

Performance Analysis of a User-level Memory Server

Scott Pakin and Greg Johnson

Performance and Architecture Lab (PAL)
Los Alamos National Laboratory

18 September 2007

Observations

- Users need to analyze large volumes of data
 - *Example:* Gather access-pattern statistics (e.g., reuse distance) for an address trace of a large application
- Scripting languages are convenient for data analysis
 - Quick to write custom programs for processing results
- Big data → big memory demands
 - “Read all data, process in memory, write results”
 - Working set may inherently be very large
 - ◆ Large data structures with little locality (e.g., nested hash tables)
 - Potentially only small computation between data accesses
 - Want to avoid the disk if at all possible

Alternatives for Large-memory Scripts

- Buy more memory
 - Expensive, especially if most apps. don't need it
 - May not be able to fit in an existing workstation
 - Parallelize the script
 - Scripts not often worth parallelizing
 - May be run only a few times
 - Significant effort to convert, e.g., Perl to C + MPI
 - Not much available parallelism in many scripts
- ➔ Run a memory server

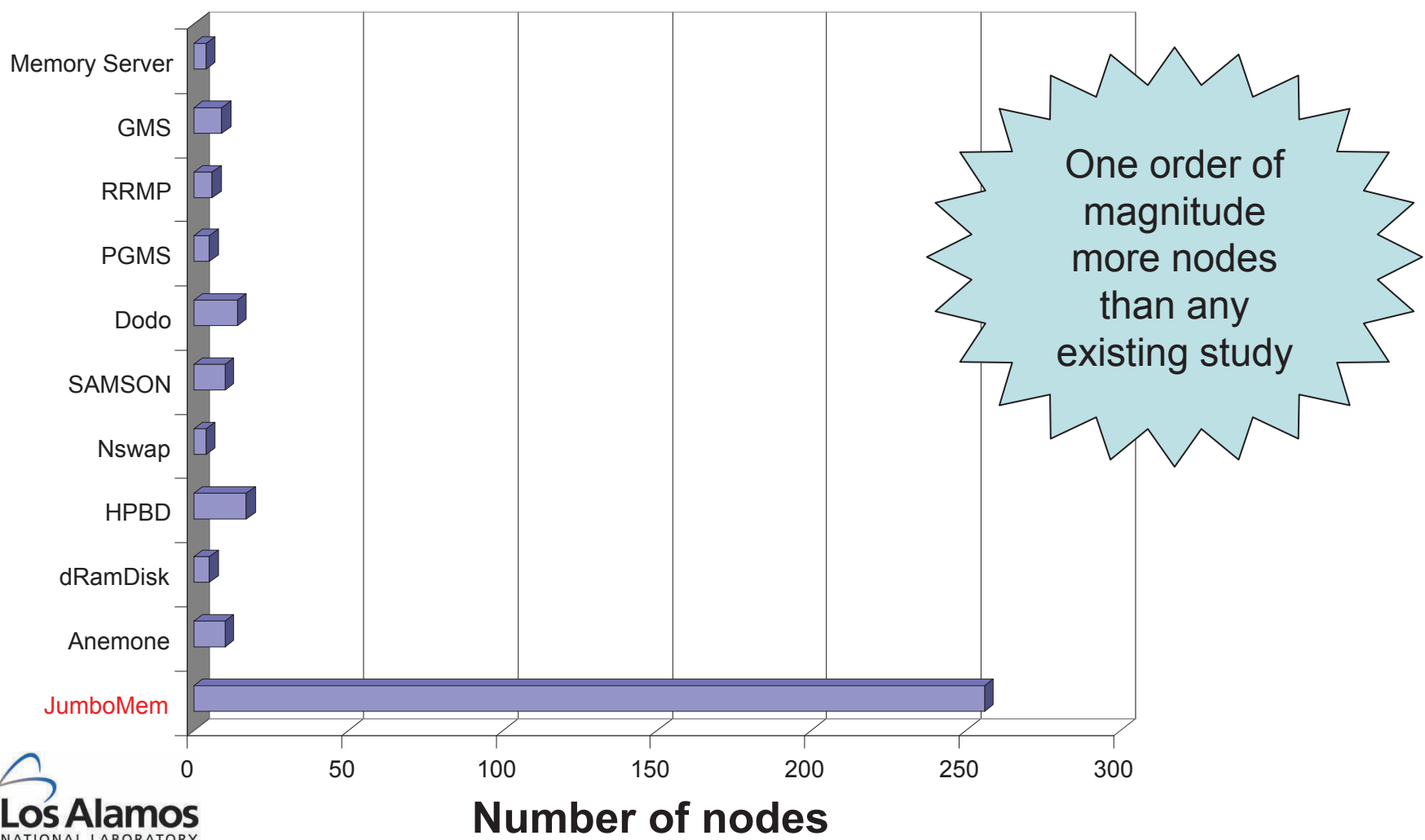
Memory Servers

- Make global cluster memory available to a sequential program
 - Paging from remote memory
 - Additional level of memory hierarchy
- Old concept
 - First proposed (arguably) in Garcia-Molina et al., *A Massive Memory Machine*, IEEE TC, May 1984
 - “[F]or argument’s sake let us say we want on the order of *tens of billions of bytes* of main physical memory...”
- Still applicable today
 - Programs and data grow to fit available resources

