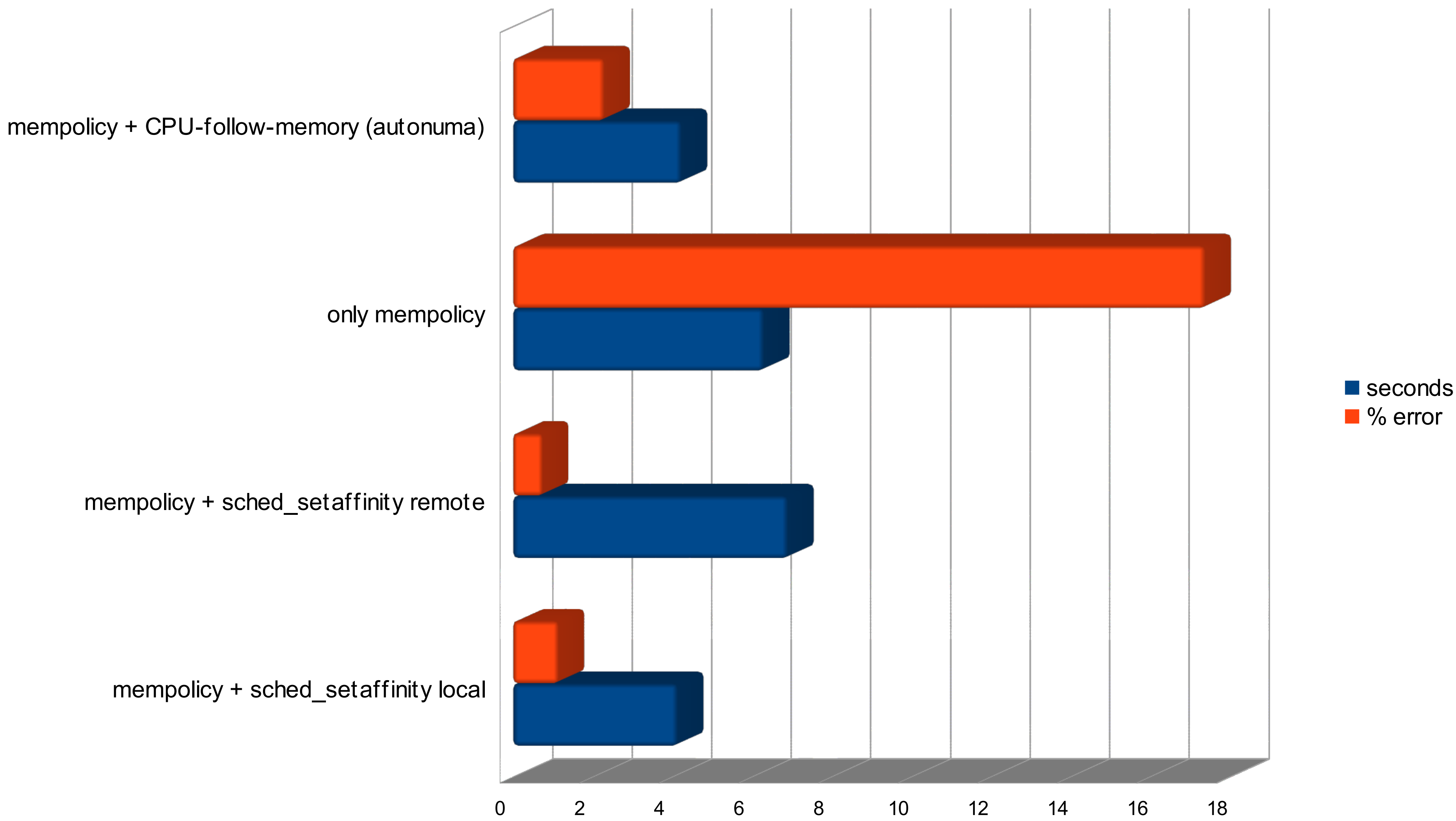


The host CPU scheduler understand the parent process has most of the RAM allocated in NODE 0 and the child in NODE 1

