Transparent Distributed Shared Memory Library

Robert Gasparyan, Angela Gong, Judson Wilson

Introduction

Distributed shared memory (DSM) allows physically separate memory (in multiple machines) to be addressed as one shared address space [1]. This enables applications to scale across multiple machines while preserving the shared-memory interface of a single machine. We implemented a DSM system in userspace with a familiar, transparent interface, which is both correct and scalable.

Consistency Model

The most common method for memory synchronization is the use of mutual exclusion locks (mutexes). We use release consistency [2] semantics to provide the guarantees needed for this model. Mutexes are acquired from a master server, then an acquire operation is performed to protect memory pages so that upon the next access a fault handler fetches the latest version of the data. Unlocking is preceded by a release operation, which fetches the latest version of the pages (discussed below), applies any local changes, and increments each page version. When using these mutex locks, any memory modifications in a critical section that precedes the current critical section (guarded by the same locks) will be observable.

Note that such mutexes do not communicate to the system "what", if any, memory they are protecting, so release and acquire apply to all pages. Also, the protected memory objects may be on the same page as objects protected by other locks, so we use difference-based patching to infer sub-page modifications. On release, we patch our local differences on top of the most recent page version, which must be fetched if there are new modifications.

Interface

The following code illustrates use of the interface for a simple program which does only 1 locked operation:

```
dsm_share(shared_region_addr, length);
dsm_start(argc, argv); // Args from deployer
...
dsm_lock(LOCK_A);
/* Use memory protected by LOCK_A */
dsm_unlock(LOCK_A);
...
dsm_wait(); // Wait until others finish
```

Figure 1. Extra code needed for DSM to work.

Only 3 calls are needed for setup and teardown. Memory sharing is completely transparent, and locking calls are similar to those used in any threaded application with shared memory.

* See the following pages for test and benchmark results.

Application Deployment

We've created an easy-to-use deployment system that distributes a program binary to all machines and allows successive deployment of different programs. A deployment server is started at one machine, and daemons that connect to the server are started on all other machines. The library takes care of packaging up binaries and sending them off to the daemons to be executed.

Transparent Memory Consistency

To update memory contents and track modifications transparently, the `mprotect` interface is used to protect pages. Upon an acquire operation, any unprotected pages are protected from access. A `SIGSEGV` fault causes a page to upgrade to read access after fetching the latest data. A further fault indicates a write, so protection is lifted and the page is marked as modified. Release operations do versioning operations, clear the modified state, and write protect. All visible data synchronization is done within fault handlers, lock/unlock and acquire/release calls.

System Organization

Our system involves several networked processes, one of which is the master, and the rest which are application workers (see Figure 2). From the application thread, workers make requests to the master server to acquire a lock and discover and updated page versions. They get the latest page data from other workers' "page servers", a background thread that serves page data requests from other workers (see Figure 3). Page data fetching and version updatings require the workers to message the master for version info, and page servers for data. Lock requests received at the master spawn threads which acquire a corresponding `pthread_mutex_lock` before responding.

Communications

The master server and page servers maintain TCP connections to every worker to receive requests. The servers block using `epoll` to wait for messages from any connection. Messaging is done through our `msg` system which abstracts TCP data streams into distinct message units, coalescing reads as needed.

Conclusion

We implemented a system that provides transparent memory synchronization using a standard mutex lock model. We demonstrated that it operates correctly. Due to the short duration of this project, we were unable to make many scalability enhancements we desired, and instead focused on correctness first. Also, we were originally going to intercept standard library interfaces for transparency, but found this not to be useful for our final model.

Correctness Test

To test correctness of our system, we made a DSM application called "`lock_races`" which randomly chooses between a set of 11 operations that each modify 2 shared counter-variables (out of 4 total, spanning 2 pages), each protected by their own lock. The locking order is composed in different ways, including nested locks, maintaining a discipline to prevent deadlock. Equal counts are added to shared variables and corresponding local counters, or "moved" between the two shared counters. Two example operations appear below, where `*A`, `*B`, `*C`, and `*D` are the shared variables, and `a_counts`, `b_counts`, `c_counts`, `d_counts` are the local counters:

```
...
case 2:
  dsm_lock(LA);
  dsm_lock(LC);
  ++*C; ++c_counts;
  dsm_unlock(LC);
  ++*A; ++a_counts;
  dsm_unlock(LA);
  break;
case 3:
  dsm_lock(LA);
  dsm_lock(LD);
  temp = *D;
  *D = 0;
  dsm_unlock(LD);
  *A+=temp;
  dsm_unlock(LA);
  break;
...
```

These operations should maintain the invariant that the sum of the shared counter variables equals the sum of ALL the local counter variables across all workers.

