Project Report: Network RAM over MPI

EJIKE, BRIAN CHUKWUJEKWU
14337107

EEEE4034/H64ACE:
Applied Computational Engineering with Project

UNIVERSITY OF NOTTINGHAM
MSc. ELECTRONIC COMMUNICATIONS AND
COMPUTER ENGINEERING

September 24, 2019

1 Introduction

With the advent of so-called Big Data in applications today, there is often a need to operate on vast amounts of data, often in the hundreds of gigabytes and well into the terabytes. However, the installed RAM on single compute nodes, hardly ever more than 100 GB and often a lot less, is typically a lot smaller than the size of these data sets. This may be seen as an opportunity to parallelize the intended application on multiple nodes in a cluster and so simultaneously achieve the necessary memory as well as the compute power of multiple devices, working on the same task.

However, parallelization can often be complex and difficult depending on the algorithms employed by an application. It can sometimes be preferable to gain access to all the available memory on a cluster while retaining the sequential nature of an application. The implementation of this alternate solution, a sort of 'distributed swap', is the goal of this project. The result will permit a serial program to access all the memory of a networked cluster of nodes, potentially in the hundreds of terabytes, through manipulation of the 64-bit virtual address space and interception of kernel signals. MPI provides the 'transport layer' for communication and the transfer of data between the nodes. The following sections provide a background of the key elements of the solution, a description of the actual implementation and the tests that were executed to validate its functionality and measure its performance.

2 Background

The proposed solution would not be feasible without certain features of modern kernels, Linux specifically: virtual memory, lazy allocation and demand paging. These features enable operating systems to effectively manage precious RAM on devices. Since this solution was written for a compute cluster running Linux, the background provided here will be in terms of Linux behaviour.

2.1 Virtual Memory

Modern applications running on CPUs do not access physical memory directly but rather work through the abstraction of a virtual address space. The available virtual address space typically depends on the bit-ness or architecture of the operating system. For instance, a 32-bit OS and a 64-bit OS would be able to address 2^{32} bytes and 2^{64} bytes of virtual space respectively. This is in contrast with the *physical* memory accessible to a program, which is typically limited to the bit-ness of the CPU or more specifically, the width of its address bus. Thus, it is possible to have far greater virtual memory than is actually physically addressable and vice-versa.

Each running program/process possesses its own virtual address space, allocated upon launch as shown in Figure 1.

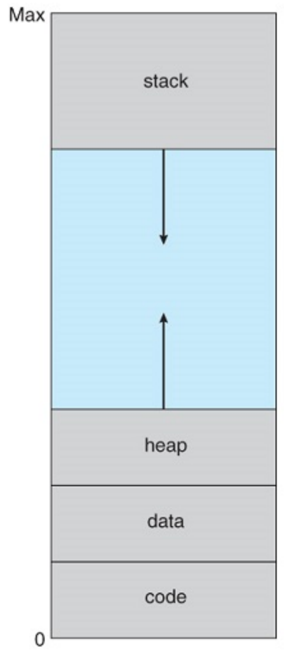


Figure 1: Virtual address space of a process [1]

When the program requests for heap memory from the OS (through calls like *malloc()* and *mmap()*), what is actually returned to the caller is a virtual address somewhere within the virtual space allocated for the heap as shown above. Upon access anywhere within the allocated area, the address is translated by the Memory Management Unit (MMU) into a physical address in RAM before the subsequent read or write operation. Virtual memory has proven immensely useful since most programs do not use all of their allocated space, at least not at once [1]. This permits the allocation of a lot more memory than actually exists physically in a system, only loading or mapping the sections that are actually needed into physical memory, in other words, lazy allocation and demand paging.

2.2 Lazy Allocation and Demand Paging

As stated earlier, when a block of memory is requested, a virtual address is returned pointing to the start of the block. However, no physical memory is mapped to the virtual memory area (VMA) yet, the Translation Lookaside Buffer (TLB) is not yet updated with the new block. The kernel simply allocates a block of virtual space, record this in its page tables, and returns the starting address immediately to the calling program [2]. When this block is eventually accessed by the user for R/W, the CPU generates a *page fault* (originating from the MMU) because it is unaware of a mapping for the faulting address.