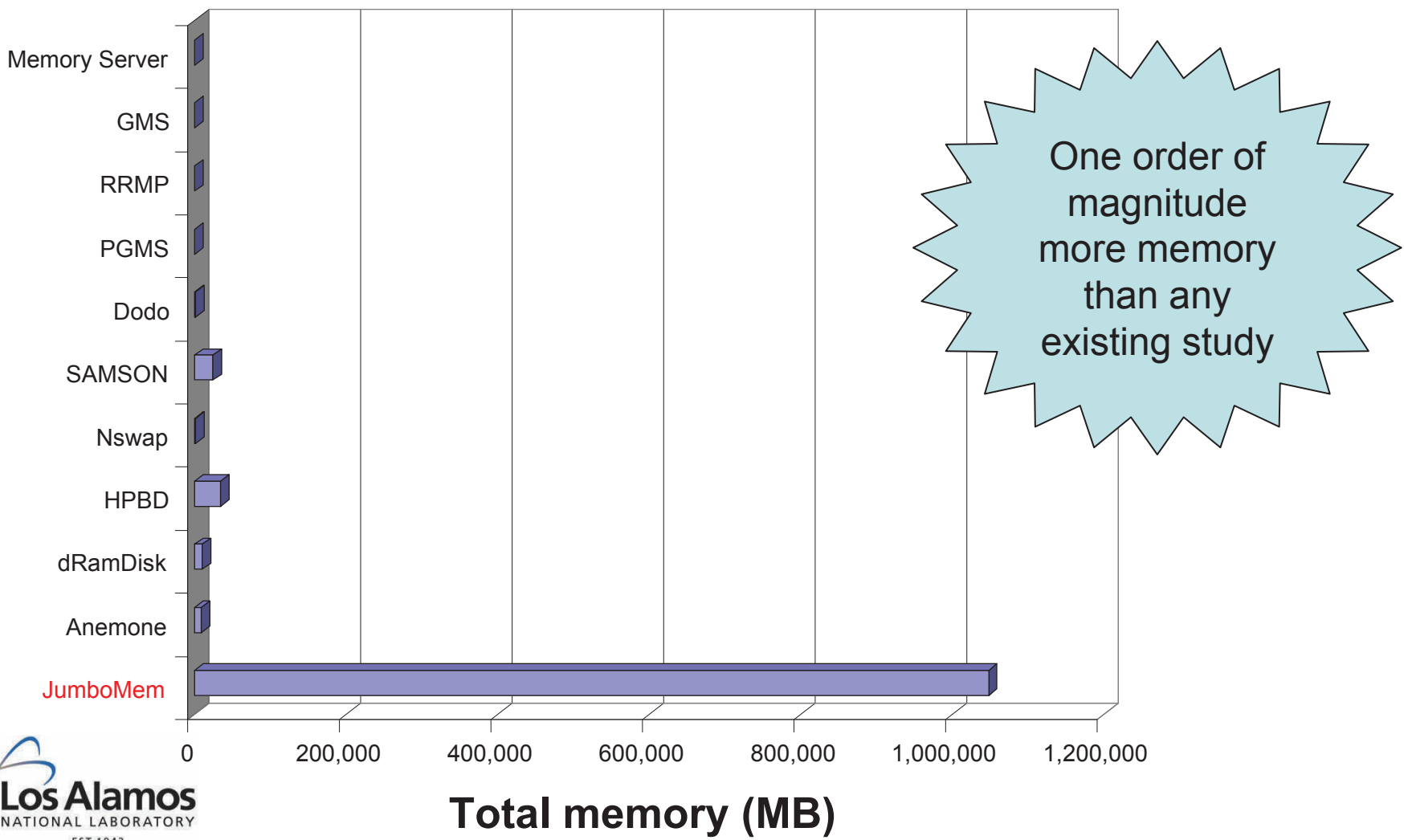
Our Contributions

- Can one construct a *transparent, entirely user-level* memory server?
 - Access to large, high-performance clusters is commonplace
 - Administrative access to large, high-performance clusters is not
 - What are the limitations?
 - What's the performance?
- How scalable are memory servers?
 - Number of nodes
 - Amount of memory being managed

Scalability in Cluster Size



Scalability in Memory Capacity



JumboMem Goals

- *Unmodified binaries* should be able to access all of the memory in the cluster
- No administrative access needed to install or run
 - Precludes modifications to the OS paging code
 - Precludes use of a networked swap device
- No limits on scalability
 - Precludes keeping track of every global page
 - ◆ On a 1 TB system, 1 word/4KB page = 1 GB of bookkeeping memory

JumboMem Implementation

- Every page has a remote home node (slave)
 - Master is 100% cache—owns no pages of its own
- Custom `malloc()`, `free()`, etc.
 - Return address space w/o backing store
 - Use `LD_PRELOAD` to replace existing functions
 - ◆ User runs “`jumbomem -np <nodes> <program> <arguments>`”
- **SIGSEGV** handler traps page faults
 - Select page to evict
 - Send victim page data to page’s home node
 - Unmap victim page
 - Map replacement page
 - Request replacement page data from page’s home node

JumboMem Implementation (cont.)

- Modular design
 - Easy to replace the communication layer
 - ◆ MPI, SHMEM, ARMCI
 - Easy to change the page-replacement algorithm
 - ◆ Pseudo NRU, true NRU, NRE (not recently evicted), FIFO, random
 - Easy to select the mapping of pages to home nodes
 - ◆ Striped, blocked
 - Easy to alter the JumboMem page size
 - ◆ Any multiple of the OS page size
- Favored simplicity of initial implementation
 - Assume homogeneous nodes
 - No fault tolerance, memory harvesting, job migration, multiple masters per node, nodes that are both masters and slaves, etc.

Tricky Bits

- Challenges

- No ordering between JumboMem `init()` and C++ constructor `new`
 - ◆ `LD_PRELOAD` guarantees only symbol loading, not execution order
- `read()` and `write()` fail on unmapped pages without faulting
- `mmap()` may allocate in the middle of a JumboMem page
- OS may still evict pages to disk at any time
 - ◆ `mlockall()` is a privileged call

