Hyperthreading + Multicore + NUMA

Red Hat, Inc.

Andrea Arcangeli
aarcange at redhat.com

Architectures, tools and methodologies for developing efficient large
scale scientific computing applications

Bertinoro, Italy

28 Oct 2011

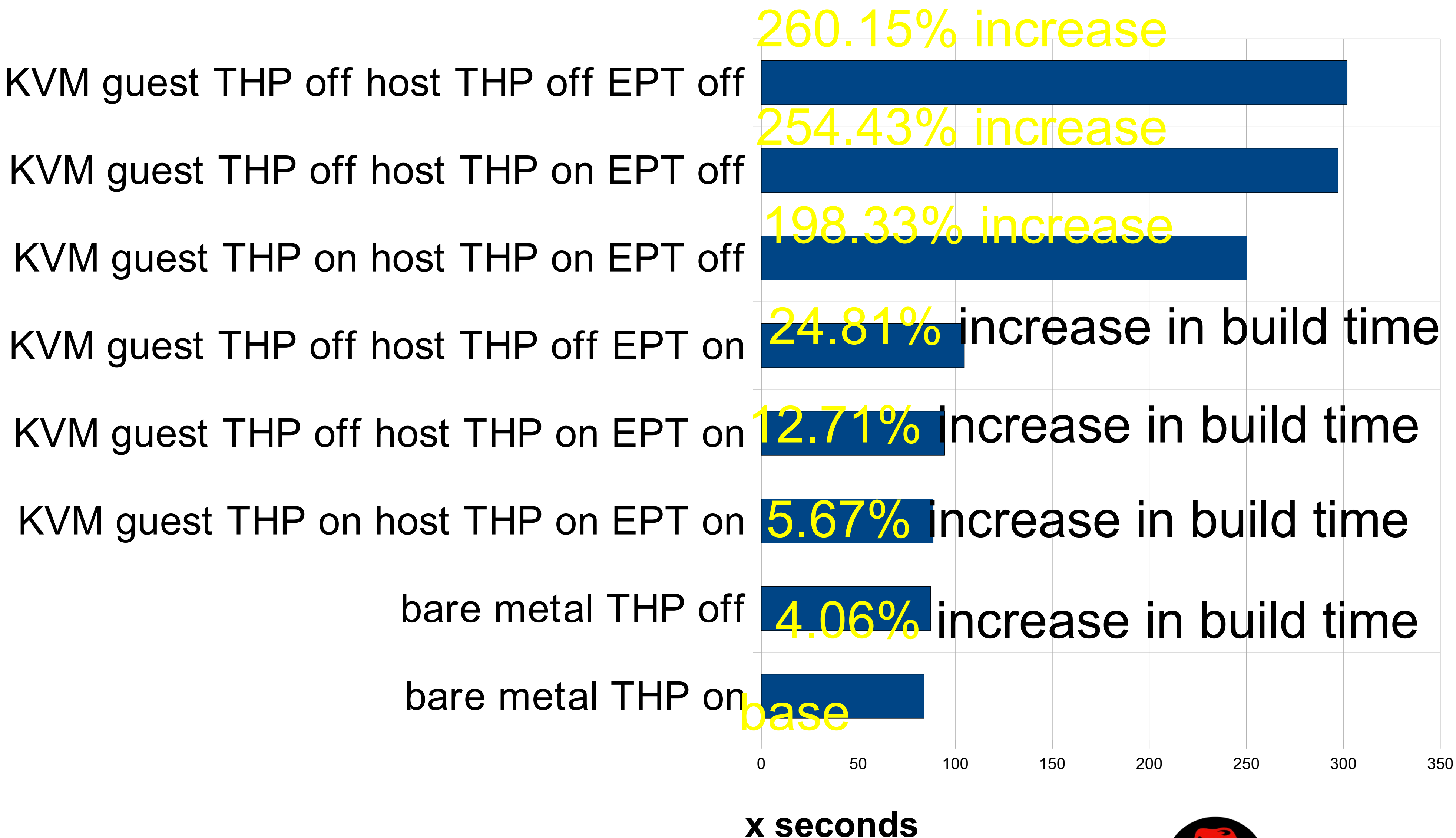
THP

- If your HPC program uses lots of anonymous memory (i.e. malloc) you absolutely need THP
- Performance and scalability boost for Virt & HPC
- To be sure hugepages are allowed in hardware use:
  - posix_memalign(&ptr, 2*1024*1024, 2*1024*1024*N)

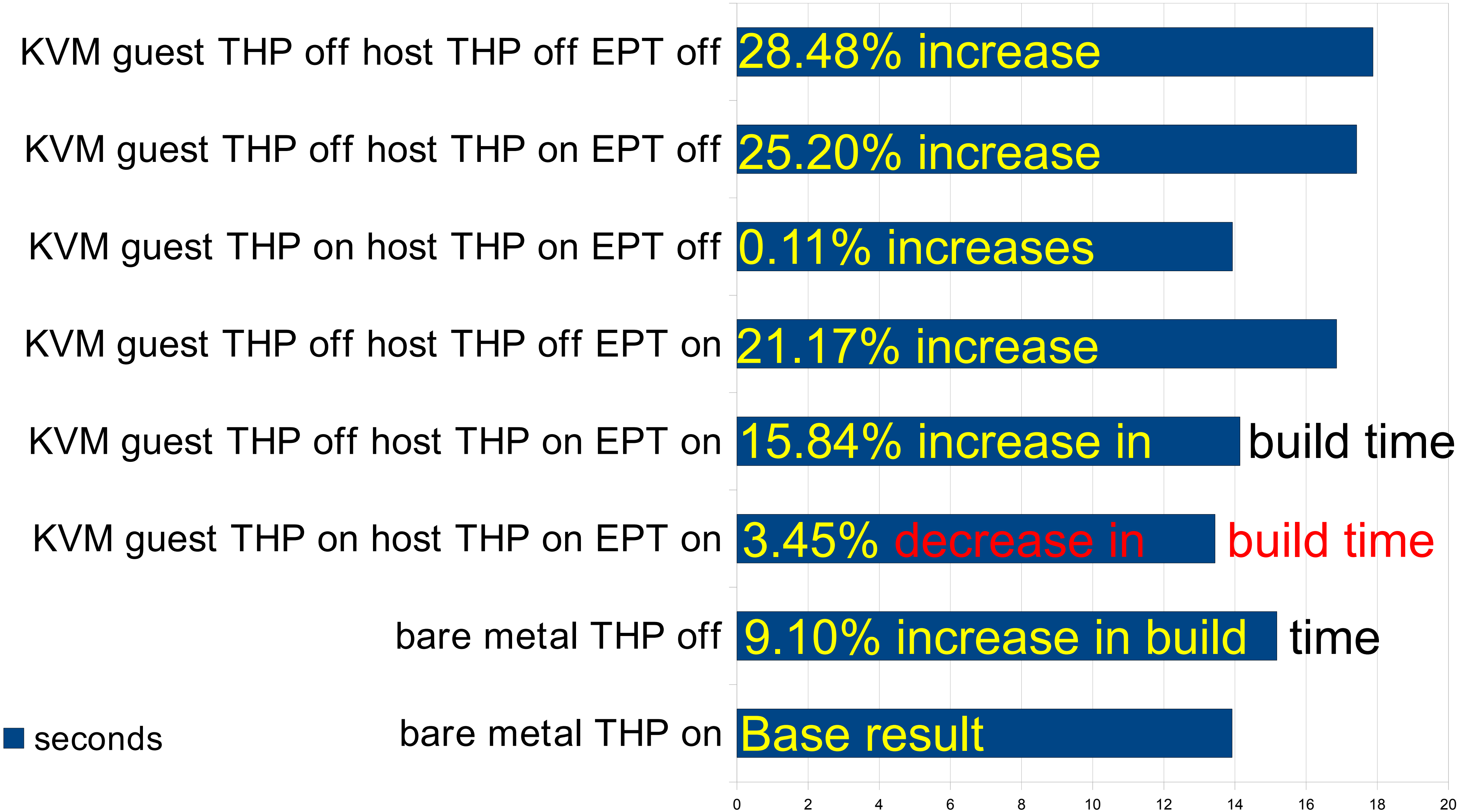
QEMU THP alignment

```
@@ -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);
         }
     }
```

kbuild bench
build time: lower is better



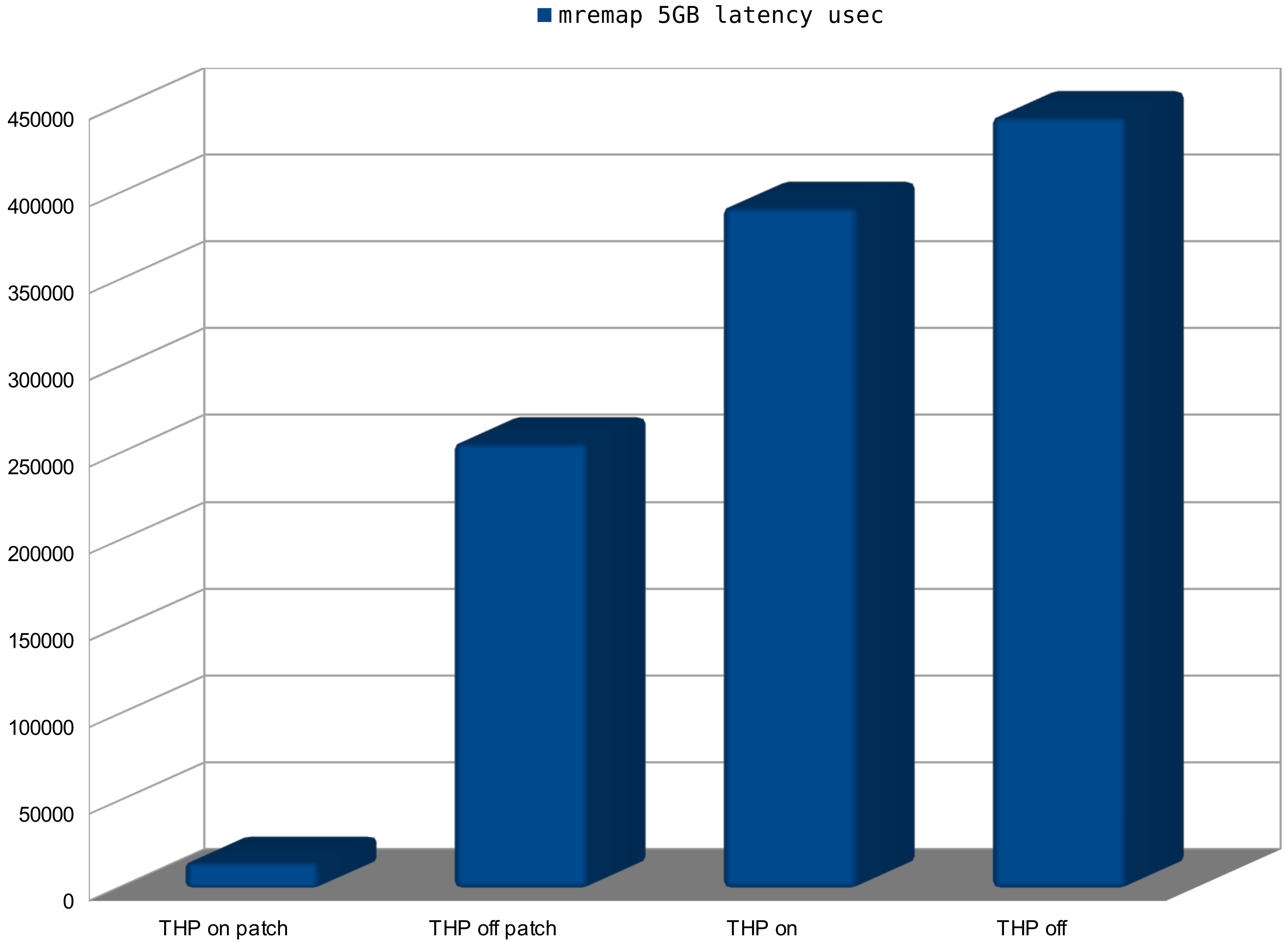
kbuild “EPT off” build time: lower is better



