libdsmu (Distributed Shared Memory in Userspace)
6.824 Final Project

Webb Horn, Ameesh Goyal, Tim Donegan, Julián González
Available on Github: https://github.com/webbhorn/libdsmu

Introduction

libdsmu is a Distributed Shared Memory system similar to Ivy [1]. It is implemented as an application library that runs completely in userspace, easily turning a parallelizable Linux application into one that is DSM-capable. Clients of our library call an initialization function that prepares the program to run in a DSM style, and then resume execution normally. We require that clients of our library implement their programs in a style that handles parallelism on their own. For example, there might be a shared struct in memory that each processor will consult to learn about the work that it should execute. Alternatively, arguments provided to a program at runtime indicate its shard of the work. In addition to the DSM library, we have created a matrix multiplication benchmark to evaluate the performance of our system.

Design

libdsmu is designed as a client library for C programs and a standalone manager server. The coherence strategy is based on the invalidation protocol described in Ivy: the manager enforces the per-page invariant that either (1) a single client has write access to a page with no clients having read access, or (2) no clients have write access to a page, and many can have read access. This strategy yields an intuitive “strict consistency” memory model for application programmers.

libdsmu does not require any modifications at the kernel level—as its name suggests, it is implemented entirely in userspace. There are no fault tolerance guarantees or checkpointing systems in our design.

Although a distributed manager would have been more efficient (see Extensions), we decided to use a centralized manager server to simplify the implementation of our system, given our time constraints.

Implementation

Client Library

Applications link in our client library to use libdsmu. The client library must be initialized on startup. The initialization procedure creates a connection between the client and the manager, registers libdsmu’s custom page fault handler with the kernel, and prepares the data structures required to coordinate shared pages. The application must also indicate to the library what regions of memory are meant to be shared between clients. Finally, the initialization procedure creates a thread that listens on the socket for new messages from the manager. We use pthreads for synchronization and threading primitives.

The client library maintains an array of pthread mutexes for each shared page with an associated condition variable. We also keep an array that tracks shared memory regions.

The library registers libdsmu’s page fault handler as a signal handler for the SIGSEGV signal. Linux sends the SIGSEGV signal to a process when the process causes a segmentation violation (including page faults). Our signal handler determines whether the access violation occurred in a shared memory region. If it did, it checks the trapped error code to determine if the fault was a read or a write fault. If the fault did not occur in a shared memory region, the signal handler forwards the signal to the previously registered SIGSEGV handler. Because our mechanism for determining whether a fault was caused by a read or write access depends on the value in a particular register, our solution is non-portable across platforms (non-heterogeneous).

When a client requests that a particular region of memory is shared, libdsmu invalidates read and write access permissions to the page using the mprotect() system call. The library can also zero out the newly shared page with mmap() to guarantee that all nodes are initialized with the same shared memory contents.

After initialization, there are only two ways that the client library is invoked.

The first occurs when the background thread receives a message from the manager. When an invalidate message arrives, the client library invalidates the requested page, and replies with a confirmation. The confirmation includes a base64-encoded string of the invalidated page's contents if the manager requested it. When a read or write request confirmation arrives, the page's data is copied into the page with memcpy(). We also set the correction permissions for the newly updated page with mprotect(). Finally, a condition variable for the page is signaled so that the waiting page fault handler can return control to the application.

The second occurs during a page fault. On a page fault, Linux sends the SIGSEGV signal to our process, which is handled by our page fault handler. If the fault is an access violation in shared memory, we request read or write access (depending on the fault type) from the manager and wait until the confirmation arrives by calling wait() on the condition variable for that page.

Manager server

Our manager server (written in Python) keeps a page table entry for each shared page and a dedicated socket for each client. When a client request for page access arrives, it is handled in a new thread that acquires a lock for the page. Therefore, only one request for each page is handled at a time. The manager server also caches pages received from clients to improve response time.

Protocol

libdsmu's design choices of a Python server and C clients necessitated an RPC system that could communicate across both languages. We chose to implement our own RPC structure for simplicity. Originally, an attempt was made at using SCTP, which provides datagram transport on top of TCP's stream transport. However, SCTP completely failed on our test setup, and debugging such an uncommon protocol was difficult.