The kernel handles this exception by checking its page tables for a valid entry containing the faulting address. Only then does it map a physical page (typically 4 kB) to the virtual page containing the faulting address [2]. It also creates a new entry in the TLB, which is a fast MMU-hosted cache for the virtual-physical address mappings already held in the page tables. Finally, the kernel returns from the exception handler and user code is resumed with the original R/W operation succeeding this time. This is the basis of lazy allocation which enables the kernel to allocate physical RAM on an as-needed basis, regardless of the actual size requested by a process. Figure 2 shows the entire virtual-physical address translation process.

So far, the presumption has been that the kernel succeeds in locating the faulting virtual address within its page tables. However, in a situation where no valid entry can be found, the kernel raises a *segmentation fault* also known as *SIGSEGV* to indicate that the program is trying to access memory for which it has no permissions. This exception is propagated to the program to give it a chance to handle the fault. If the program has no registered *SIGSEGV* handler, this results in the termination of the process. And this provides the basis for this solution: *SIGSEGV* exceptions for protected virtual space are intercepted and handled by mapping the faulting virtual space to memory on remote cluster nodes.

Demand paging is closely related to lazy allocation except it has to do with the swapping out/in of pages between physical RAM and the backing store under memory pressure. When

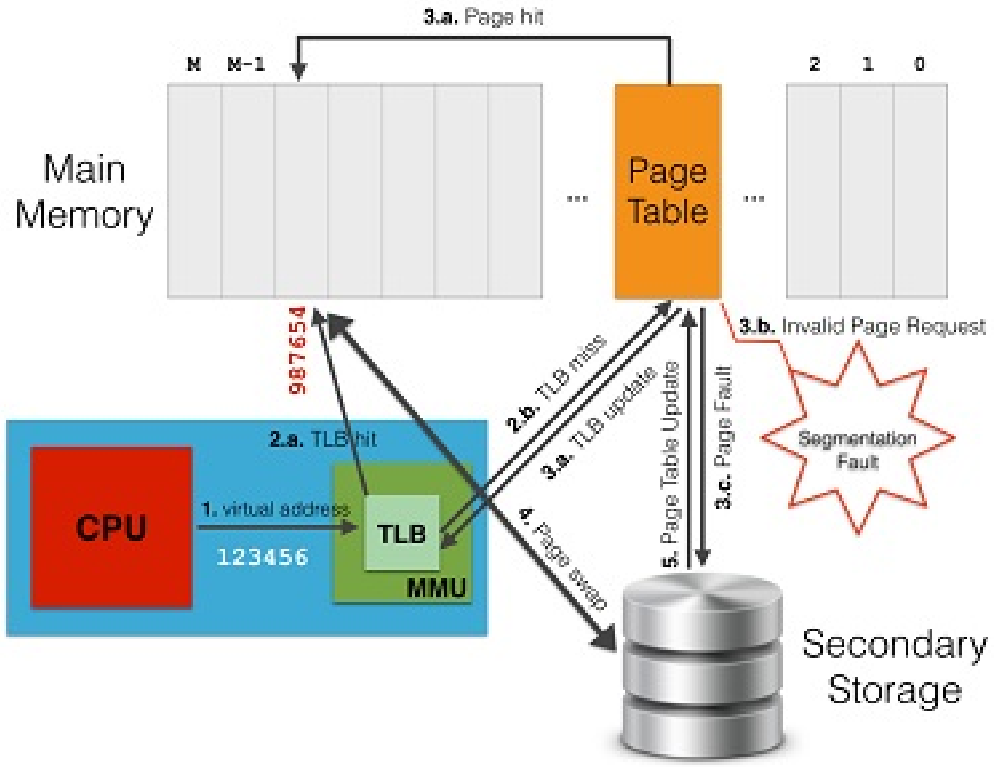


Figure 2: Virtual-physical address translation [3]

the physical RAM is used up and a process suddenly requests more pages by accessing parts of its allocated virtual area, some of the presently unused pages in physical RAM have to be swapped out of memory to disk in order to make room for the current demand. In handling the consequent page fault, some algorithm is used to determine which pages to swap out to the backing store i.e. a *page replacement* algorithm. The most common of these are:

- **FIFO page replacement**, which adds newly loaded pages to the tail of a queue and swaps out the pages at the head of the queue [1]. This is relatively simple but hardly ever optimal.
- **Least Recently Used (LRU) page replacement**, which differs from the FIFO method in that it swaps out the pages with oldest *use* time, as opposed to the oldest *load* time [1]. This method requires vital hardware support to be feasible considering it requires tracking every *memory access* (as opposed to every *page load*).