No scheduler hints from userland

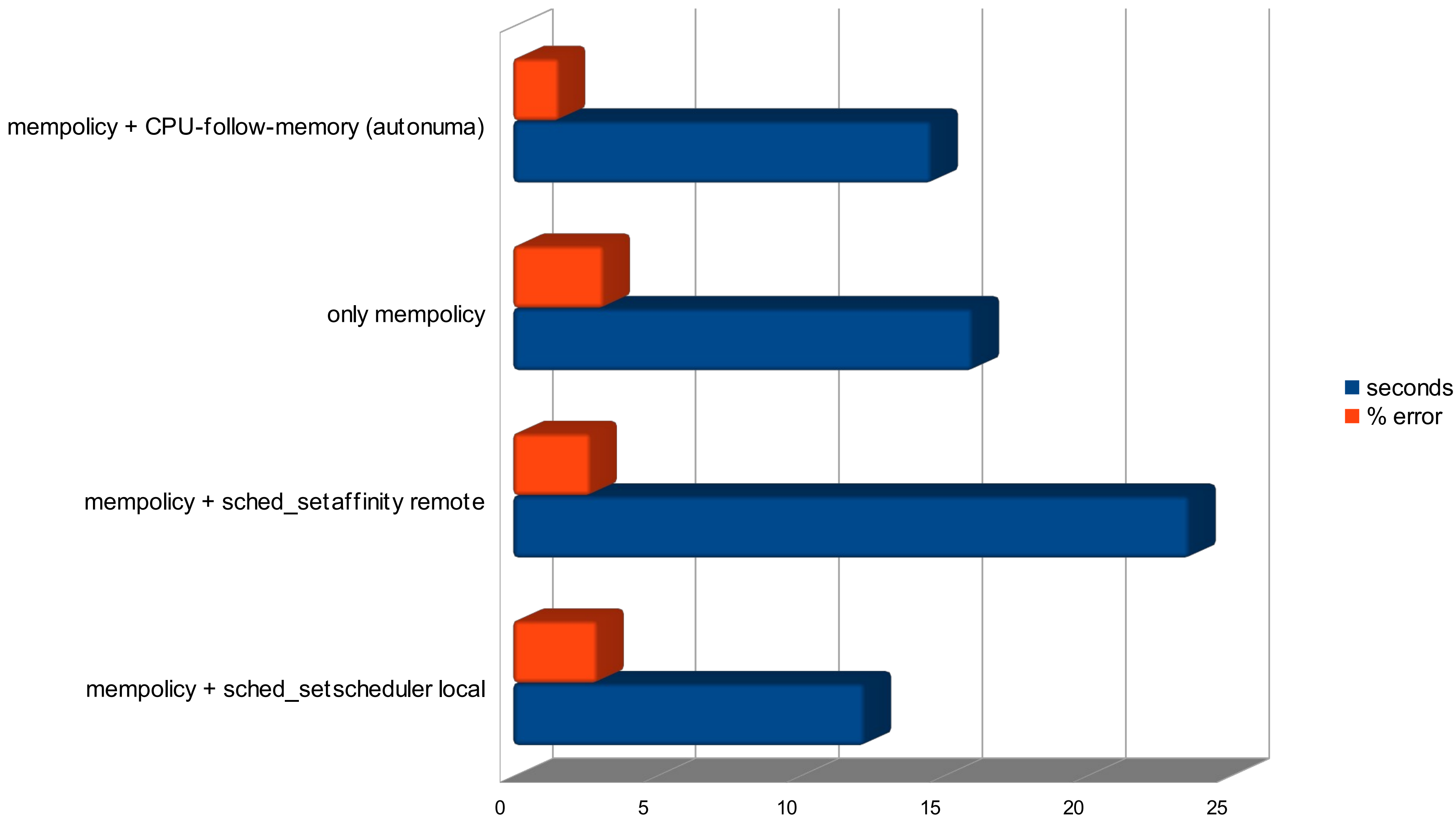
Mempolicy() doesn't have any scheduler effect