Phoronix test suite

- http://www.phoronix.com/scan.php?page=article&item=linux_transparent_hugepages&num=2
- IS.C test of NASA's OpenMP-based performance boost more than 20%
  - No virt
  - On thinkpad T16 notebook
    - Core 2 Duo T9300
    - 4GB of RAM
  - A bigger boost is expected on server/virt

mremap(5GB) latency usec



Multicore

- The only one not really giving any new problem compared to traditional SMP

Hyperthreading

- Fairness problems
  - Some CPUs may run faster than others
- Performance issues
  - If you have 4 HT and you use 2 cpus that are in the same physical core
- The scheduler has SIBLINGS class and is aware
  - CPU bindings may prevent the scheduler to do its job
  - Especially troubling with virtual machines if the hyperthreading CPU topology isn't visible by the guest OS

Hard NUMA bindings

- /dev/cpuset
- taskset wrapper
- sched_setaffinity/pthread_setaffinity_np
- set_mempolicy/sys_mbind
  - MPOL_DEFAULT
  - MPOL_BIND
  - MPOL_PREFERRED
  - MPOL_INTERLEAVE
    - F_STATIC/RELATIVE_NODES
- move_pages

redhat

NUMA topology

- Available in /sys/devices/system/node

```
./possible
./online
./has_normal_memory
./has_cpu
./node0
./node0/cpumap
./node0/cpulist
./node0/meminfo
./node0/numastat
./node0/distance
./node0/vmstat
./node0/scan_unevictable_pages
./node0/compact
./node0/cpu0
./node0/cpu1
./node0/cpu2
./node0/cpu3
./node0/cpu4
./node0/cpu5
./node0/cpu12
./node0/cpu13
./node0/cpu14
./node0/cpu15
./node0/cpu16
./node0/cpu17
./node0/hugepages
./node0/hugepages/hugepages-2048kB
./node0/hugepages/hugepages-2048kB/nr_hugepages
./node0/hugepages/hugepages-2048kB/free_hugepages
./node0/hugepages/hugepages-2048kB/surplus_hugepages
```

Scheduler domains

- Available in /sys/devices/system/node

./possible
./online
./has_normal_memory
./has_cpu
./node0
./node0/cpumap
./node0/cpulist
./node0/meminfo
./node0/numastat
./node0/distance
./node0/vmstat
./node0/scan_unevictable_pages
./node0/compact
./node0/cpu0
./node0/cpu1
./node0/cpu2
./node0/cpu3
./node0/cpu4
./node0/cpu5
./node0/cpu12
./node0/cpu13
./node0/cpu14
./node0/cpu15
./node0/cpu16
./node0/cpu17
./node0/hugepages
./node0/hugepages/hugepages-2048kB
./node0/hugepages/hugepages-2048kB/nr_hugepages
./node0/hugepages/hugepages-2048kB/free_hugepages
./node0/hugepages/hugepages-2048kB/surplus_hugepages