> Most hardware do not possess the features needed to fully implement this technique and so approximate methods with fewer requirements have been devised which still yield excellent results and are among the most common replacement algorithms in use [1].

Since this solution is primarily a proof-of-concept, the FIFO method was chosen for its simplicity, despite poor efficiency and susceptibility to thrashing.

2.3 SIGSEGV handling

The *SIGSEGV* signal is a key component of this solution, without which the distributed demand paging would not be possible. Essentially, when the kernel identifies an attempt by the process to access previously protected virtual space, it invokes the *SIGSEGV* handler which accepts the faulting virtual address, among other arguments. The handler in this implementation interprets the passed address and uses it to retrieve the relevant pages from the cluster nodes, or alternatively swap out some physical RAM pages to make room for the requested region.

It is considered dangerous to do much more than set flags in a signal handler, considering that signals can occur at any time and most system and C library calls are non-reentrant and also not *asynchronous-safe* i.e. capable of being called within a signal handler with absolute guarantee that the main thread will not be affected [4]. And yet, in order to ensure the smooth running of the main thread with as little awareness of the 'background' swapping as possible, most of the important operations (unlocking of VMAs, swapping, etc) must be performed within the handler, just like the kernel handles page faults. Thus, this implementation is inherently dangerous, considering a interrupted non-reentrant call in the main thread could potentially cause the program the crash as a result of undefined behaviour.

To reduce the chances of things going wrong within the handler, a lot of other signals (SIGINT, SIGTERM, etc) are blocked while the handler runs. The handler also tries to minimize the number of library functions it uses but since MPI calls also occur within the handler, there is no guarantee that some non-reentrant call like *malloc()* is not called by them somehow.

Besides consistency, the execution time of the handler is also important; it is essential that the handler complete as quickly as possible. The greatest factor affecting the handler's speed is the MPI send and receive calls which handle the swapping of data between the host and the backing store. These calls are necessarily blocking since they directly write to and read from the user-allocated space; the calls must complete before exiting the handler else the main thread, as well as MPI in the background, risk reading wrong data upon handler return. The speed of these calls in turn depend primarily on the amount of data being transferred and the underlying MPI transport layer (Ethernet, Infiniband, etc.). The transport layer becomes particularly important as the amount of data transferred per MPI call increases. Infiniband was the primary transport of choice in this implementation with Ethernet served as a fallback, albeit with significantly worse performance.

2.4 Memory Overcommit

One other relevant aspect of memory management in Linux is the concept of memory overcommitment. As explained earlier, when a call like `malloc()` is made, virtual space of the requested size is reserved for the calling process by the kernel. But this is not the end. Apart from updating its record of the amount of virtual memory reserved, the kernel additionally increments its count of the amount of *committed* memory. Committed memory essentially refers to the amount of memory for which the kernel or OS guarantees (in principle, at least) that it can find physical RAM pages or swap space when the allocation is eventually accessed. The amount of committed memory is limited by the size of the installed RAM and swap space, in contrast to virtual memory space which is limited purely by address width.

Thus, in this sense, overcommit refers to the state where the kernel commits more memory than it knows it can guarantee, in the hopes that physical memory will be eventually available when the excess committed space is accessed. The amount of presently committed memory system-wide is held in the `Committed_AS` field of `/proc/meminfo`, whereas the `CommitLimit` field holds the maximum commitable memory by all processes. This limit is calculated from `/proc/sys/vm/overcommit_ratio` % × `MemTotal` + `SwapTotal` (the latter fields are in `/proc/meminfo`).

However, this limit is not enforced, unless `/proc/sys/vm/overcommit_memory` is set to 2, which means '*don't overcommit at all*'. The default setting is 0 which basically tells the kernel to use some (obscured) heuristics to allow *some* overcommit but also limit `malloc()` and `mmap()` from going overboard with ridiculous requests. For instance, while experimenting, it was discovered that `malloc()`s above 90 GB failed on *node001*, which corresponds to about 90 % of the combined RAM and swap. `mmap()` similarly failed beyond this limit, but if the `MAP_NORESERVE` flag is specified, the limit can then be exceeded, the requested virtual space no longer counts toward committed memory and is spared the kernel's heuristics.