Instead, we implemented a simple datagram transport scheme on top of TCP by prepending the size of the RPC message to each message. We used a MSG_PEEK/MSG_WAITALL scheme to ensure that we were handling entire messages and not parsing requests midstream.

Performance

Amazon EC2 Test Setup

We used 10 Amazon EC2 m3.medium instances running Ubuntu 14.04 to performance test libdsmu, all located in the same availability zone (us-east-1a). One instance served as the manager server, and the other 9 were used as clients to perform tests. A testing script ssh'ed into each client using the appropriate key and concurrently launched the tests across the instances at the same time. To perform speedup testing, we ran tests using a range of 1 to 9 instances.

Matrix Multiply

Matrix multiply is a good benchmark for DSM systems because it performs a lot of computation that can be evenly split over processors. We created a matrix multiply application that uses the naive $O(n^3)$ algorithm. We made two versions of the matrix multiply app. The first version assigned alternating rows to alternate processors. For example, if there were three processors, the first processor would be responsible for rows 1, 4, 7, etc of the matrix. The second version of the app assigned chunks of rows to different processors. For example, the first processor would be responsible for the first third of the rows. The second version of the app is better for our system because it has less page contention, particularly for large matrices. Since matrix rows are stored sequentially in memory, sequential rows are likely to be on the same page. Of course, when the size of the matrix is large enough, a row can take up an entire page on its own. By manipulating the size of the matrix and the starting address of the matrix, we can manipulate the amount of contention for write access for pages that will be present in the execution of the program.

Results

Figures 1 and 2 describe libdsmu's performance results. Figure 1 has zero contention for write access to pages because the rows are page-aligned at 8192 bytes (2 pages). It was generated by the second version of the matrix multiple program described above. Clearly, it performs extremely well and demonstrates nearly linear speedup.

Figure 2 has significant contention for write access to pages because the rows are less than 4096 bytes (<1 page) and are not page-aligned. It was generated with the first version of the matrix multiply program, where rows are alternately assigned to processors. Due to all the page-contention, this test resulted in almost no speedup at all.

As a control, we also ran a 2048x2048 matrix multiply on one of our test instances locally, without using libdsmu. Our control ran only 4.97% faster than a libdsmu test with 1 client and 1 server. Thus, the overhead of libdsmu is small and reasonable.

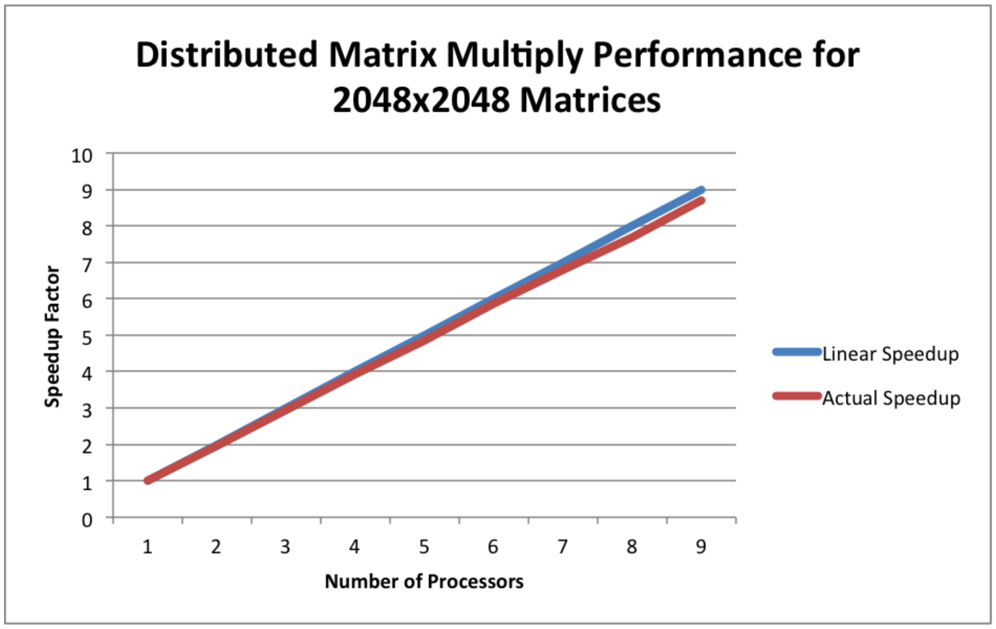

Figure 1: Matrix multiplication speedup for page-aligned matrices, with no contention.