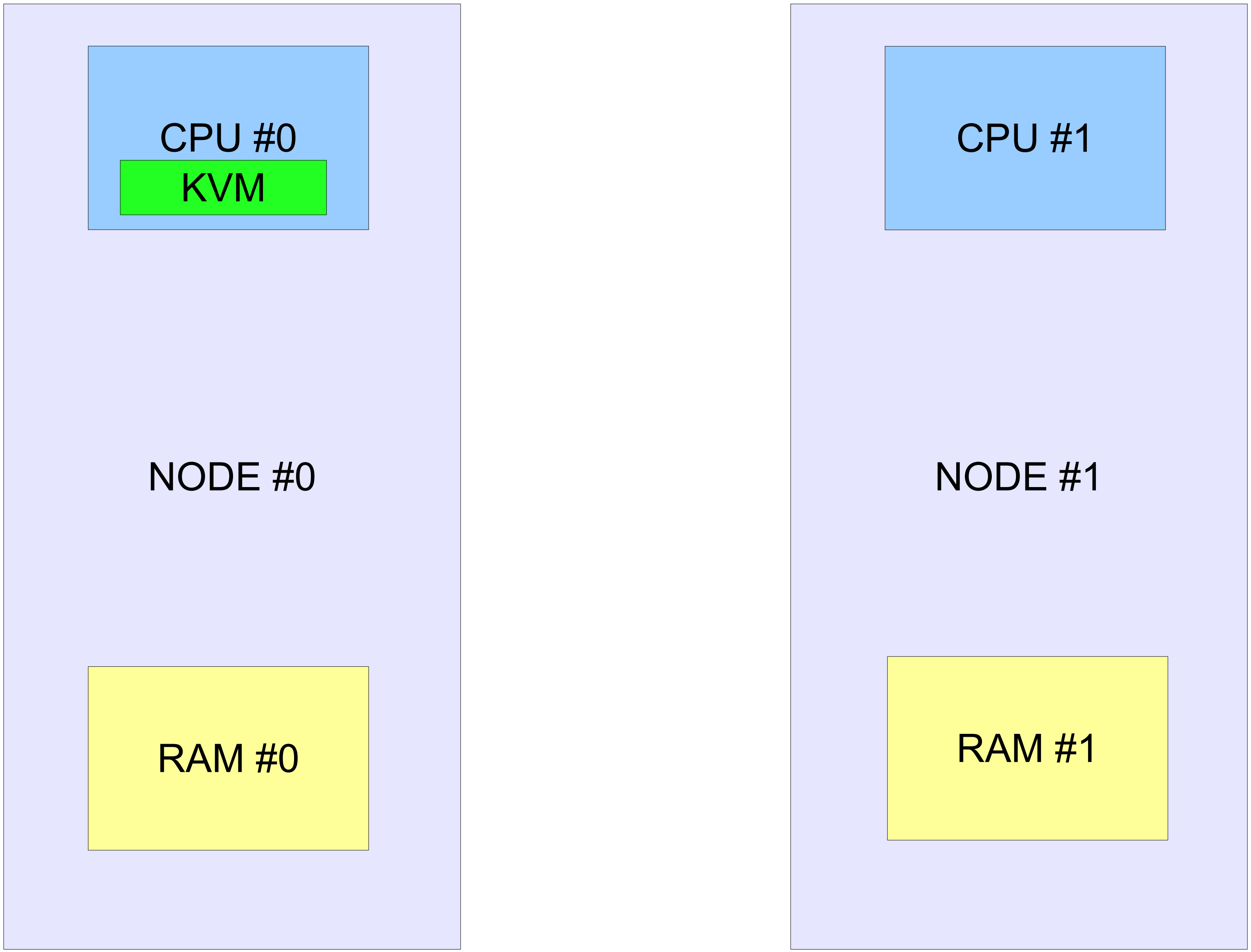
redhat

KVM NUMA awareness

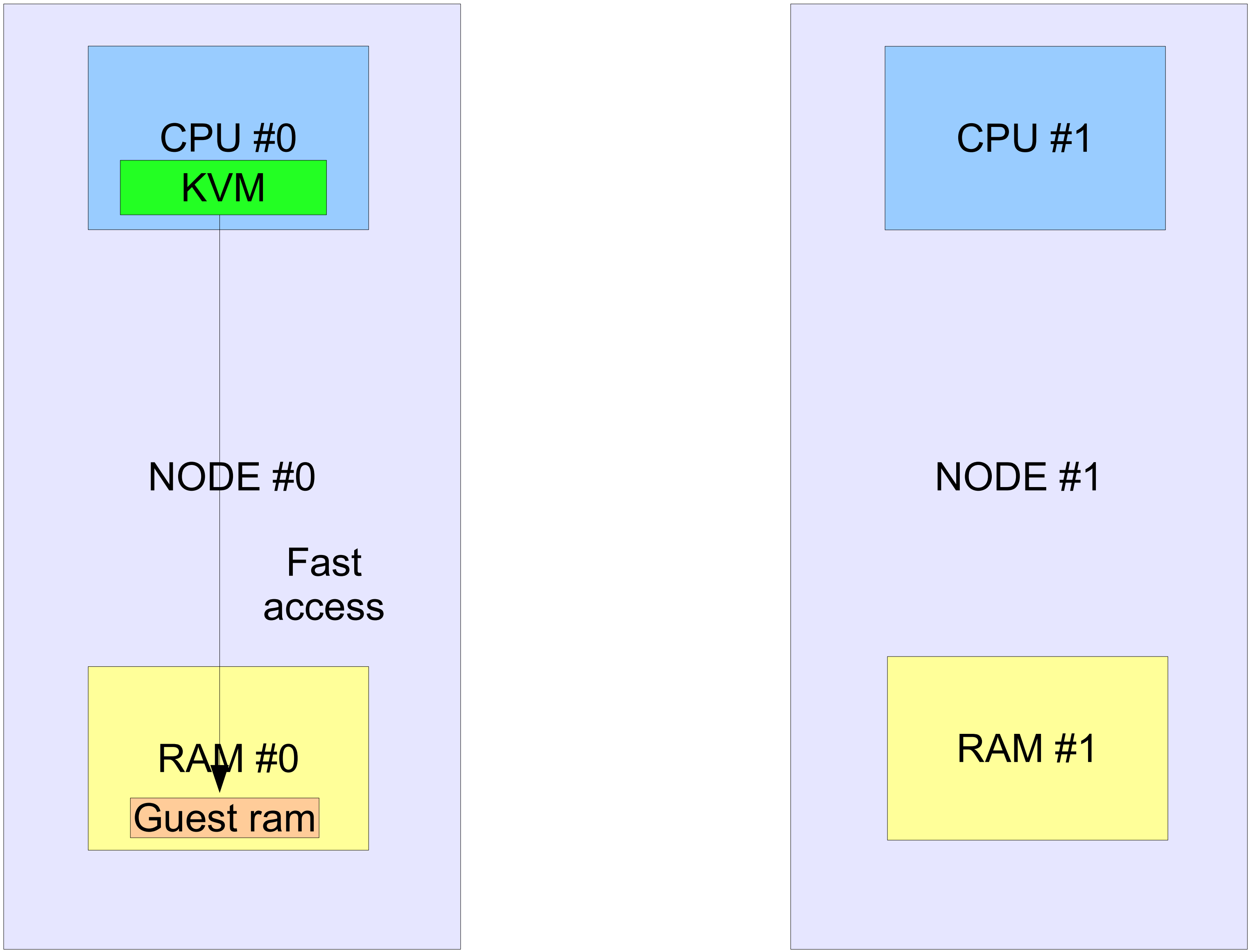
- I.e. making Linux NUMA aware
- The Linux Scheduler currently is blind about the memory placement of the process
- MPOL_DEFAULT allocates memory from the local node of the current CPU
- It all works well if the process isn't migrated by the scheduler to a different NUMA node later
  - Or if the memory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

redhat

KVM startup on CPU #0

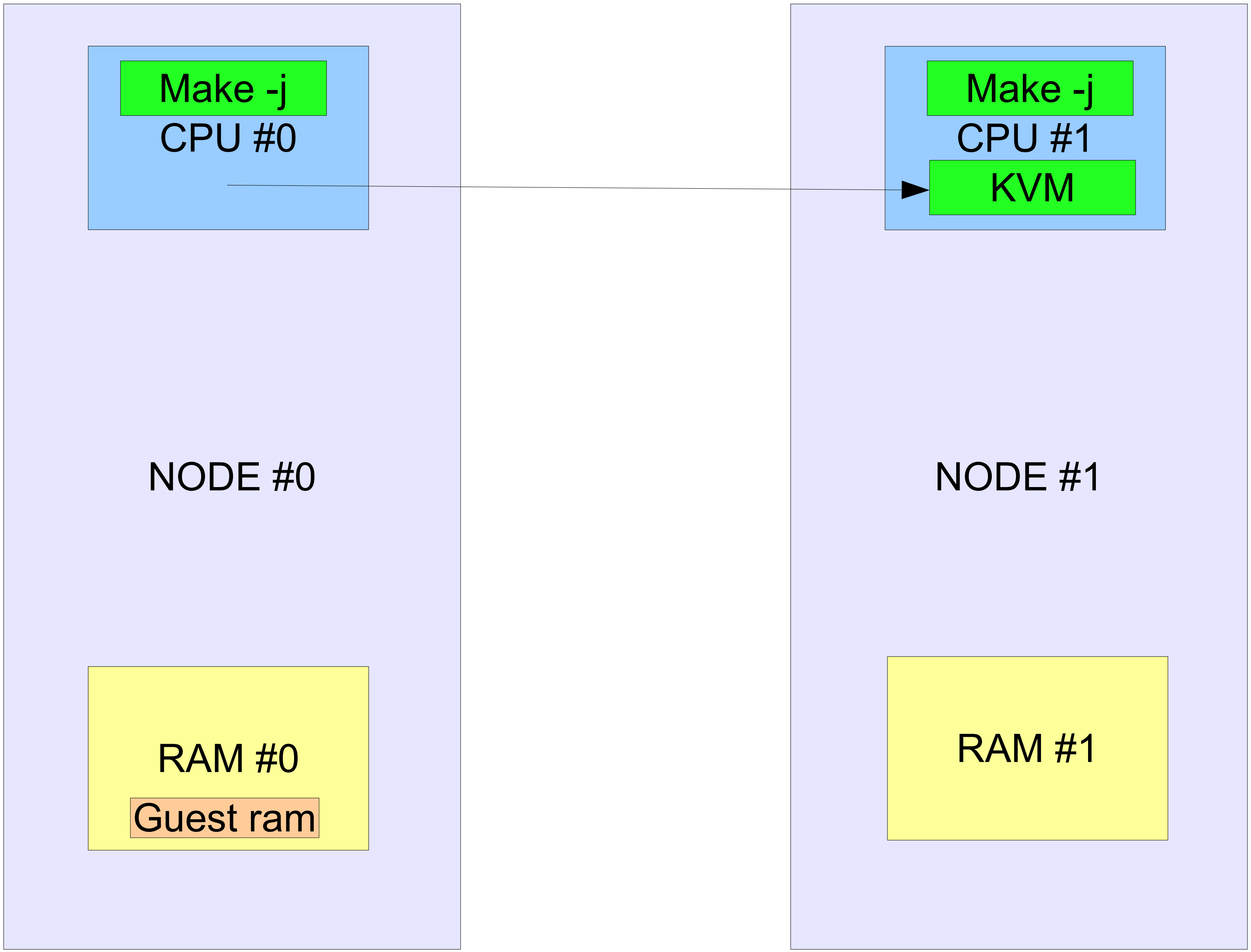


KVM allocates from RAM #0



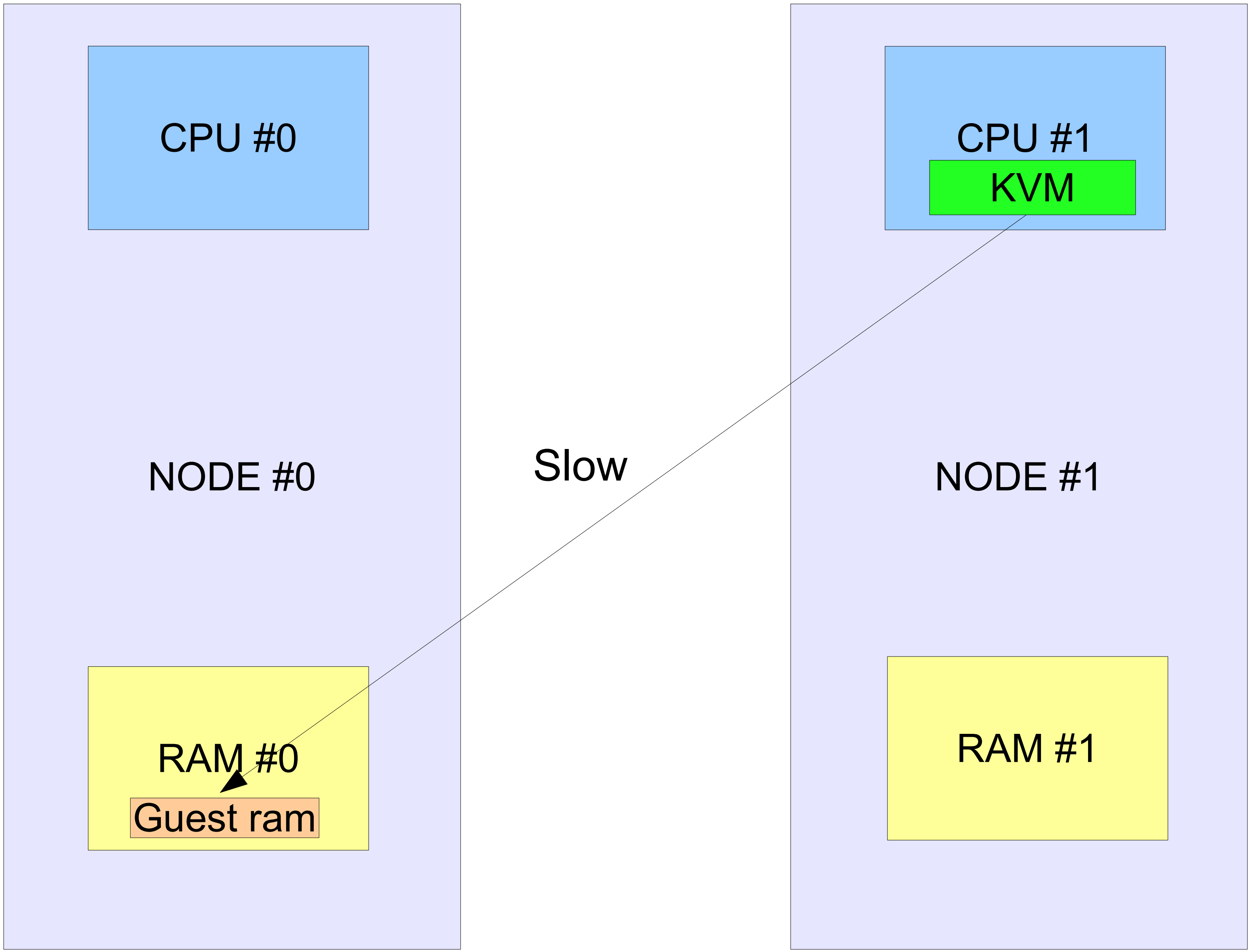
No NUMA hard bindings and MPOL_DEFAULT policy