Another (cleaner) way to allocate pure virtual space with no commit charge, is to simply call `mmap()` with the `PROT_NONE` flag which locks the VMA, preventing physical pages from being allocated. This method is preferred because unlike `MAP_NORESERVE`, it works even when `/proc/sys/vm/overcommit_memory` is set to 2 to hard-limit committed memory, which is why it is used in this implementation. Figure 3 is a simple graphic showing the relationship between virtual memory, committed memory and physical memory.

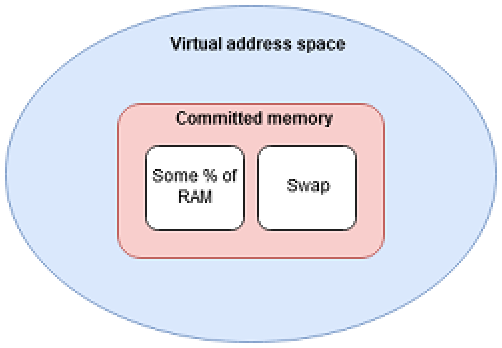


Figure 3: Relationship between virtual memory, committed memory and physical memory

3 Implementation

The solution was written in C, considering the chosen data structures are simple and greater low-level control was desired than offered by C++ built-in data structures. Using C++ would have also required calling methods of these standard data structures within the signal handler. These methods internally manipulate dynamic memory for operations like creating elements and also likely employ static variables for accounting, thus they are highly unsafe for signal handling.

This implementation, dubbed 'NetRAM', follows a typical master-slave architecture, where in this case, there are 6 compute nodes which serve as slaves. These nodes provide the backing store for the master or head node. The library and program which run on the master are different from what runs on the slaves. The master's library offers functions for:

- Initialization of NetRAM: `netram_init()`
- Memory allocation: `netram_malloc()`
- Memory deallocation: `netram_free()`
- Finalization of NetRAM: `netram_end()`

The slave's library, on the other hand, only offers functions for:

- Initialization of NetRAM: `netram_slave_init()`
- Main program loop or listener: `netram_slave_loop()`

As is, the smallest allocatable unit is 1 MB. This limit was imposed to reduce the complexity of memory accounting that would be needed for memory blocks. This unit, defined as `NETRAM_UNIT_SZ` in `netram.h`, is used throughout the library and all calls made to `netram_malloc()` are rounded up to the nearest megabyte. While this limit can be theoretically adjusted in the header file, no value other than the default 1 MB has been tested so there may yet be hidden bugs.

`netram_init()` accepts the maximum number of *units* (not bytes) the program is expected to use. This is to enable the library (and kernel, ultimately) map the entire virtual area in one call and then manage the contiguous space internally, rather than map several non-contiguous fragmented areas on demand and have to manage them. When 0 is supplied, the default number of units `MAX_MAPPABLE_UNITS` is used; this is currently equal to 500 GB and can also be changed in `netram.h`.

When `netram_init()` is called, the master initializes MPI and `MPI_Gather()`s the available capacity of each slave node. Each slave determines its capacity by simply subtracting its presently committed address space `Committed_AS` from its commit limit `CommitLimit`. The calculated capacity is not perfect, especially considering `overcommit_memory` is 0 by default and not 2, which would have enforced the `CommitLimit`. Without root privileges,

/overcommit_memory could not be changed. However, the CommitLimit still serves as a rough estimate as to how much virtual memory can be reasonably guaranteed to find backing physical RAM or swap when needed, at least at the time of the inquiry. To further guarantee availability, the slaves use mmap() with PROT_READ | PROT_WRITE, as opposed to PROT_NONE used by the master.

Based on the capacities of the nodes, the master then distributes the maximum VMA (passed earlier) between the slaves. An attempt is made to evenly distribute the space between the slaves and prevent allocating everything to a few slaves; this is to reduce the risk of later segfaults by using up too much memory on any one slave. Next, the VMA is claimed with mmap() + PROT_NONE. madvise() is also used here (as in several other places in the code) with MADV_DONTNEED to further inform the kernel that physical pages are not yet needed for this mapping.

A separate area is also mapped for holding block metadata. A *block* here refers to an allocation that has been made on behalf of the user through netram_malloc(), with a minimum size of NETRAM_UNIT_SZ. The metadata of all such blocks is held in a singly-linked list, where each node holds the size of the block and flags for knowing if the block is free and if it has ever been swapped out. The space allocated for the blocks is proportional to the number of units passed to netram_init(). This is so that the worst case of a VMA filled with tiny 1-MB blocks can be handled but the cost is space overhead in the tens of megabytes even if only one allocation is made. However, this cost was considered acceptable since the overhead is insignificant compared to the vast amount of memory this solution is supposed to provide. Besides, it has been established that space that never gets written to does not consume physical RAM pages, so you mostly pay for only what you use. Finally, the initialization routine marks the entire VMA as free, sets up the *SIGSEGV* handler and returns. The return value should be used as an indication of success or failure.