The test was run using 3 workers doing 10,000 operations each. Afterwards, we verified the invariant was maintained, indicating that the system appears to correctly provide the desired semantics and is deadlock free. This procedure unearthed many bugs, often in conditions that we assumed would be bug free. All were found and fixed.

Benchmarks

We ran two benchmarks with and without our DSM library for performance comparisons. Our setup for benchmarking without DSM is a single machine with 20 CPUs, 64 GB of RAM, running on Ubuntu 14.04. Our DSM setup is on 10 Amazon EC2 m3.medium instances running Ubuntu 12.10, each with 1 CPU and 3.75 GB of RAM. We plot the inverse of completion time, which is effectively the total rate of work completion.

Matrix Multiplication

Matrix multiplication is computationally intensive and ideally has linear speedup as the number of processors increase. We implemented the most naive $O(n^3)$ multiplication algorithm so we could associate speedups directly with the number of processors instead of with clever speedups and blocking found in better multiplication algorithms. See Figures 4 and 5.

Word Count

Our other benchmark was a simple word count on a large corpus. This is an I/O-bound process that would see a benefit if run on multiple machines. See Figures 6 and 7.

Analysis

Our results are quite interesting. We notice on a single machine that matrix multiplication stops scaling after about 9 cores. This is probably because the operations are saturating the memory bus or there is system call-related contention, so that more CPUs doesn't actually improve anything. We suspect this is due to the network latency -- the overhead of 10 machines communicating with each other over the Internet masks any improvement in speed due to DSM. We had to scale down the size of the matrices on our Amazon tests, for reasons that we have not yet been able to investigate. Larger matrices should lead to more computation per unit network traffic, and yield better results.

As described earlier, word count is IO-bound and we see this in the single-machine graph. After about 6 CPUs, the runtime of word count is pretty much constant. This is because each core is competing with each other for read locks, and thus having more cores doesn't allow quicker file reads. On the other hand, word count scales very well with our DSM library. This is because each machine can read the files separately without overrunning each other.

Generally speaking we were focused on correctness over scalability - a terribly broken system is worse than one which does not perform well. There are many places we can improve scalability in the future. We found that the C standard library is difficult to use for such purposes, because it lacks basic things such as hash maps, and efficient parallel socket communications code tends to have many lines of text to do what would seem like simple tasks. Such issues forced us to make compromises, given our limited time.

References

[1] Nitzberg, B. & Lo, V. (1991). Distributed Shared Memory: A Survey of Issues and Algorithms. Distributed Shared Memory-Concepts and Systems. (pp. 42-50).

[2] Gharachorloo, K., et al. (1990). Memory consistency and event ordering in scalable shared-memory multiprocessors. ACM. (Vol. 18, No. 2SI, pp. 15-26)

Figures

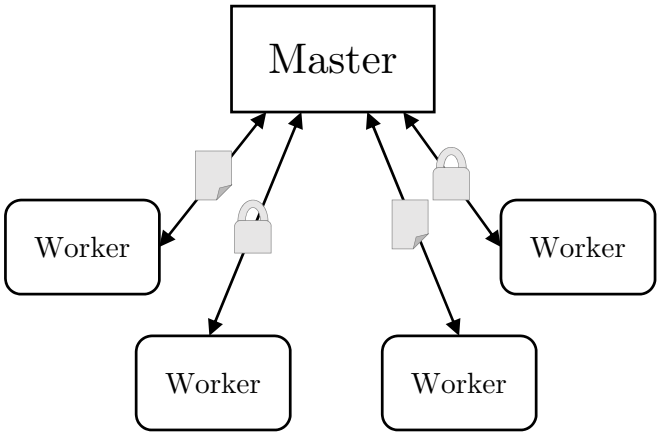

Figure 2. Workers connect to the master server to for lock requests and page version number synchronization.