Scheduler CPU migration



redhat

“make -j” load goes away



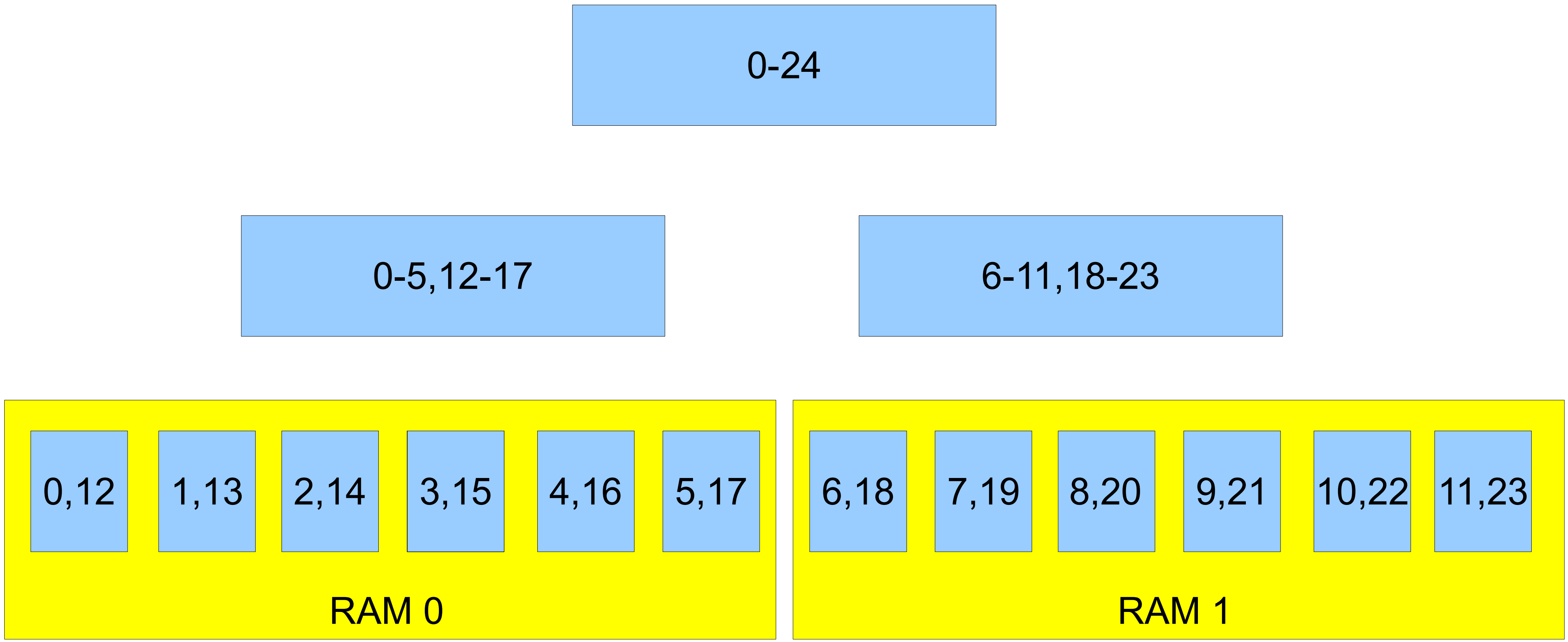
The Linux Scheduler is blind at this point: **KVM may stay in CPU #1 forever**

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

Scheduler domains



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

/proc/schedstat

```
version 15
timestamp 4294923310
cpu0 0 0 30689 5581 6746 3453 4433191001 409355508 7428
domain0 001001 1469 1469 0 0 0 0 0 1469 16 16 0 0 0 0 0 16 2623 2618 3 2778 2 0 0 2618 0 0 0 0 0 0 0 0 0 1229 26 0
domain1 03f03f 1452 1448 4 1450 0 0 0 1448 3 3 0 0 0 0 0 0 2621 2568 53 35054 0 0 4 2564 0 0 0 0 0 0 0 0 0 757 115 0
domain2 ffffff 293 293 0 0 0 0 1 292 1 1 0 0 0 0 0 0 2621 2503 117 69133 1 0 8 2495 0 0 0 0 0 0 0 0 0 1183 13 0
cpu1 0 0 6901 3432 2776 446 223141188 3127007 3468
domain0 002002 1002 998 4 4708 0 0 0 998 3 3 0 0 0 0 0 3 1055 1028 27 18616 0 0 0 1028 0 0 0 0 0 0 0 0 0 174 1 0
domain1 03f03f 993 983 9 11884 1 0 0 983 3 3 0 0 0 0 0 0 1055 1017 37 24802 1 0 1 1016 0 0 0 0 0 0 0 0 0 640 4 0
domain2 ffffff 217 217 0 0 0 0 0 41 0 0 0 0 0 0 0 0 1054 908 146 77215 0 0 5 903 0 0 0 0 0 0 0 0 0 1515 2 0
cpu2 0 0 2998 1498 1549 96 71761221 1380590 1500
domain0 004004 304 304 0 81 0 0 0 304 0 0 0 0 0 0 0 0 301 300 1 593 0 0 0 300 0 0 0 0 0 0 0 0 0 11 0 0
domain1 03f03f 256 243 12 11254 1 1 0 243 0 0 0 0 0 0 0 0 301 269 31 16642 1 0 1 268 0 0 0 0 0 0 0 0 0 301 0 0
domain2 ffffff 102 102 0 0 0 0 0 2 0 0 0 0 0 0 0 0 300 242 57 16244 1 0 0 242 0 0 0 0 0 0 0 0 0 1140 0 0
cpu3 0 0 2882 1441 1395 73 58279507 928100 1441
domain0 008008 232 232 0 0 0 0 0 232 0 0 0 0 0 0 0 0 163 162 1 88 0 0 0 162 0 0 0 0 0 0 0 0 0 4 0 0
domain1 03f03f 211 204 7 6752 0 0 0 204 0 0 0 0 0 0 0 0 163 139 24 16387 0 0 0 139 0 0 0 0 0 0 0 0 0 413 0 0
domain2 ffffff 92 92 0 0 0 0 0 0 0 0 0 0 0 0 0 0 163 136 26 9417 3 0 1 135 0 0 0 0 0 0 0 0 0 904 0 0
cpu4 0 0 142 74 52 46 22458588 281180 68
domain0 010010 170 170 0 0 0 0 0 170 1 1 0 0 0 0 0 1 70 70 0 0 0 0 0 70 0 0 0 0 0 0 0 0 0 0 0 0
domain1 03f03f 147 140 7 6725 0 0 0 140 1 1 0 0 0 0 0 0 70 63 6 4219 1 0 0 63 0 0 0 0 0 0 0 0 0 3 0 0
domain2 ffffff 86 86 0 0 0 0 0 1 0 0 0 0 0 0 0 0 69 66 3 1884 0 0 0 66 0 0 0 0 0 0 0 0 0 2 0 0
cpu5 0 0 136 71 53 45 22263992 312805 65
domain0 020020 181 181 0 0 0 0 0 181 0 0 0 0 0 0 0 0 67 67 0 0 0 0 0 67 0 0 0 0 0 0 0 0 0 2 0 0
domain1 03f03f 161 153 8 6956 0 0 0 153 0 0 0 0 0 0 0 0 67 62 4 3518 1 0 0 62 0 0 0 0 0 0 0 0 0 3 0 0
domain2 ffffff 88 88 0 0 0 0 0 0 0 0 0 0 0 0 0 0 66 63 3 5400 0 0 0 63 0 0 0 0 0 0 0 0 0 2 0 0
cpu6 0 0 9520 4338 4539 1848 515457042 24326084 5180
domain0 040040 1123 1123 0 0 0 0 0 1123 3 3 0 0 0 0 0 3 1469 1468 1 65 0 0 0 1468 0 0 0 0 0 0 0 0 0 232 11 0
domain1 fc0fc0 914 908 1 10780 8 0 0 908 0 0 0 0 0 0 0 0 1469 1445 22 21487 2 0 1 1444 0 0 0 0 0 0 0 0 0 1441 88 0
domain2 ffffff 222 215 7 5809 0 0 0 215 1 1 0 0 0 0 0 1 1467 1374 92 109754 1 0 2 1372 0 0 0 0 0 0 0 [illegible] 0 1017 5 0
[..]
```

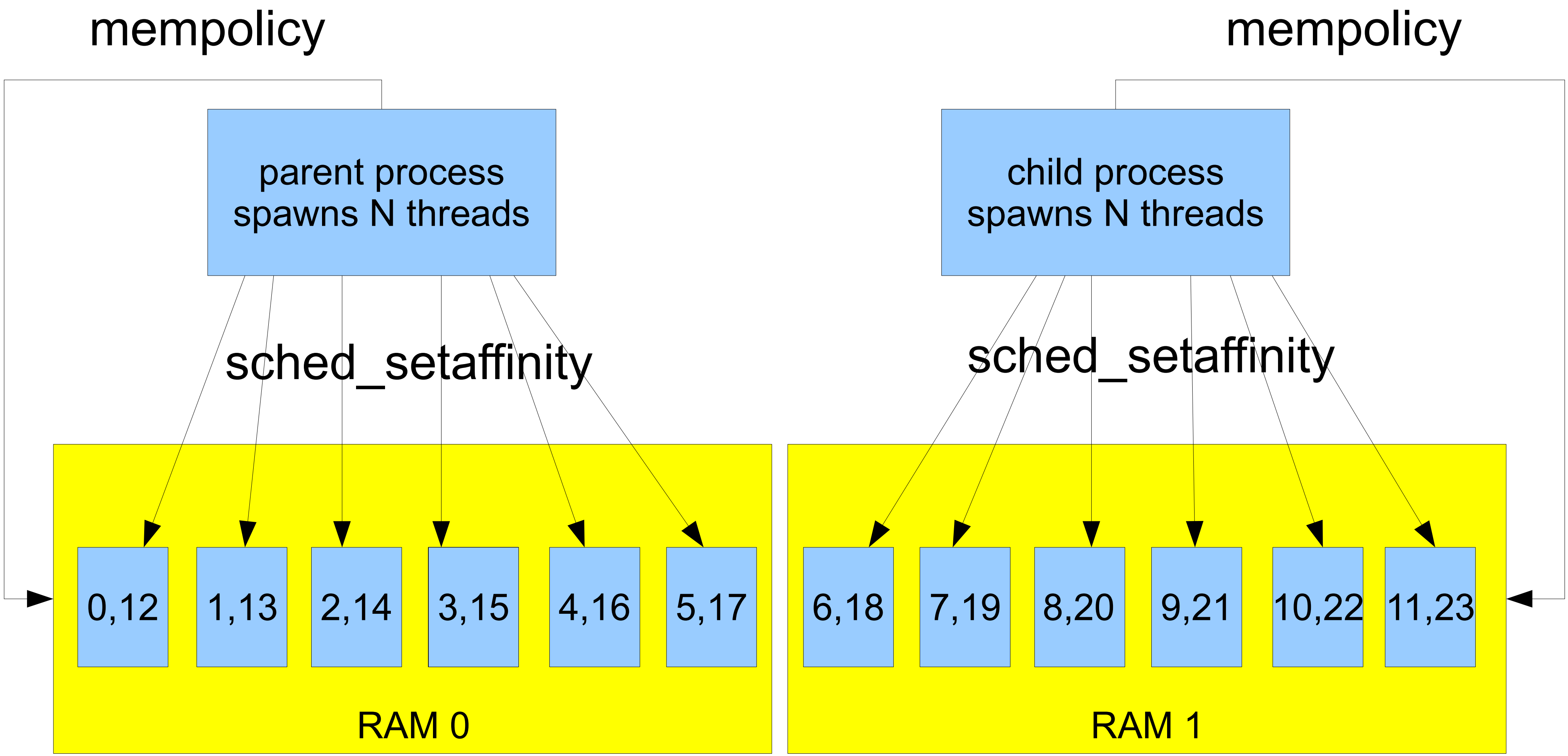
Hard bindings and hypervisors

- 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)