After initialization, netram_malloc() can now be called to request memory just like malloc(). It basically walks through the list of blocks (the metadata linked list specifically) and searches for one that fits the requested size. If a block is found, it is split to the exact size requested and a pointer to it is returned. The returned area is still locked and only gets unlocked after the first attempt to access it. If a size smaller than NETRAM_UNIT_SZ is passed to netram_malloc(), malloc() is invoked instead internally and its result is returned.

netram_free() similarly works just like free() and walks through the list of blocks searching for one that matches the passed address. If it is found, it is marked free, the region is locked with `mprotect()` as before, and the underlying physical pages are freed with `madvise()` and `MADV_DONTNEED`. If the desired block is not located in the list, the address is passed on to `free()` instead. Coalescing of contiguous free blocks is also performed lazily during the iteration to help reduce fragmentation.

`netram_end()` performs cleanup functions i.e. unmapping VMAs, shutting down slaves and ending MPI. The slaves are instructed to shut down through a special message with zero-valued fields. This makes them exit their listening loop so that the entire communicator can collectively finalize.

The *SIGSEGV* handler is where the most time is spent during the normal program execution. The handler executes on a alternate stack to enable it remain robust against stack overflows in the main program thread. The handler primarily ensures that the used memory on the master, dubbed 'active memory', does not exceed a set amount, `NETRAM_MAX_ACTIVE_MEM`. When the active memory is full, blocks with the oldest load time in a FIFO are swapped out to the slaves and their regions locked to make room; this is the FIFO replacement technique explained earlier.

The FIFO space is allocated statically and can hold `NETRAM_MAX_ACTIVE_UNITS` elements to accommodate the worst case. The maximum amount of data transferred during swapping is `NETRAM_MAX_TRANSFER_BYTES` and can be altered in `netram.h` like the other defines. The faulting area is then unlocked and `madvise()` with `MADV_NORMAL` is used to ensure physical pages get allocated as needed. To further reduce the effects of the handler on the main thread, `errno` is saved upon handler entry and then restored upon exit.

The slaves, on the other hand, do not do much more after initialization, where they each allocate (with `mmap()`) their own portion of the total VMA assigned by the master. Each node now simply listens for request messages from the master. Each data transfer between master and slave is preceded by one such message which contains the starting virtual address and the size of the data to be transferred. The message's tag is used to communicate if the transfer will be a READ (slave-to-master) or WRITE (master-to-slave) operation. The data transfer follows subsequently, the data directly written to or read from the corresponding region in the slave's buffer. If a slave receives a request message with zero-valued fields, it interprets that as a command to clean up and shut down.

4 Usage and Testing

This section describes how to test the library on the Jenna cluster. First, to ensure that the OpenMPI module gets loaded upon login by all nodes, the following line should be appended to `.bashrc` in the home directory:

```
module load openmpi/1.6.5-openib
```

This loads the Infiniband variant of the OpenMPI module. At first, Infiniband failed to work on the cluster, always raising errors about a missing transport module, so Ethernet was used instead. But after some digging, it turned out that the head node Jenna itself does not possess any working Infiniband interface; this was discovered through the `lspci` command which yields a list of installed PCI devices. The other 6 nodes do have Infiniband interfaces and so are perfectly capable of running MPI over that link. So instead, to execute the NetRAM code, one must first ssh into any of the compute nodes with `ssh node001`, for instance. This node now serves as the NetRAM master, while the remaining 5 serve as slaves. The head node Jenna is excluded entirely from the system.