1 thread x 2 processes



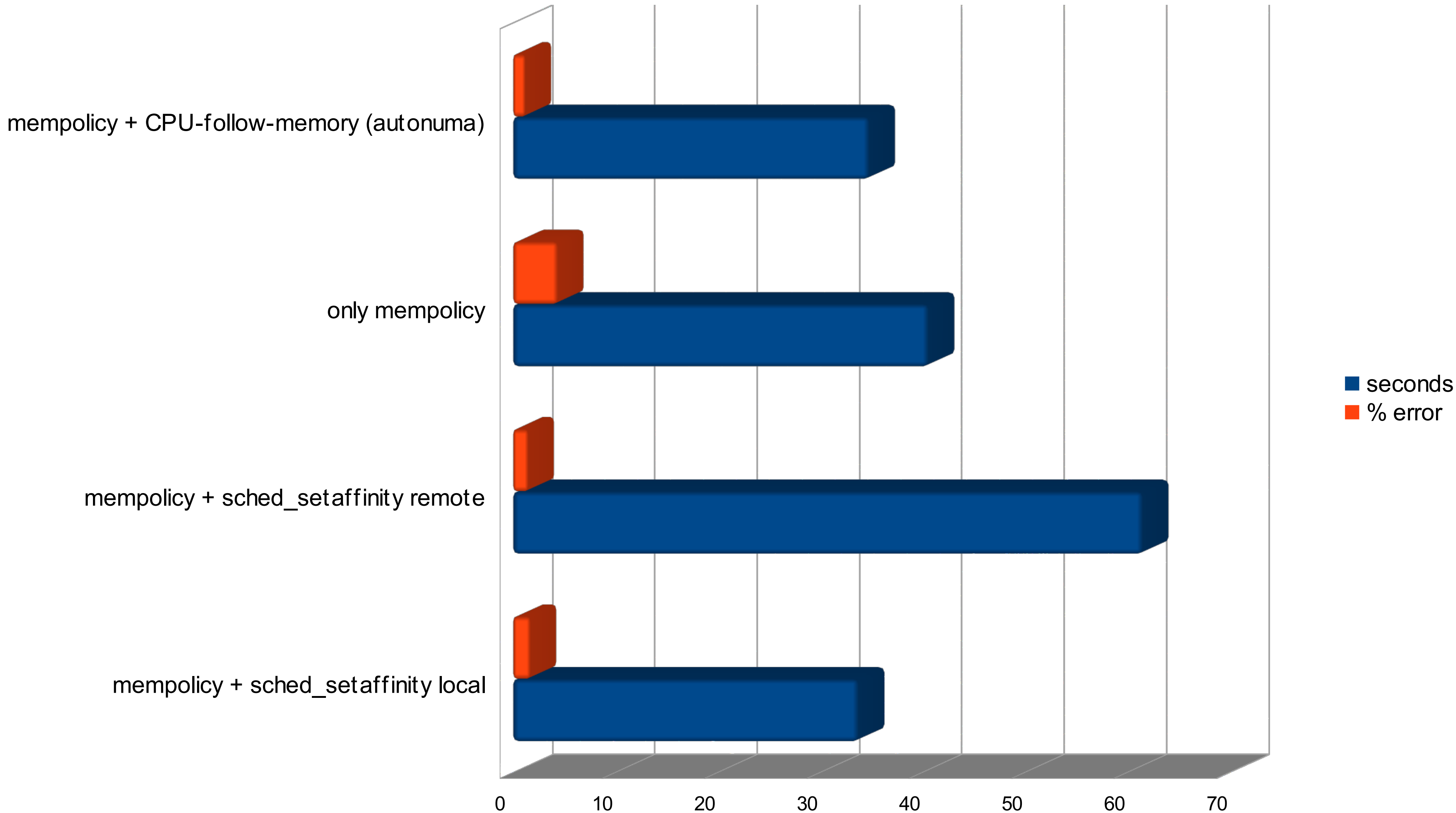
Only 2 CPUs used, 2 nodes 2 sockets 12 cores 24 threads