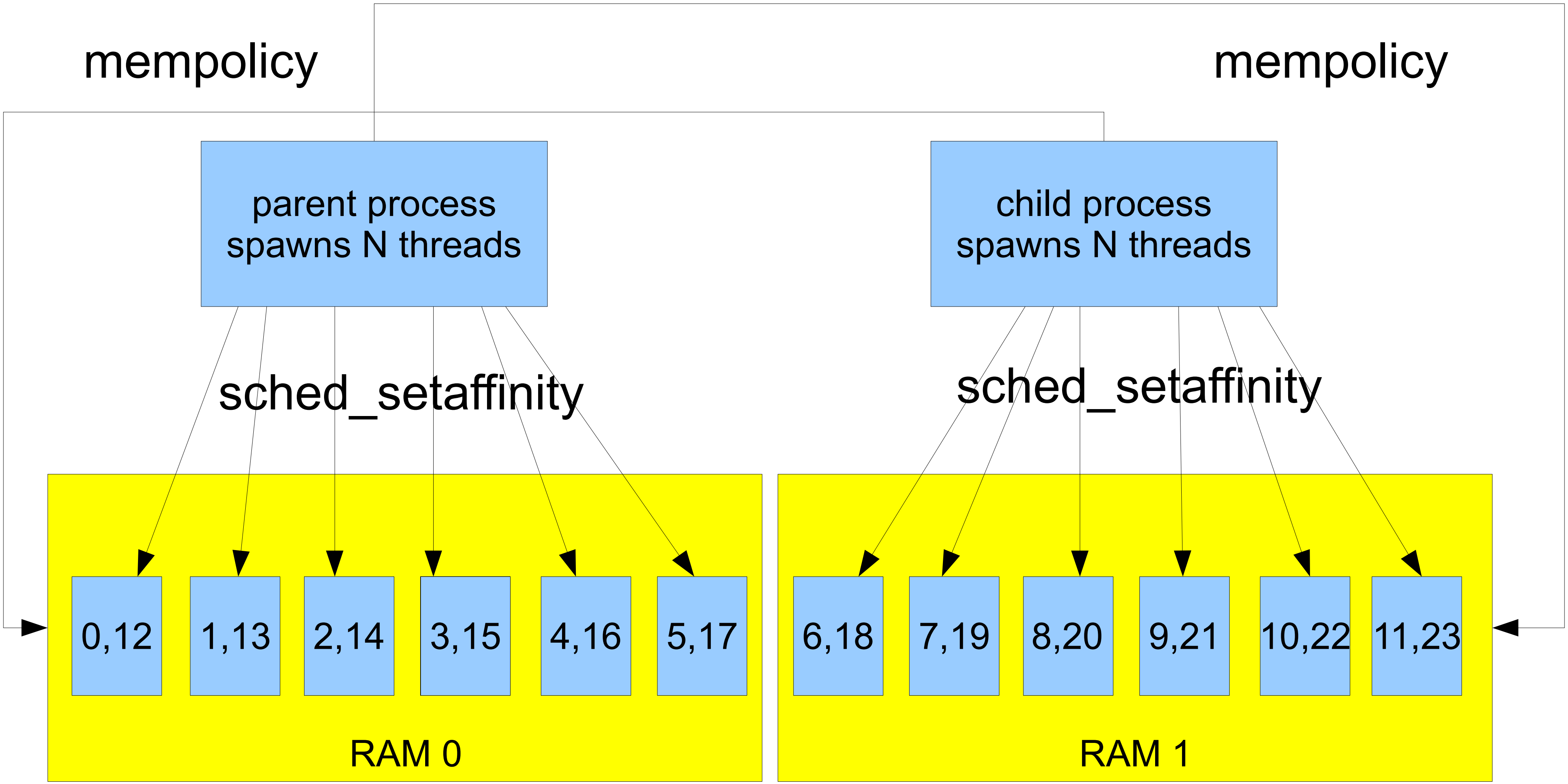
```
#define SIZE (6UL*1024*1024*1024)
#define THREADS 24

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

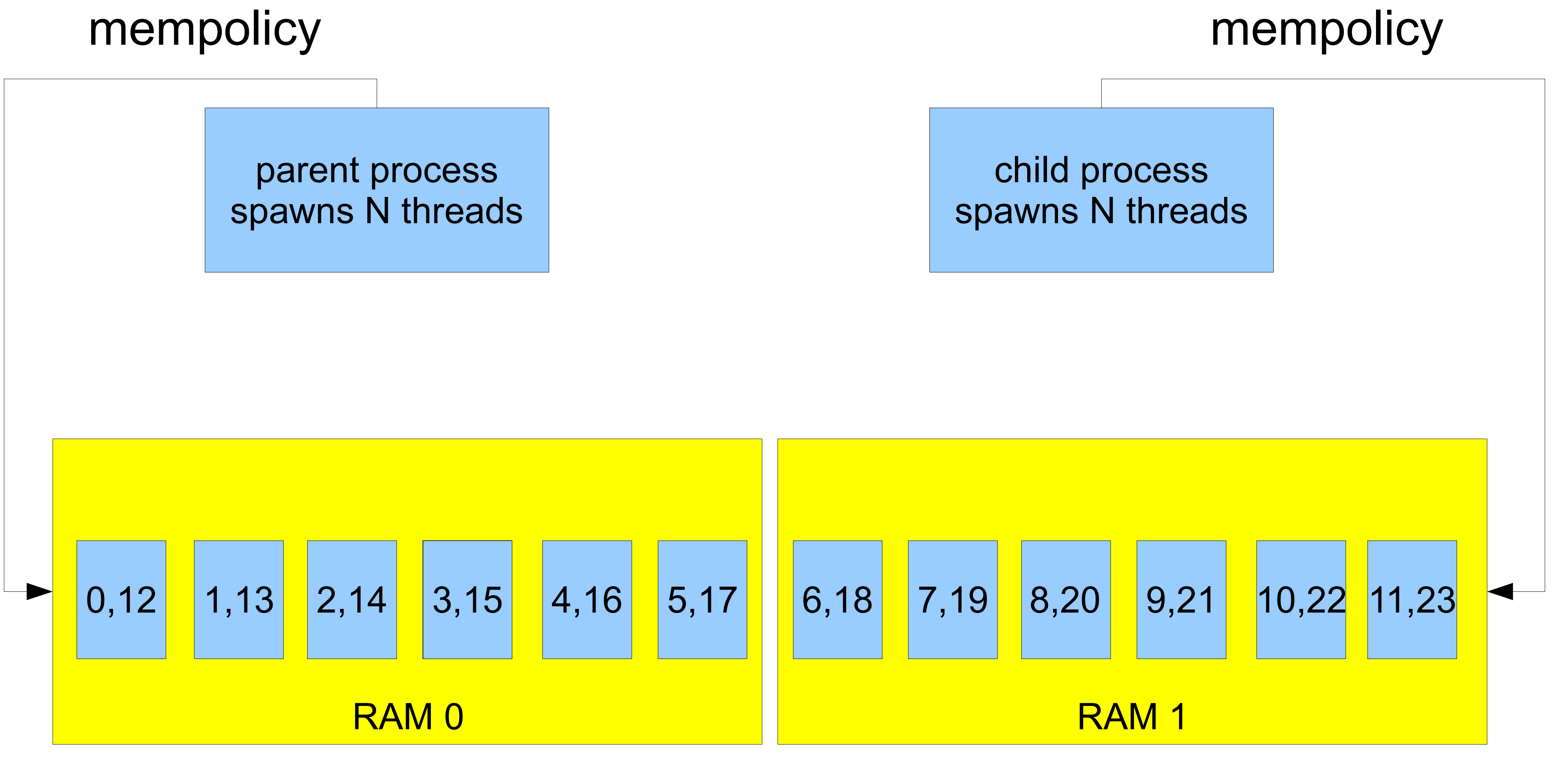
mempolicy + setaffinity remote



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

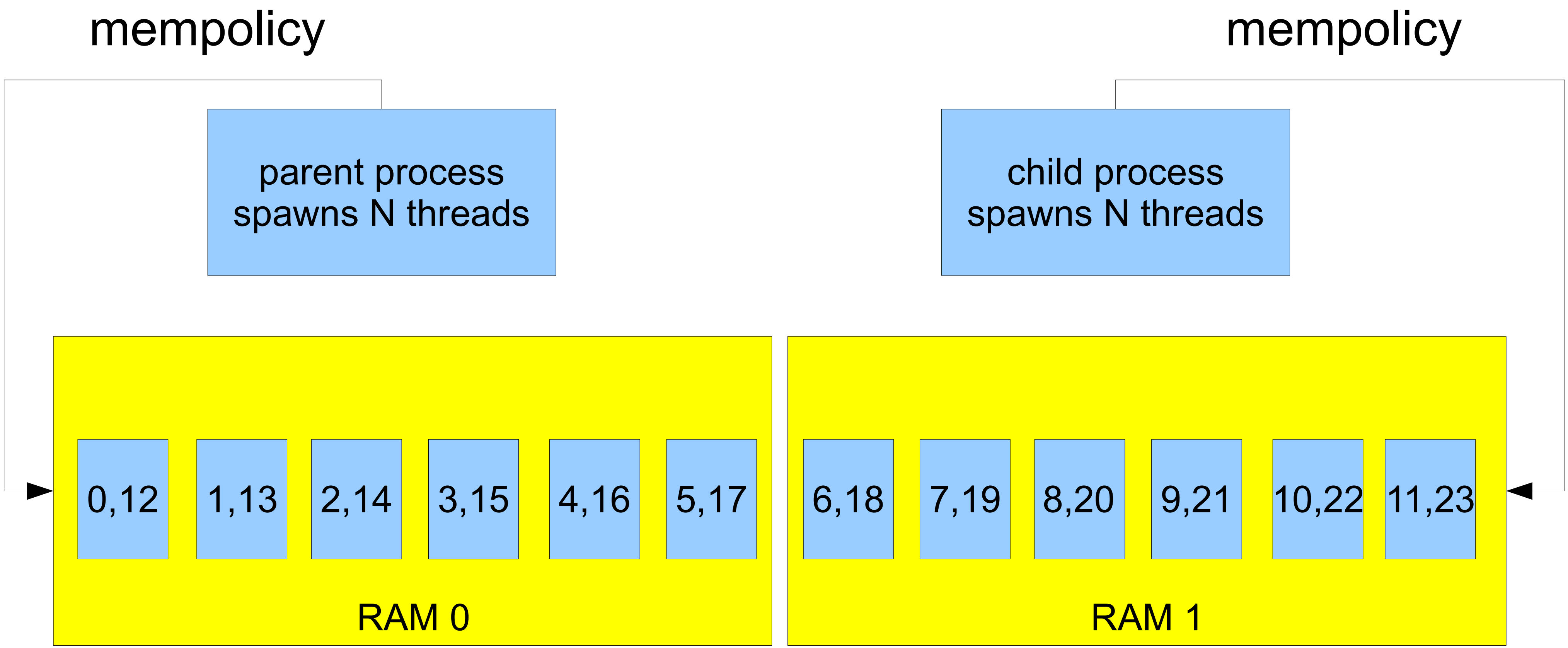
redhat

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

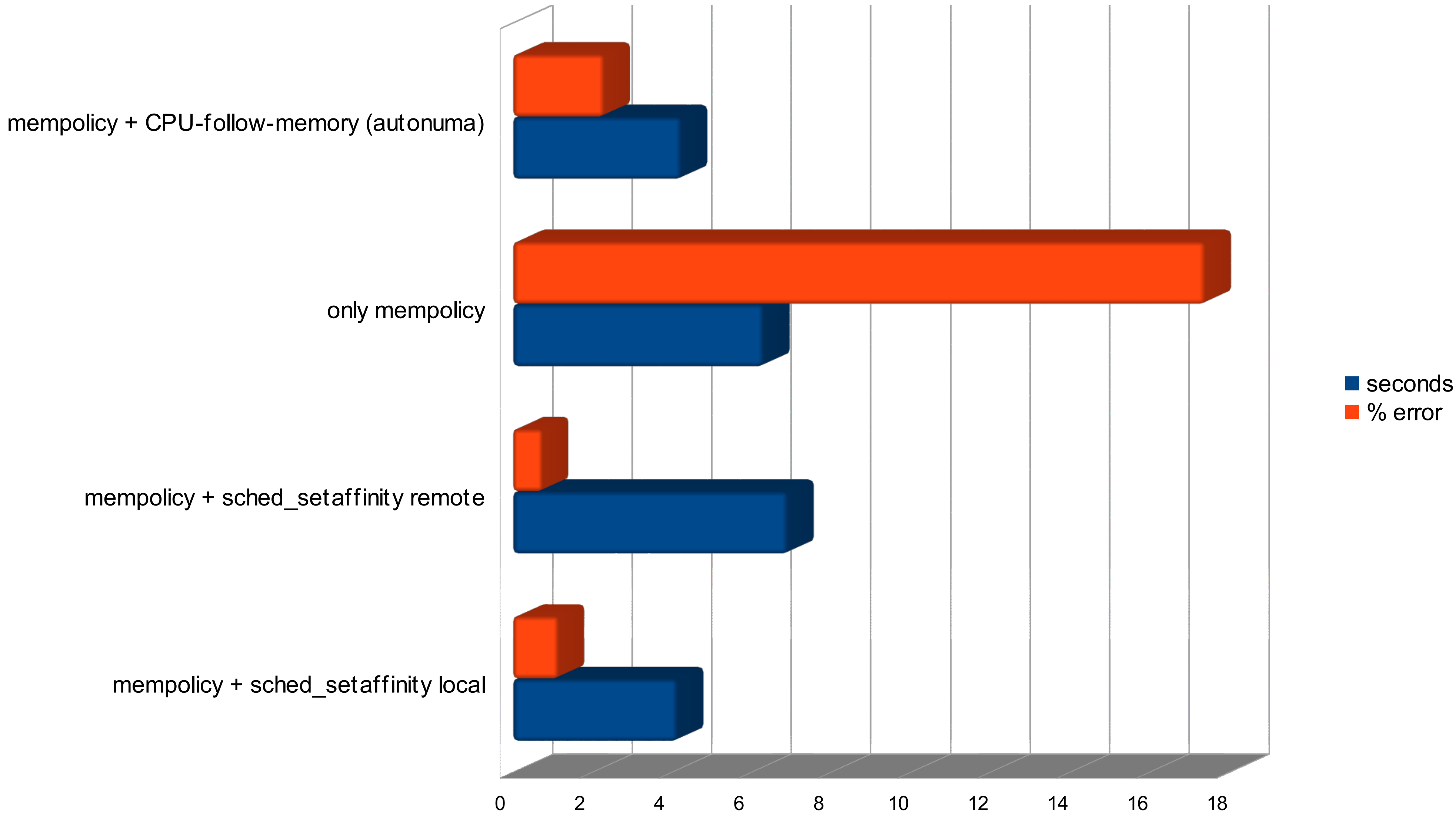


The host CPU scheduler understand the parent process has most of the RAM allocated in NODE 0 and the child in NODE 1

No scheduler hints from userland

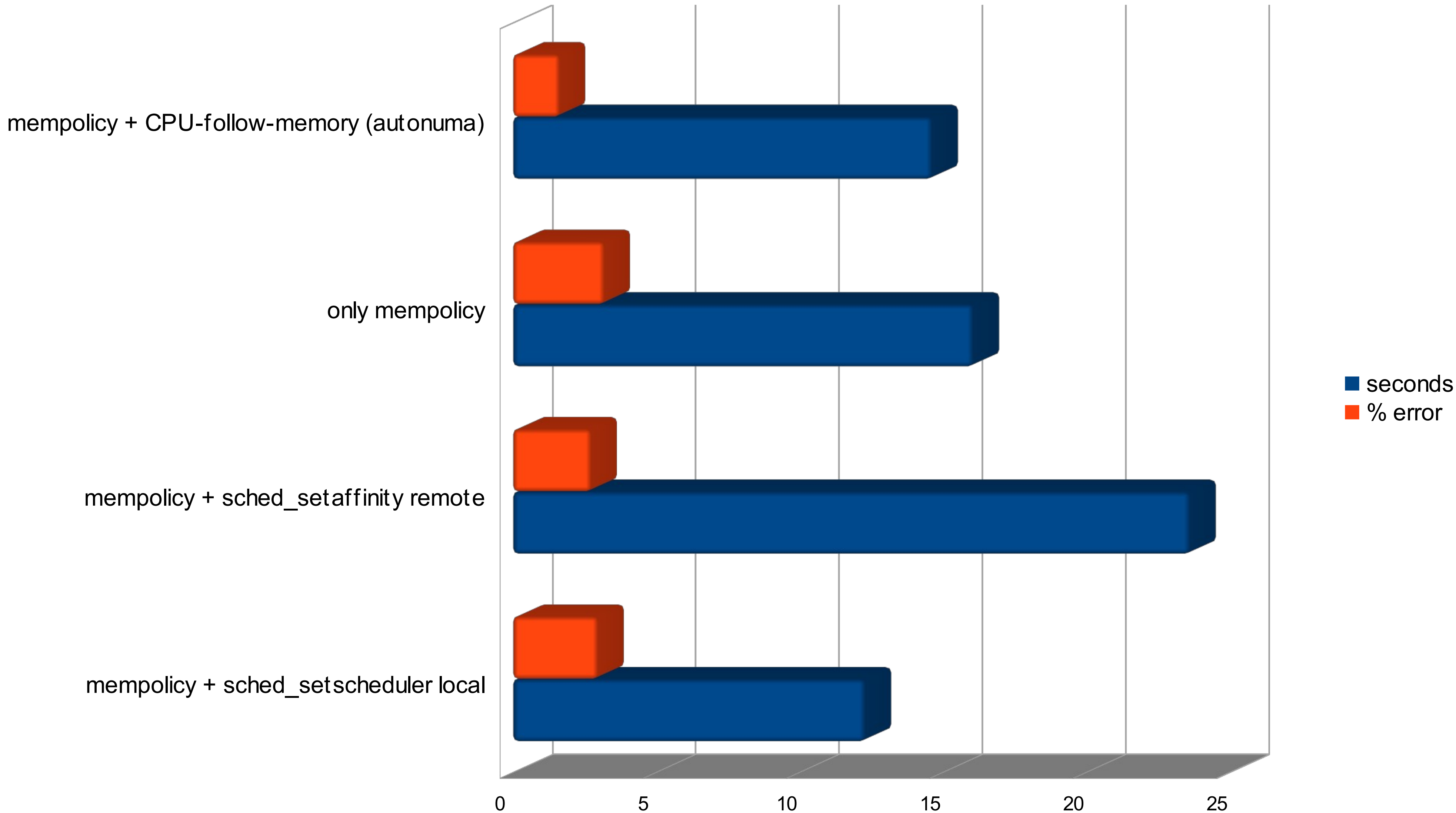
Mempolicy() doesn't have any scheduler effect

1 thread x 2 processes



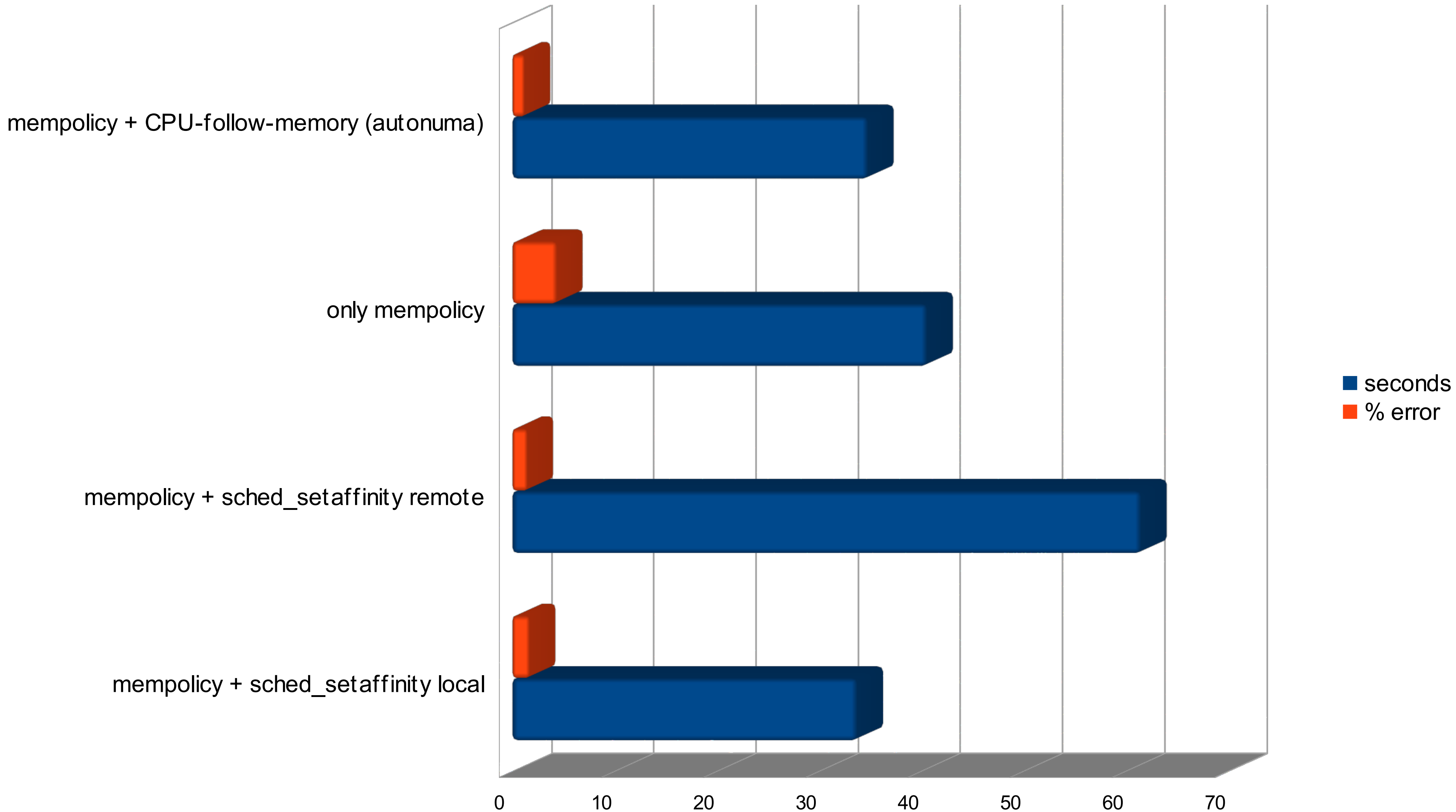
Only 2 CPUs used, 2 nodes 2 sockets 12 cores 24 threads

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)

redhat