Now, to compile the single-file C test program named `netram_test.c` (as well as any other single-file programs), the required command is:

```
mpicc lite_fifo.c netram.c netram_test.c -o netram_test -std=gnu99
```

`netram_test` can be replaced with the name of any other single-file C program, to be compiled against this library. Similarly, to compile the slave's program `netram_slave_test`, the command is:

```
mpicc netram_slave.c netram_slave_test.c -o netram_slave_test -std=gnu99
```

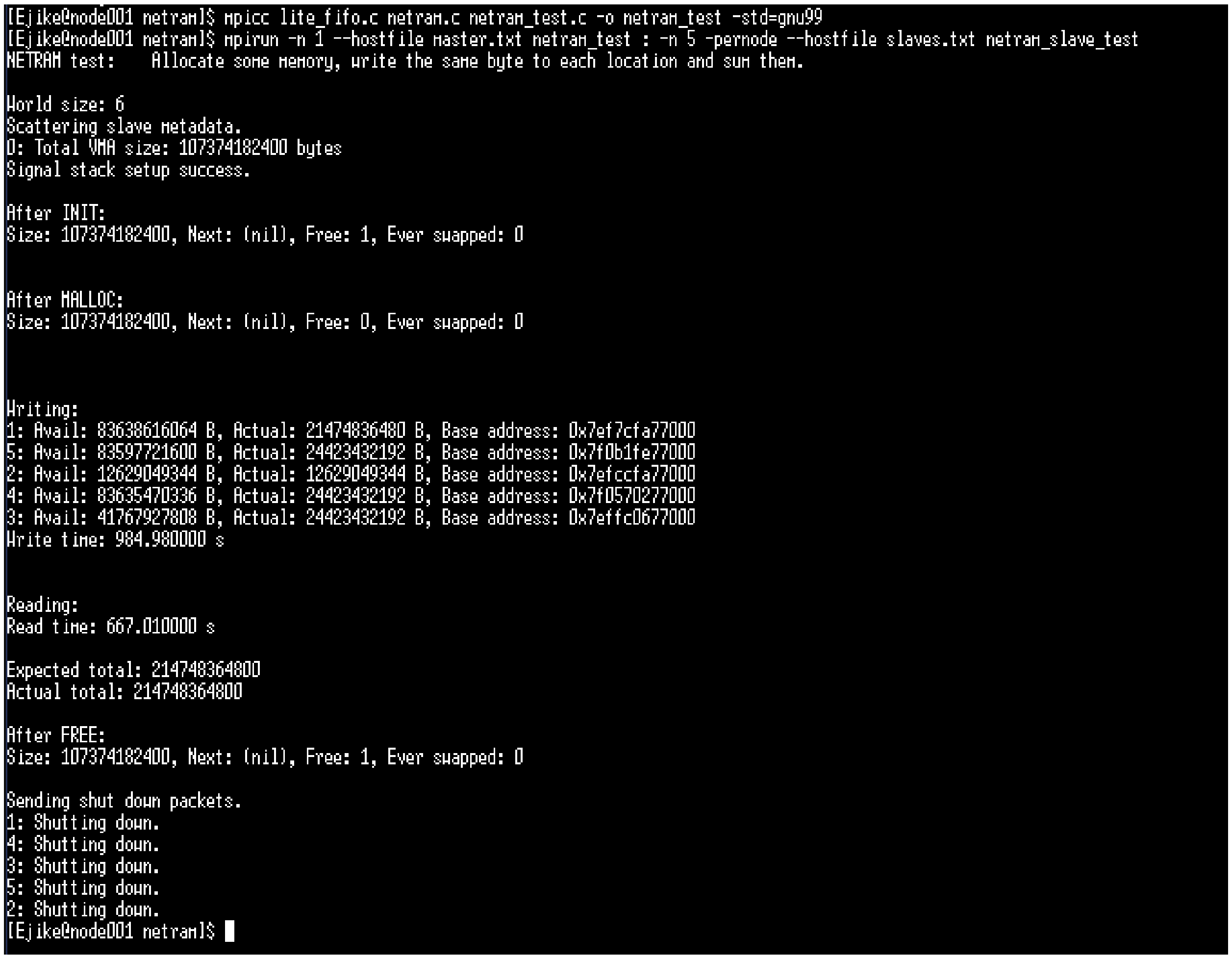
There will be no need to change the slave's code from the provided default, since all the slave is expected to do is serve as a backing storage for the master which is the one actually responsible for interfacing with application code. All files are to be saved to the same directory.