12 threads x 2 processes



All 24 CPUs maxed out, 2 nodes 2 sockets 12 cores 24 threads

24 threads x 2 processes



Double CPU overcommit, 2 nodes 2 sockets 12 cores 24 threads

CPU-follow-memory

- Implemented as a proof of concept
  - For now only good enough to proof that it performs equivalent to sched_setaffinity()
- CPU-follow-memory not enough
  - We still run a sys_mempolicy!
- Must be combined with memory-follow-CPU
- When there are more threads than CPUs in the node things are more complex
  - “mm” tracking not enough: vma/page per-thread tracking needed (not trivial to get that info without page faults)

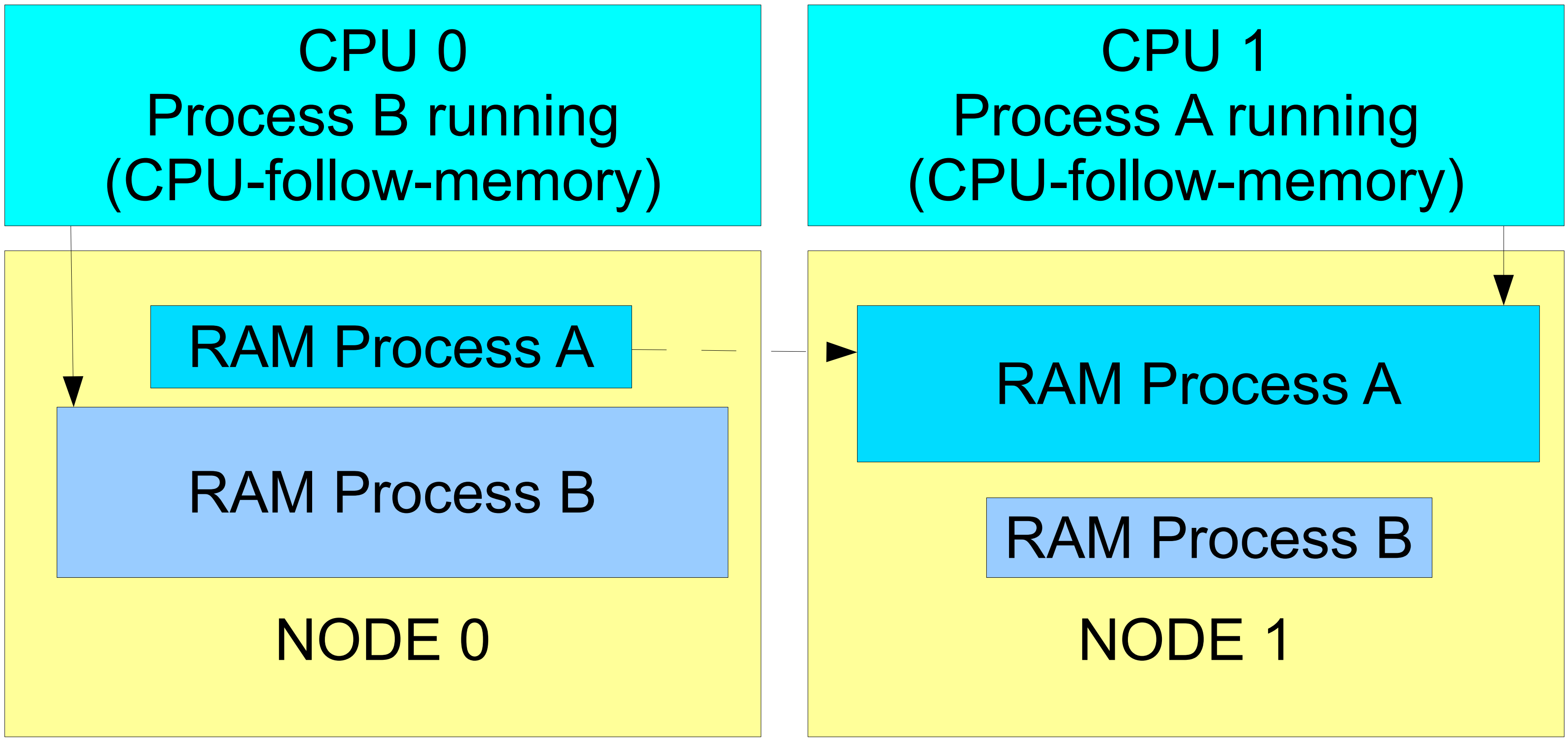
memory-follow-CPU

- Converge the RAM of the process into the node where it's running on by migrating it in the background
- If CPU-follow-memory doesn't follow memory because of too high load in the preferred nodes