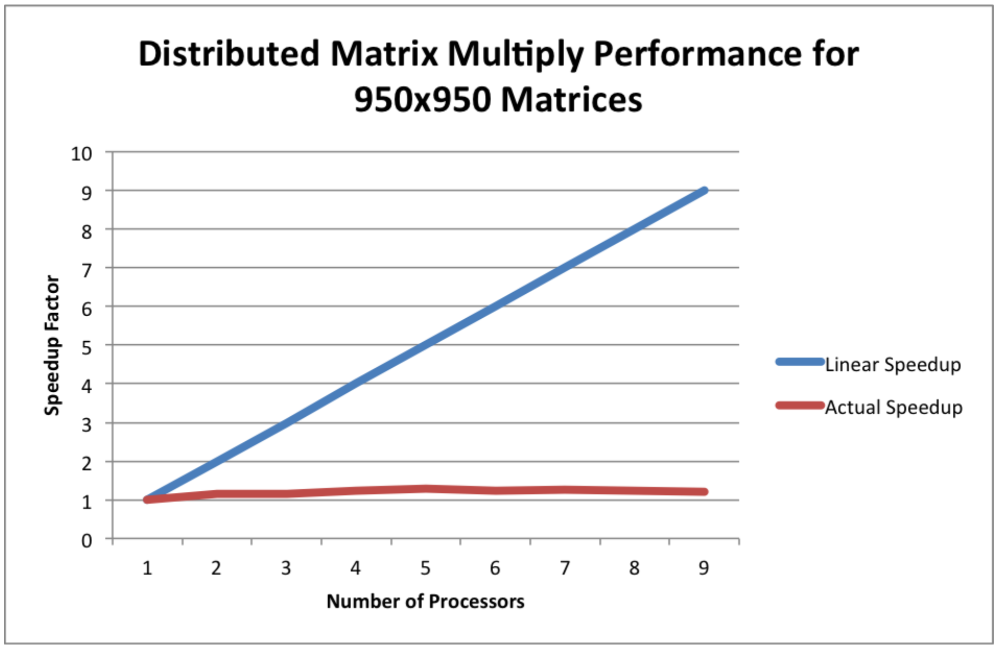

Figure 2: Matrix multiplication “speedup” for non-page-aligned matrices, resulting in page-contention.

Extensions

As demonstrated above, libdsmu can perform very well for parallelizable applications that do not exhibit very much page-contention. However, some extensions are necessary or useful to turn libdsmu into a production system. For one, authenticated RPC messages were out of scope for our project. In order to be used as a production system, libdsmu would need to secure its client-server communication. Fortunately, the primitive RPC currently implemented in our project can be easily switched out with any off-the-shelf RPC system, including those that provide authenticated messages. Furthermore, libdsmu's manager server is a single point of failure, and can possibly become a bottleneck for performance. A useful extension to libdsmu's current design would be to implement a distributed manager server, akin to the one described in Ivy. Even if the role of the manager is not completely distributed, performance can be improved by having clients send pages to each other without using the manager as an intermediary, as discussed in [1].

References

[1] Ivy: http://css.csail.mit.edu/6.824/2014/papers/li-dsm.pdf

[2] TreadMarks: http://css.csail.mit.edu/6.824/2014/papers/keleher-treadmarks.pdf

[3] Distributed Shared Memory: A Survey of Issues and Algorithms: http://www.cdf.toronto.edu/~csc469h/fall/handouts/nitzberg91.pdf