- Limitations

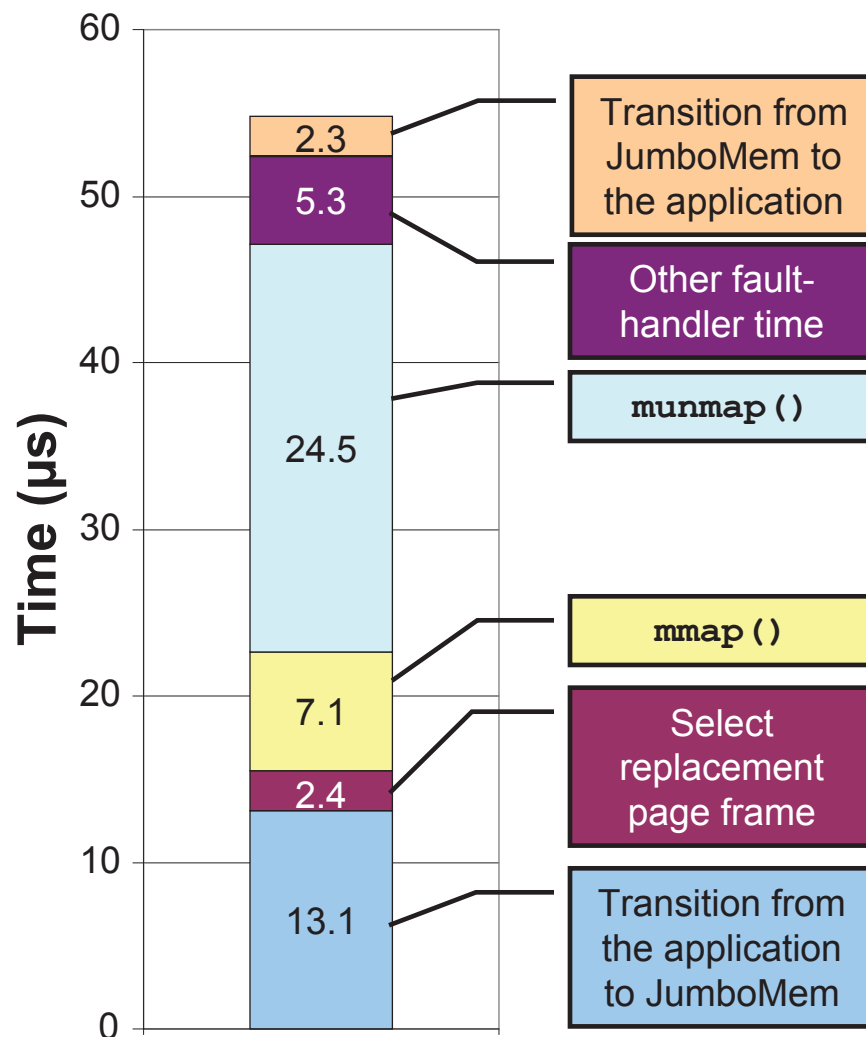
- Large, static memory blocks don't call `malloc()`
 - ◆ Not a significant problem in practice
- Threaded programs fail
 - ◆ Race between one thread accessing a page and another faulting it in