  - Migrate the memory of the process to the node where the process is really running on and converge there
  - Have CPU-follow-memory temporarily ignore the current memory placement and follow CPU instead until we converged

Auto NUMA memory migration

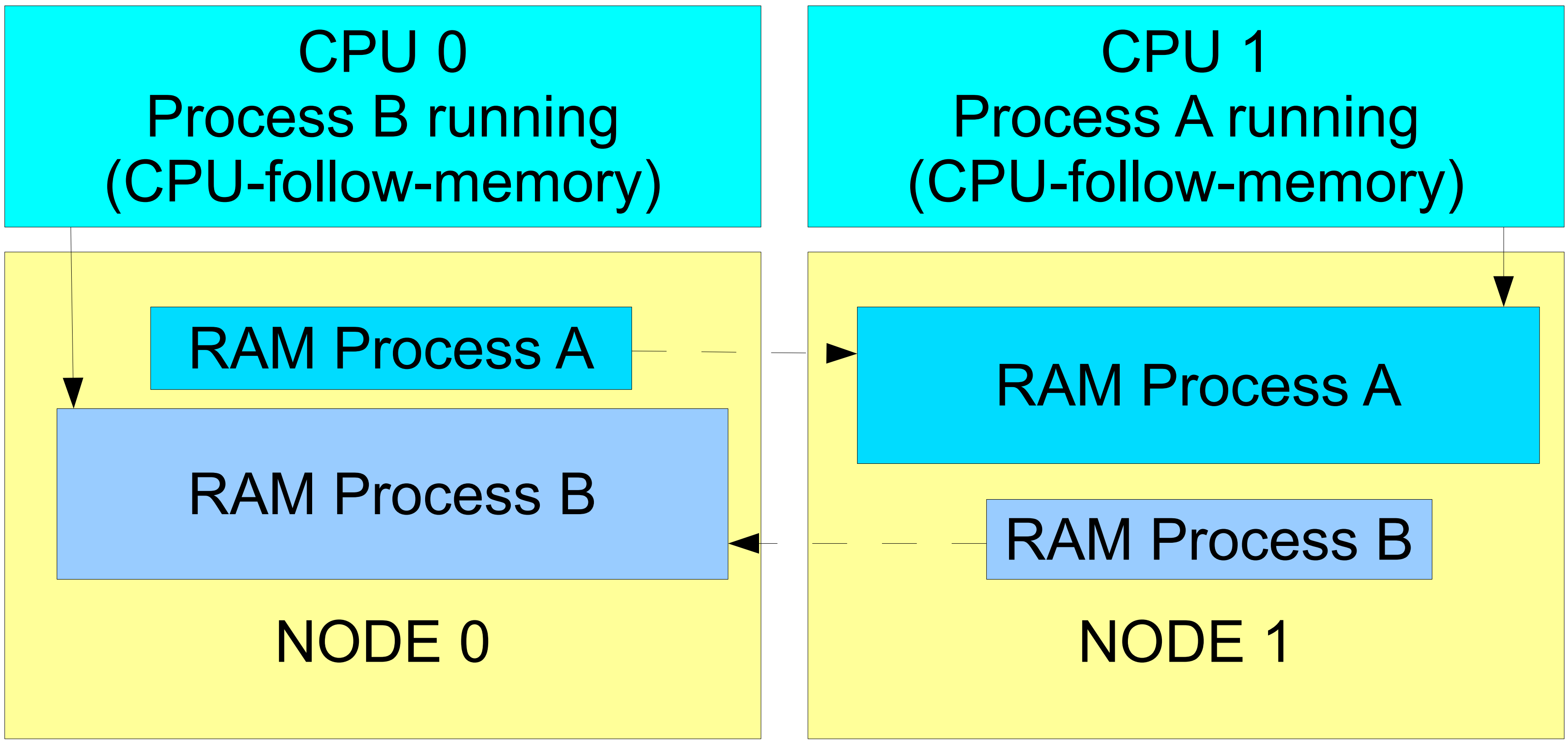
- We need to find a process that has RAM in NODE 1 and wants to converge into NODE 0, in order to migrate the RAM of another process from NODE 0 to NODE 1
  - This will keep the memory pressure balanced
  - Pagecache/swapcache/buffercache may be migrated as fallback but active process memory should be preferred to get double benenfit
- Memory-follow-CPU migrations should concentrate on processes with high CPU utilization
- The migrated memory ideally should be in the working set of the process