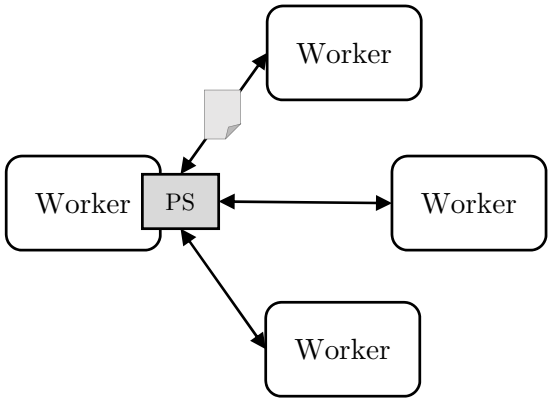

Figure 3. Page server running on a worker serves requests for pages owned by this worker.

Figure 4. Matrix multiplication on single machine without DSM.

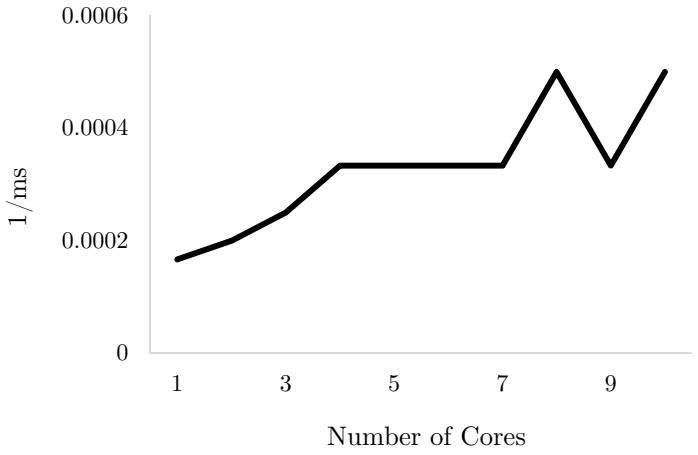

Figure 5. Matrix multiplication using distributed shared memory.

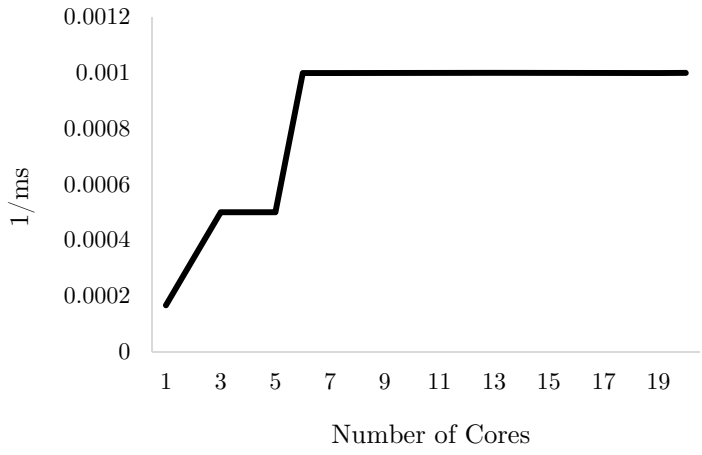

Figure 6. Word count on single machine without DSM.

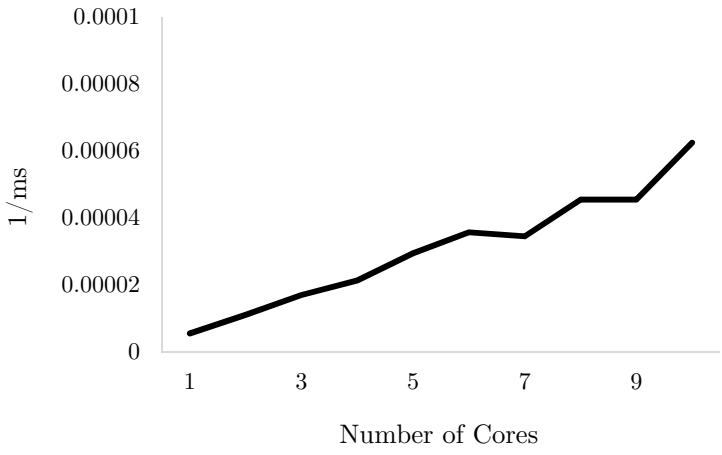

Figure 7. Word count using distributed shared memory.