memory-follow-CPU

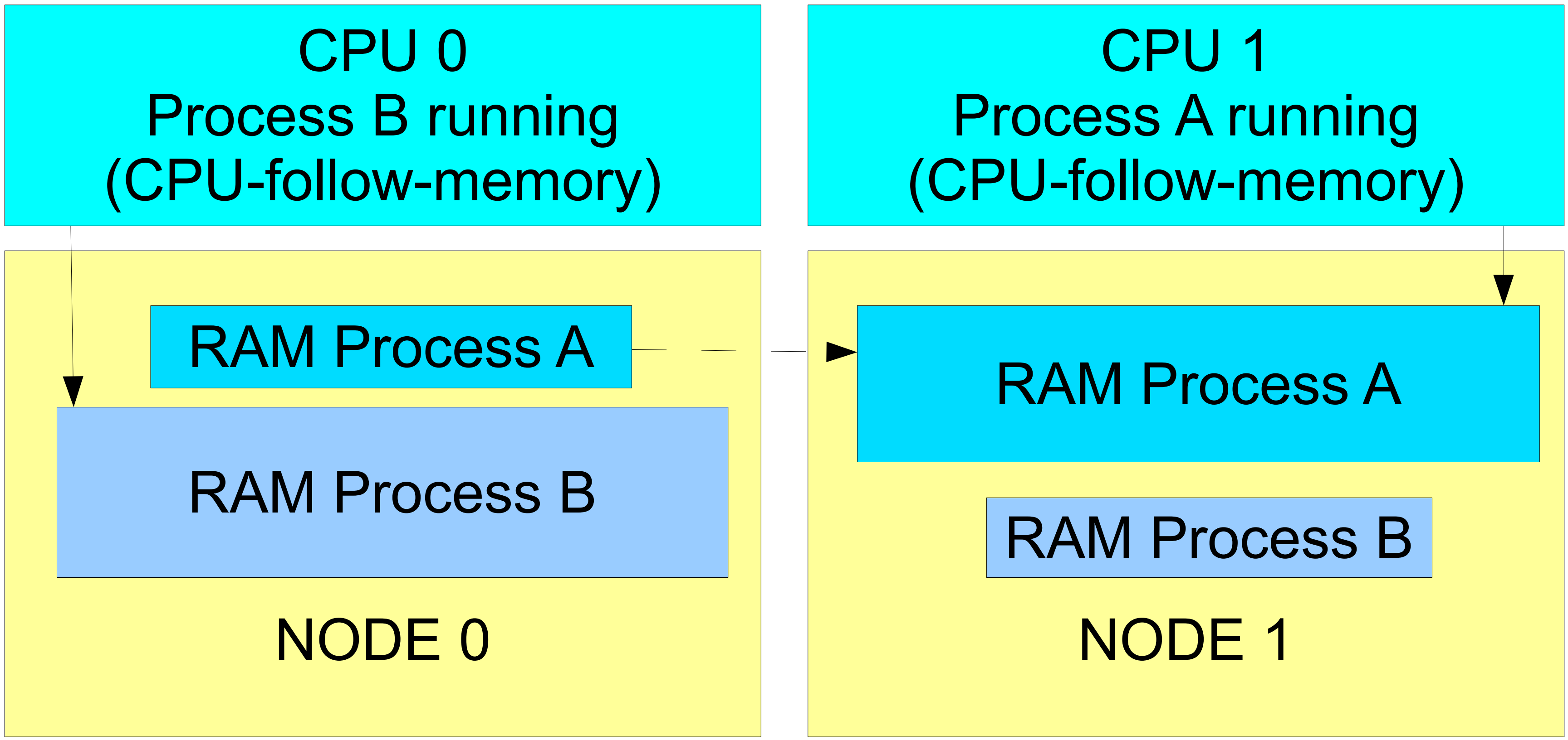
- Converge the RAM of the process into the node where it's running on by migrating it in the background
- If CPU-follow-memory doesn't follow memory because of too high load in the preferred nodes
  - Migrate the memory of the process to the node where the process is really running on and converge there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

redhat

Auto NUMA memory migration

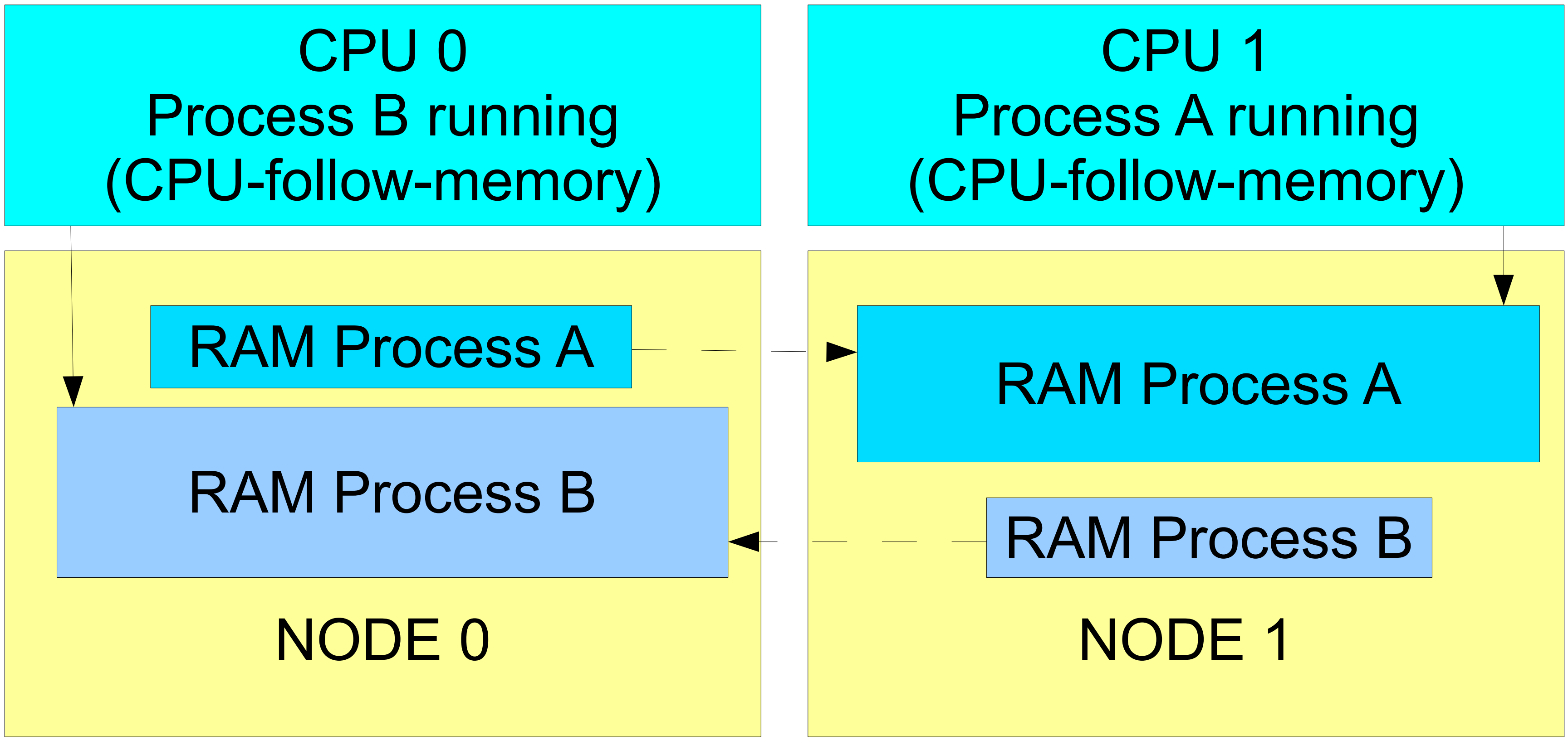
- We need to find a process that has RAM in NODE 1 and wants to converge into NODE 0, in order to migrate the RAM of another process from NODE 0 to NODE 1
  - This will keep the memory pressure balanced
  - Pagecache/swapcache/buffercache may be migrated as fallback but active process memory should be 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