Auto NUMA memory migration



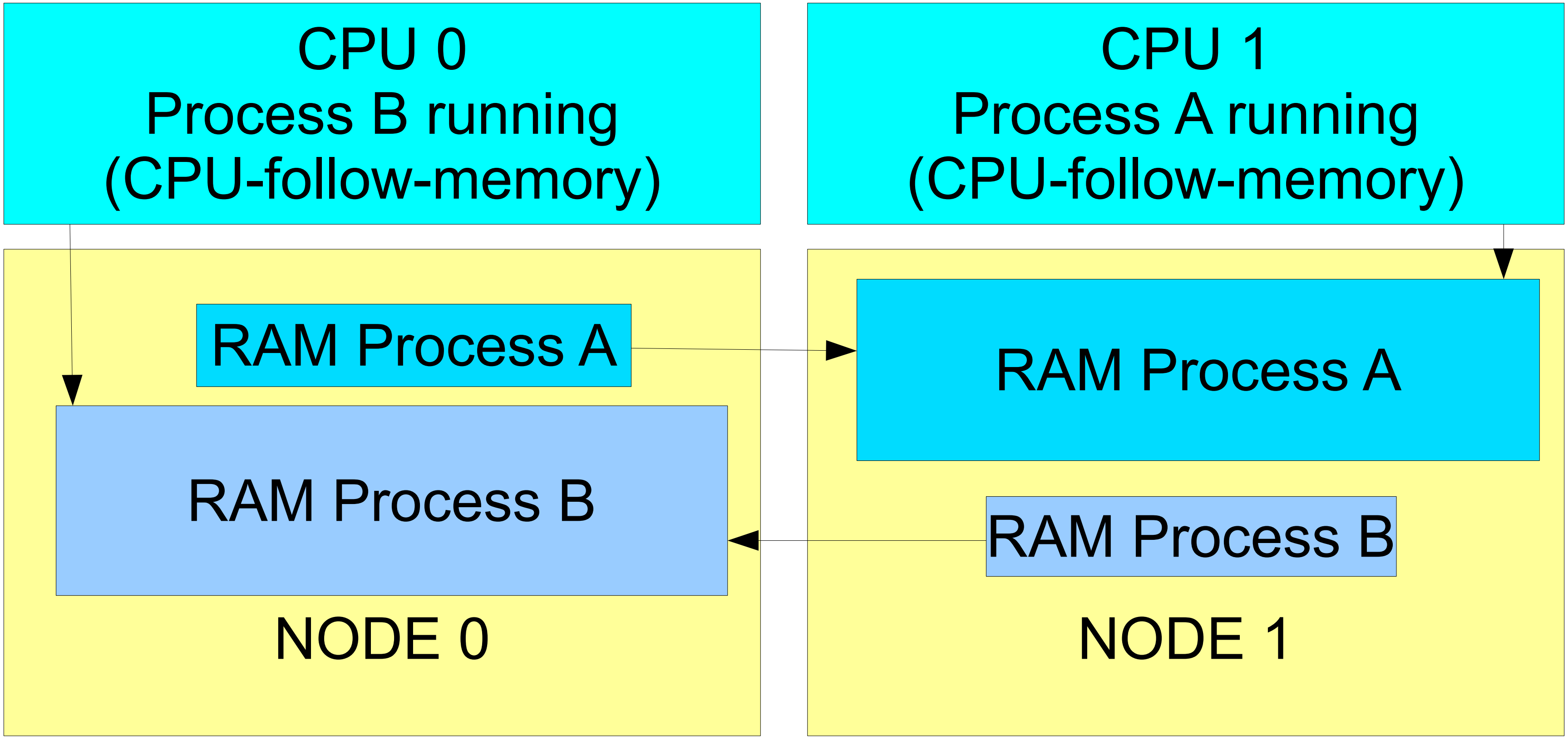
memory-follow-CPU wants to migrate the RAM of Process A from NODE0 to NODE 1

