George Hatzis-Schoch (ghatziss)
Charles Wang (cwang3)
Project Report
May 6, 2019

A Lock-Free Scheduling Optimization of Spark98's Lock-Based Shared Memory Symmetric SMVP

Summary

We optimized Spark98's shared memory lock-based implementation of sparse matrix vector product (SMVP) by implementing a scheduling approach to partitioning the sparse matrix array, removing the need for locks and allowing better write-locality at expense of read-locality and pre-processing the matrix. Our implementation achieves a peak throughput difference of about 3500 MFlops over the lock-based (LMV) implementation.

Background

The goal of the Quake project at Carnegie Mellon is to develop the capability of simulating the ground movement during strong seismic events using computer simulations. These simulations can be used to predict what areas of a section of the Earth are likely to be more affected during an earthquake. This in turn could help emergency resources to plan responses to earthquakes, saving lives in the process.

The Quake simulations work by inputting a three-dimensional mesh graph representing a portion of the earth with undirected edges labeled with the stiffness of adjacent nodes (the stiffness between two nodes captures the motion of the nodes in relation to each other). The undirected attribute of the graph implies that the matrix of the graph's stiffness values is symmetric. The simulation consists of repeatedly performing a Sparse Matrix Vector Product (SMVP) operation on this symmetric matrix. The repeated SMVP operations "accounts for 70% of the simulation's computation time", and so optimizing this operation would lead to much more efficient earthquake simulations (O'Hallaron 5).

The mesh graph is input to the SMVP program in the form of a symmetrix compressed sparse row (CSR) matrix A, where each element is a 3×3 matrix of floating point values

representing the stiffness of the Earth. Due to the data-intensive needs of accurate simulations, it is crucial to efficiently use memory. Thus, only the non-zero elements along the diagonal and in the upper triangle of the symmetric stiffness matrix *A* are stored in memory for use in the simulation. Taking advantage of the symmetric property of *A* during SMVP in order to preserve memory makes the parallelization and optimization of the program more difficult than simply blocking off the matrix and assigning threads. We begin by introducing the basic steps of the computation below.

SMVP computation structure

- Read in input graph mesh.
- Create symmetric sparse matrices *A, A'* and dense vectors v, v'.

  For *i = 1* to # *steps*:

  w = SMVP(A, v)

  w' = SMVP(A', v')

Key observation:

- Two SMVP operations are performed back to back in order to "approximate the worst case cache behavior of the SMVP when it is used in a real finite element application" (Spark98 Tech Report pp. 5).

Spark98 has multiple implementations of the above computation using different approaches, including shared memory using reductions (RMV), message passing (MMV), lock-based shared memory (LMV), hybrid shared memory and message passing (HMV), and sequential (SMV) implementations. The program we focus on is LMV, which uses locks to protect writes to the output vector of the SMVP operations. The general flow of the LMV computation is below.

LMV computation structure

1. Each processor shares a memory address space and is assigned a contiguous set of rows partitioning *A*.

2. The output vector *w* is locked using a variable amount of locks *L* passed as input to the program (where each index *i* in *w* is managed by lock $i \% L$).
3. Each processor loops over it's rows.

For each row *i:*
For $A_{i,j}$ on row *i*:
$sum \mathrel{+}= A_{i,j} * v_j$
$w_j = w_j + A_{i,j} * v_i$
$w_i \mathrel{+}= sum$

Key observations:

- For each row *i* in A, one read/modify/write is performed on w_i
- For each $A_{i,j}$ on row *i* in *A*, one read/modify/write is performed on w_j

Due to exploiting the symmetry of the matrix, elements in the upper triangle now contribute to two unique elements of the output vector w. More specifically, each A_{ij} in the upper triangle contributes to the entries w_i and w_j. The challenge is that as LMV scans a row of *A* and computes w_i ultimately using one write to w_i after scanning row *i*, the program must update w_j for each element A_{ij} along the row. Since w_j varies it's index in w irregularly (as *A* is sparse), this gives very poor spatial locality and contention between the processors for the locks in *w*.

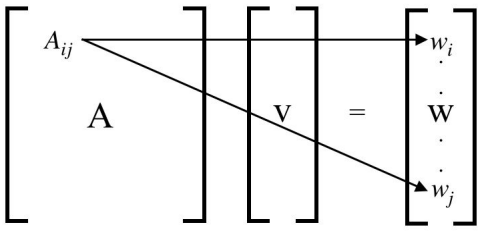


Figure 1: Symmetric Matrix Vector Multiplication Diagram

Approach

After speaking to Professor O'Hallaron after his guest lecture, we learned about the Spark98 SMVP kernel program. Spark98 has several implementation of the SMVP operation, which we described above. After discussing this kernel with Professor O'Hallaron, the LMV

program gave insight into a potential area for optimization, namely removing the locks protecting the output vector and improving the poor write locality arising due the symmetric nature of how the matrix is stored.