On the cluster, 6 compute nodes were identified by checking the list of hosts in `/etc/hosts` (while logged into the Jenna node). One of these nodes was arbitrarily chosen to serve as master as explained earlier, and the rest would serve as slaves, with their host names listed in a file named `slaves.txt` in the working directory. To finally run the test (or any other NetRAM application), the two compiled programs above must be launched in Multiple Program Multiple Data (MPMD) mode: one program for the master and one for the slaves. The command is shown below:

```
mpirun -n 1 --host node001 netram_test :
        -n 5 -pernode --hostfile slaves.txt netram_slave_test
```

As can be seen, the `host` switch is used to pass the name of the master (in this case, `node001`) which alone runs `netram_test` whereas the `hostfile` switch is used to pass a list of 5 slave hostnames which will run the second program, `netram_slave_test`. The `pernode` switch ensures that each slave node runs only one copy of the program since it would be pointless and inefficient to have multiple processes running on the same slave each managing their own swap.

The provided test program, `netram_test`, attempts to allocate 4 GB of space, sequentially write a constant value (2, in this case) to every address in the space, and then read and sum them. Figure 4 below shows the output of the execution, but instead with 100 GB allocated across all 6 nodes.



```
[Ejike@node001 netram]$ mpicc lite_fifo.c netram.c netram_test.c -o netram_test -std=gnu99
[Ejike@node001 netram]$ mpirun -n 1 --hostfile master.txt netram_test : -n 5 -pernode --hostfile slaves.txt netram_slave_test
NETRAM test:    Allocate some memory, write the same byte to each location and sum them.

World size: 6
Scattering slave metadata.
0: Total VMA size: 107374182400 bytes
Signal stack setup success.

After INIT:
Size: 107374182400, Next: (nil), Free: 1, Ever swapped: 0


After MALLOC:
Size: 107374182400, Next: (nil), Free: 0, Ever swapped: 0



Writing:
1: Avail: 83638616064 B, Actual: 21474836480 B, Base address: 0x7ef7cfa77000
5: Avail: 83597721600 B, Actual: 24423432192 B, Base address: 0x7f0b1fe77000
2: Avail: 12629049344 B, Actual: 12629049344 B, Base address: 0x7efccfa77000
4: Avail: 83635470336 B, Actual: 24423432192 B, Base address: 0x7f0570277000
3: Avail: 41767927808 B, Actual: 24423432192 B, Base address: 0x7effc0677000
Write time: 984.980000 s


Reading:
Read time: 667.010000 s

Expected total: 214748364800
Actual total: 214748364800

After FREE:
Size: 107374182400, Next: (nil), Free: 1, Ever swapped: 0

Sending shut down packets.
1: Shutting down.
4: Shutting down.
3: Shutting down.
5: Shutting down.
2: Shutting down.
[Ejike@node001 netram]$ █
```

Figure 4: NetRAM test results for 100 GB allocation

The allocated space on each node is shown, roughly 20 GB per slave node, as well as their individual base virtual addresses. It can be seen that writing to all the locations took 984 seconds ≈ 16 minutes, while reading took 667 seconds ≈ 11 minutes. In total, it all took about 30 minutes. Since the actual sum matches the expected sum, the test proves that at least, allocations across the nodes works as desired, including the swap mechanisms handled through *SIGSEGV*. The execution speed however is poor, considering this is a single simple iteration through the allocated space, unlike a real application which would likely perform such iterations multiple times and possibly in a random manner.

Further tests could not be carried out due to insufficient time as well as unreliability in Jenna, which became very slow for some reason in future logins, with `MPI_Finalize()` on the master suddenly throwing segfaults at the end of every execution, even when they complete successfully with the expected output.

5 Conclusion

As is, NetRAM works but with poor performance, at least relative to the speed of accessing physical memory, even including swap. Infiniband proved much faster than Ethernet but still incapable of matching the speed and low latency of physical memory access, especially with all the MPI overhead. There are also a lot of improvements to be made in the implementation, especially regarding the way swapping is handled. However, this implementation at least provides a proof-of-concept to be built on, even if tentatively for single-threaded applications alone, considering its terribly unsafe use of signal handlers renders it infeasible for multi-threaded applications.

References

[1] UIC Computer Science, "Virtual Memory." https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/9_VirtualMemory.html. Accessed: 27-05-2019.

[2] M. Porter, "Virtual Memory and Linux," in *Embedded Linux Conference, Europe*, pp. 50–91, October 2016. Accessed: 27-05-2019.

[3] Gabriele Tolomei, "Virtual Memory, Paging, and Swapping." https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/virtual-memory-paging-and-swapping. Accessed: 28-05-2019.

[4] GNU, "Signal Handling." ftp://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_24.html. Accessed: 31-05-2019.