Auto NUMA memory migration



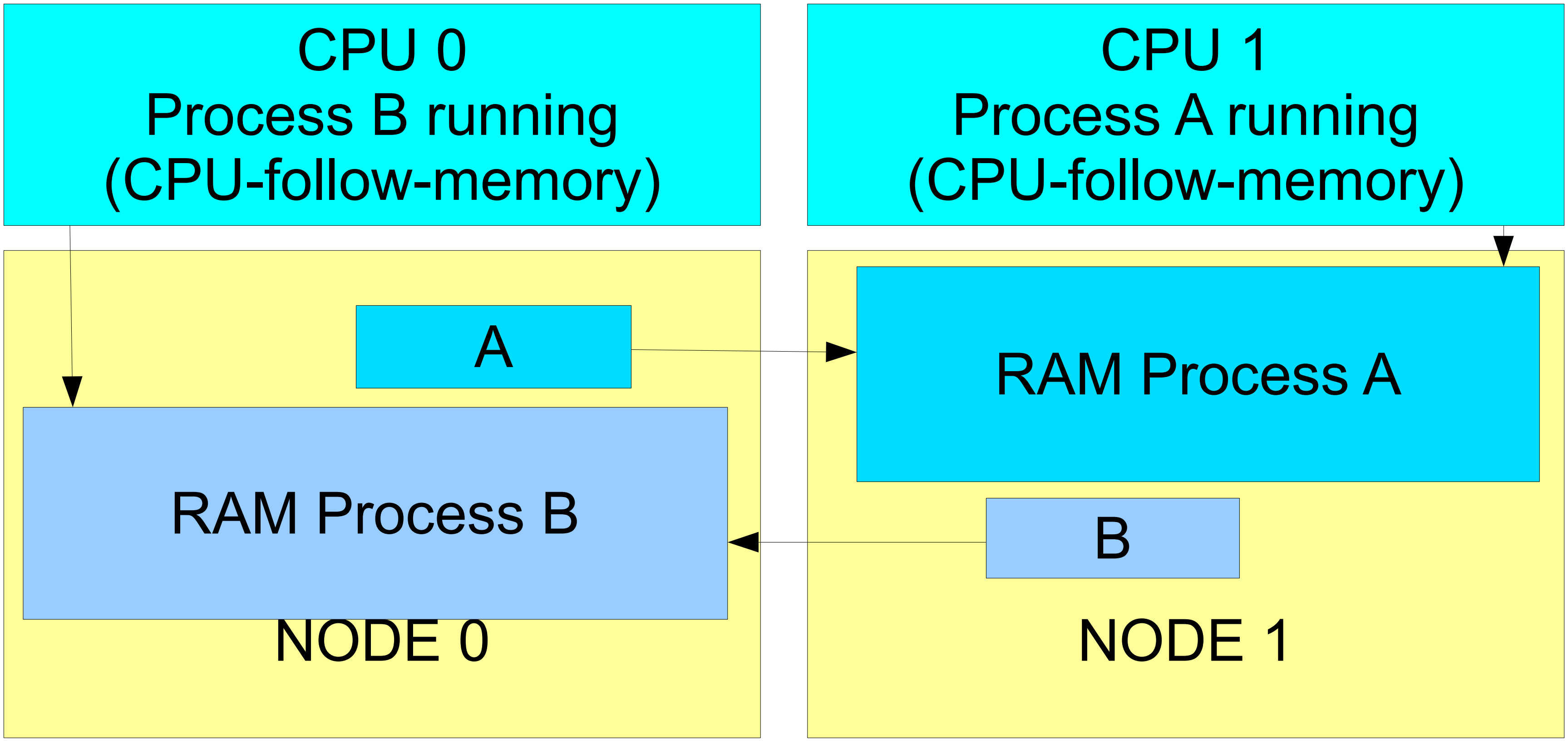
memory-follow-CPU need to find another process
with memory on NODE 1 that wants to migrate to NODE 0
Process B is ideal