Evaluation Methodolgy

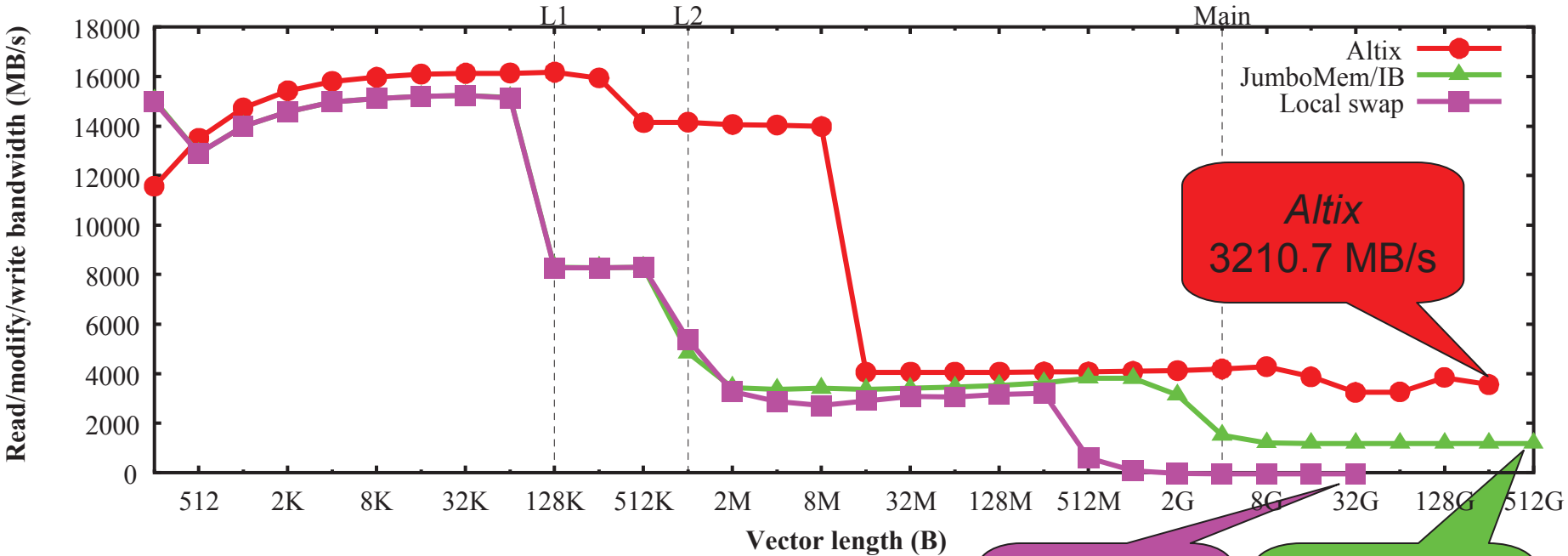
- Measure from small to large pieces of code
 - JumboMem primitives
 - Microbenchmarks (CacheBench)
 - Small programs (GNU *sort*)
 - Full applications (GNU Octave)
- Measurement platform
 - 256 dual-socket, dual-core 2.0 GHz Opterons
 - 4 GB memory per node
 - 4X InfiniBand network with OpenMPI
 - 120 GB local disks with 8 MB cache (8.3 ms avg. seek)
 - Debian Etch
 - #132 on the Top500 list when introduced in November 2005
 - ◆ Dropped off the Top500 list in June 2007

Primitive Performance

- Per-page cost of 54.7 μs
 - Large error bars ($\pm 50\%$)
- Communication cost (b bytes)
 - $2.45 \times 10^{-3}b + 81.3 \mu\text{s}$
 - Very tight fit (see paper)
- Dominant cost is communication
 - 78.9% for 4KB pages, 95.3% for 1MB pages
 - Kernel-level memory servers pay similar communication cost \rightarrow upper bound on potential performance improvement
 - JumboMem can amortize fault cost with larger pages; kernel-level memory servers can't



Microbenchmark Performance

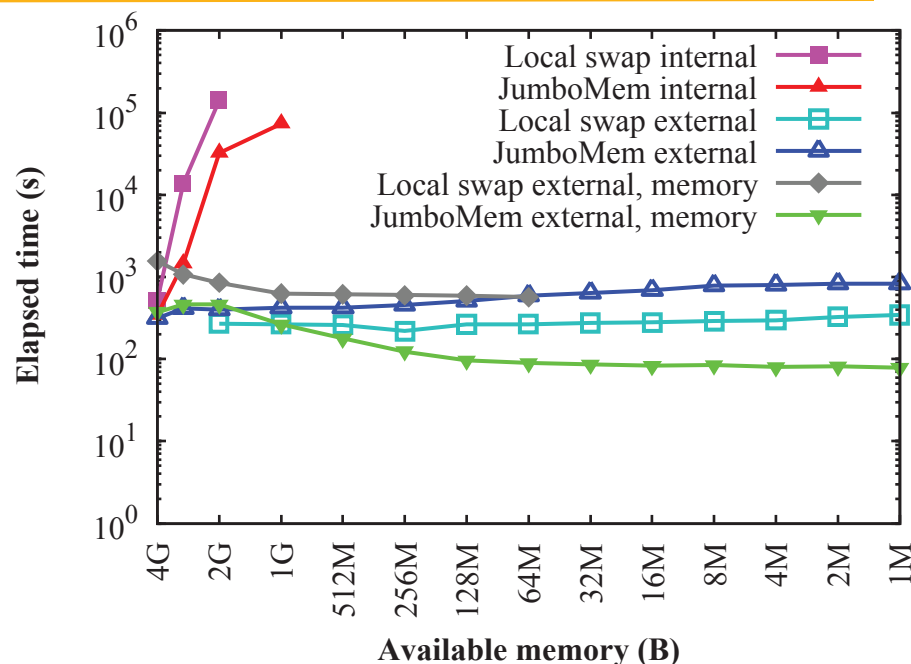


- CacheBench out to 0.5 TB

- JumboMem has stable bandwidth (up to 1000X local swap)
- JumboMem is 38% as fast as hardware-based remote memory