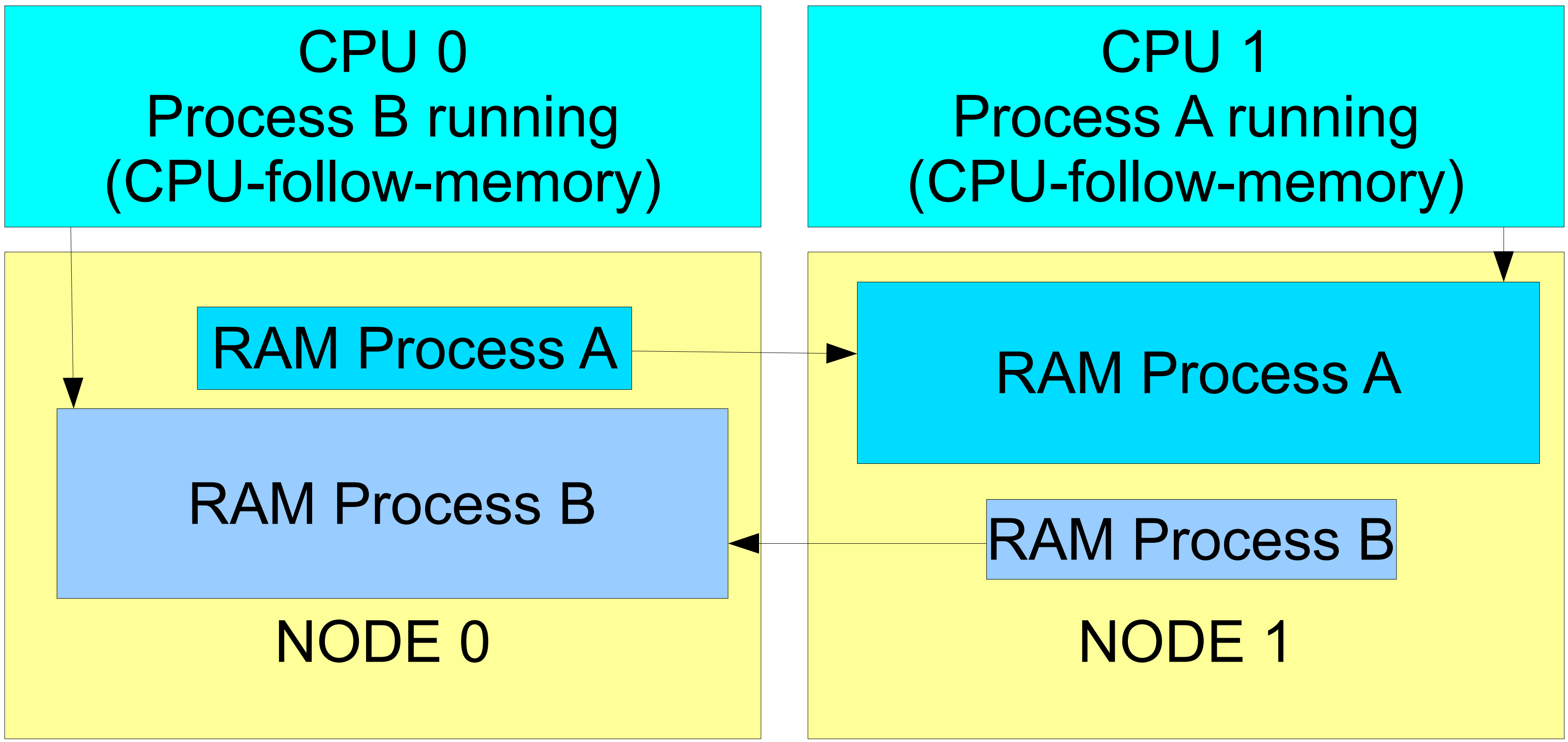
memory-follow-CPU wants to migrate the RAM of Process A from NODE0 to NODE 1

Auto NUMA memory migration



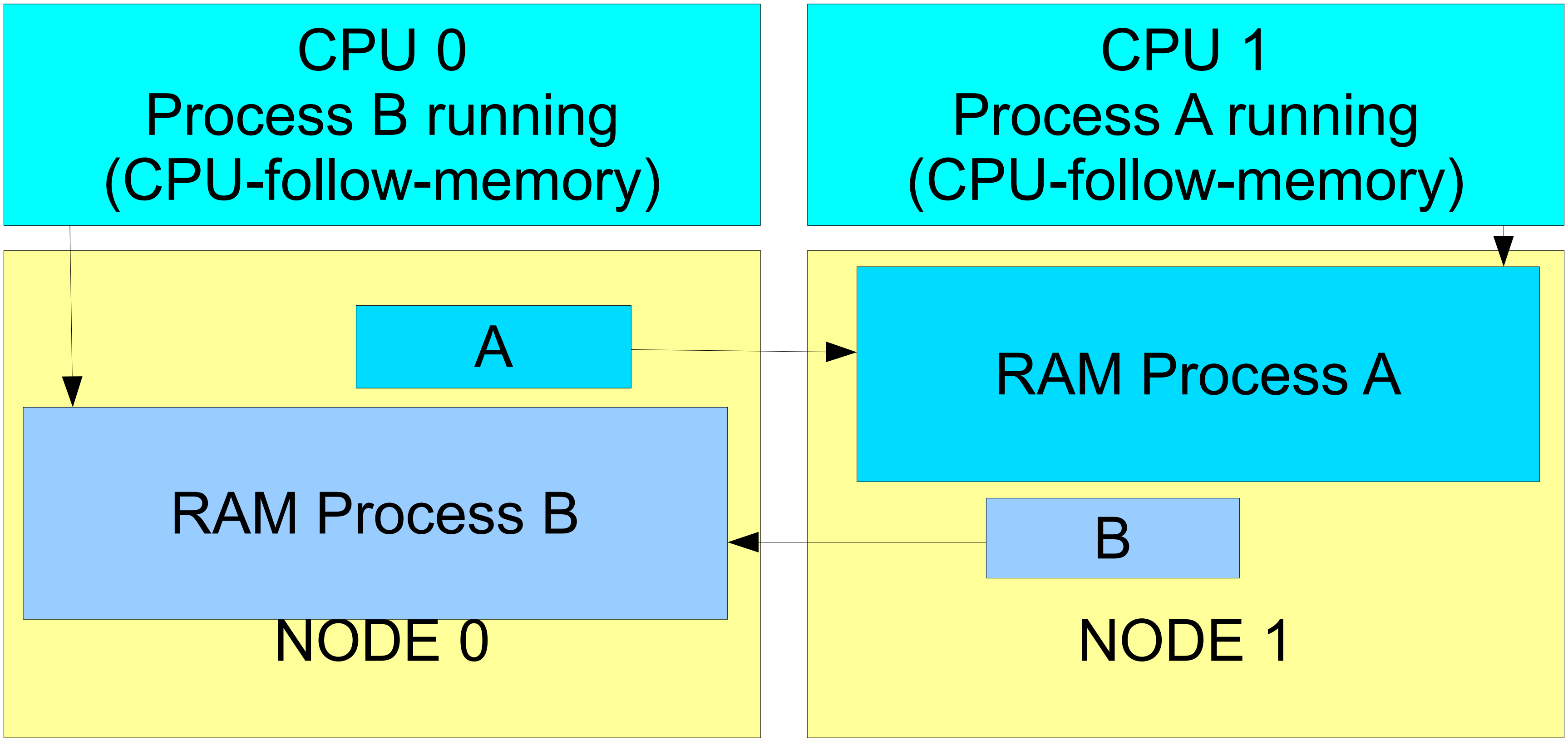
memory-follow-CPU need to find another process
with memory on NODE 1 that wants to migrate to NODE 0
Process B is ideal

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

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

redhat

Locking

- Kernel
  - RCU/SRCU
  - Seqlock
  - Spinning Mutex
  - Ticket spinlocks (FIFO)
  - rw spinlocks
  - rw semaphores
- Userland
  - pthread_mutex_lock/unlock/trylock
    - futex
  - RCU userland

`perf` profiling of translate.o

24-way SMP (12 cores, 2 sockets) 16G RAM host, 24-vcpu 15G RAM guest

THP always bare metal (base result)

```
40746051351  cycles                  ( +-   5.597% )
36394696366  instructions            #      0.893 IPC     ( +-   0.007% )
 9602461977  dTLB-loads                ( +-   0.006% )
   45123574  dTLB-load-misses            ( +-   0.614% )

13.920436128  seconds time elapsed   ( +-   5.600% )
```

THP never bare metal (9.10% slower)

```
44492051930  cycles                  ( +-   5.189% )
36757849113  instructions            #      0.826 IPC     ( +-   0.001% )
 9693482648  dTLB-loads                ( +-   0.004% )
   63675970  dTLB-load-misses            ( +-   0.598% )

15.188315986  seconds time elapsed   ( +-   5.194% )
```

git

- Crypto hash on whole repo contents
- Gpg sig on the hash through tags
- Data de-duplicating storage backend
- Very efficient and compact
- Powerful fronthand options (rebase -i, commit -i, cherry-pick, clone –reference, qgit4, git log --graph etc..)
- Kernel hacker user interface...

Q/A

- You're very welcome!