Auto NUMA memory migration



memory-follow-CPU migrates the memory...

Auto NUMA memory migration



memory-follow-CPU repeats...

knumad

- CPU-follow-memory is currently entirely fed with information from a knumad kernel daemon that scans the process memory in the background
- It could be changed to static accounting to help short lived tasks too
  - There's a time-lag from when memory is first allocated and when CPU-follow-memory notices (this explains the slight slower perf)
    - Initially, when no memory information exists yet, MPOL_DEFAULT is used
- knumad may later drive memory-follow-CPU too
- Working set estimation is possible

redhat

Anonymous memory

- knumad only considers not shared anonymous memory
  - For KVM it is enough
  - This will likely have to change
  - It'll be harder to deal with CPU/RAM placement of shared memory

Per-thread information

- The information in the pagetables is per-process
- To know which part of the process memory each thread is accessing there are various ways
  - … or old ways like forcing page faults
    - Migrate-on-fault does that
    - Migrate-on-fault heavyweight with THP
    - Migrating memory in the background should be better than migrate-on-fault because it won't always hang the process during migrate_pages()

Another way: soft NUMA bindings

- Instead of setting hard numbers like 0-5,12-17 and node 0 manually we could create a soft API:

  ```
  numa_group_id = numa_group_create();
  numa_group_mem(range, numa_group_id);
  numa_group_task(tid, numa_group_id);
  ```

- This would allow to easily create a vtopology for the guest by changing QEMU
- It would not require special tracking as QEMU would specify which vCPUs belong to which vNODE to the host kernel.
- But if the guest spans more than one host node, all guest apps should use this API too...

Soft NUMA bindings

- I think a full automatic way should be tried first...
  - Full automatic NUMA awareness requires more intelligence on the kernel side
- Cons of soft NUMA bindings:
  - APIs must be maintained forever
  - APIs don't solve the problem of applications not NUMA aware
  - Not easy for programmer to describe to the kernel which memory each thread is going to access more frequently
    - Trivial for QEMU, but not so much for other users

Q/A

- You're very welcome!