Small-Program Performance

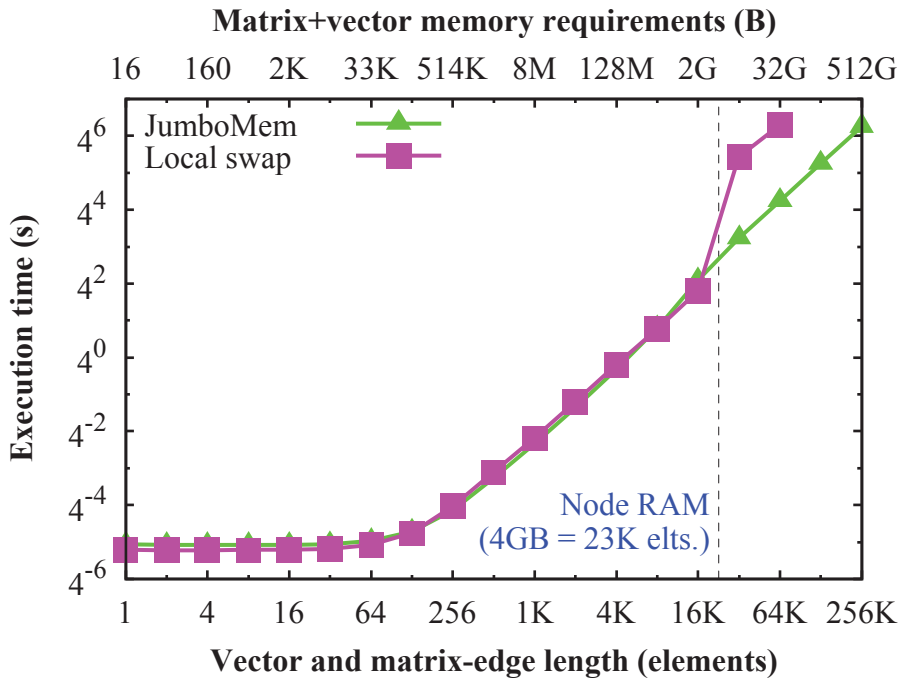
- Common to evaluate memory servers with sort performance
 - Typically measure call to `qsort()`
- We do a more thorough study
 - Internal vs. external sorts
 - Unmodified vs. modified GNU *sort*
 - Include time to read/write data
- Methodology
 - Sort 2 GB file (2^{25} lines of 64 characters)
 - Artificially limit master memory



- *Internal sort*: JumboMem 4X faster
- *External sort*: local swap 2X faster
 - Culprit was 3.5 μ s overhead per write, times 10^7 – 10^8 64B writes
- Reducing disk activity makes JumboMem 4–6X faster than swap

Full-Application Performance

- GNU Octave
 - MATLAB-like math program
- Methodology
 - Entered matrix-vector multiply script at Octave prompt
 - Multiplied up to a 262,144×262,144 matrix by 262,144 element vector (512 GB)
- Idea is to see how JumboMem works in an *interactive* scripting environment
 - Not intended to be a realistic problem



- Perfect scaling with no additional overhead over swap
 - 4X problem → 4X time
- At 64Kelts/edge, JumboMem takes 0:05:58 vs. 1:43:08

Conclusions

- Can one construct a *transparent*, entirely *user-level* memory server?
 - Yes!
 - Use `LD_PRELOAD` and a `SIGSEGV` handler
 - Key limitations include static data structures and multithreading
 - Performance should match kernel-level servers
 - ◆ Dominant cost is communication, not fixed overheads
- How scalable are memory servers?
 - Quite scalable—no inherent limitations to scalability
 - All graphs show steady performance out to 256 nodes and 0.5 TB
 - JumboMem demonstrated at 15X the nodes and 30X the memory of any prior study