When removing locks, we cannot assign each thread to a continuous partition of rows due to contention for writes to *w*. Instead, we partition the output vector *w* into contiguous chunks, assigning each thread a contiguous chunk of *w* corresponding to rows in *A* containing roughly $\frac{\#\ non-zero\ elems}{\#\ threads}$ elements. Next, our implementation allocates space for schedules in shared memory and sequentially preprocess the matrix *A* to fill in the schedules for each processor to follow. This is achieved by adding the index of each element A_{ij} to the schedules of the processors responsible for calculating the partitions of the output vector containing w_i and w_j. By storing the index of the elements of matrix *A* instead of the elements itself, we are able to preserve memory as per the needs of the Earthquake simulation. The schedule determines the indices and order of the elements in *A* that each thread computes over. We choose the schedules in such a way to guarantee that each thread only writes to its unique section of the output vector *w*, which increases write locality dramatically. The schedules also remove the need for locks on *w* as by construction the threads will not contend for write access to *w*. Lastly, we modified the SMVP operation so that each thread now loops through the elements in its unique schedule instead of the entire matrix A.

We use the latedays machines to run the Spark98 SMVP programs. Each latedays node has two six-core 2.4 GHz Xeon e5-2620 v3 processors, with 15MB L3 cache and hyper-threading capability. The sparse matrices and vectors all are in shared memory on the machine. Moreover, since the kernels are written in C, it made sense for us to also write our implementation in C.

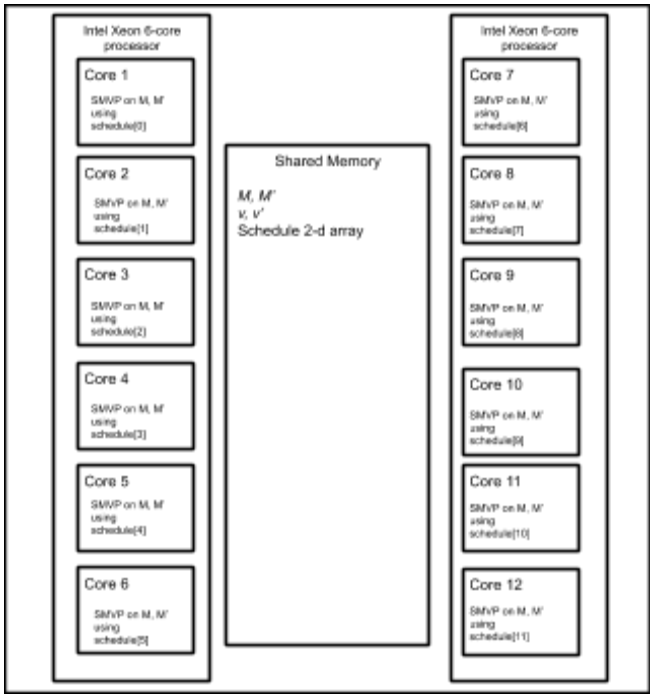


Figure 2: Diagram of Lateday System

Results

We tested the different implementations using two $n \times n$ meshes developed by the Quake project at CMU to model ground motion caused by earthquakes in the San Fernando Valley. The sf5 mesh has (n=30,169) rows and 410,923 nonzero submatrices with an average of 14 nonzero submatrices per row. The sf2 mesh has (n=378,747) rows and 5,396,875 nonzero submatrices with an average of 14 nonzero submatrices per row.

As we are interested in the speedup of the sparse matrix vector product operation $w = Av$, we measured the performance by looking at the rate of floating point operations for one SMVP operation. The number of floating point operations for one SMVP operation is calculated by multiplying the number of nonzero submatrices in the mesh by the dimensions of these submatrices. The rate is then calculated by dividing the number of millions of floating point operations by the time it takes to complete one SMVP operation. The sf2 mesh has roughly 97 million floating point operations per SMVP operation and the sf5 mesh has roughly 7 million floating point operations per SMVP operation.

The structure and location of non-zero entries of the symmetric matrix is fixed over simulation steps and separate runs on the same mesh structure (as the earth's structure remains relatively fixed), and thus the schedules can be used over different simulation steps and runs, amortizing the preprocessing time. Due to this, we ignore the timing contribution made by the preprocessing and just compare performance of the critical loops of our SMVP implementation against the sequential and LMV implementation.

We compared our implementation against the sequential implementation and the lock implementation by running each program twice on 1000 iterations of SMVP operations. We recorded the minimum time and best throughput for each trial during our tests. Our implementation and the LMV program were run on 1, 2, 4, 8, and 12 threads on the lateday clusters that support hyper-threading capability. The LMV program was run using a varying number of locks to show the fact that having less locks has lower performance due to lock contention (O'Hallaron 13). Below are figures illustrating the performance of the PSMV (our implementation), LMV, and SMV programs on the latedays cluster on SF2 and SF5.

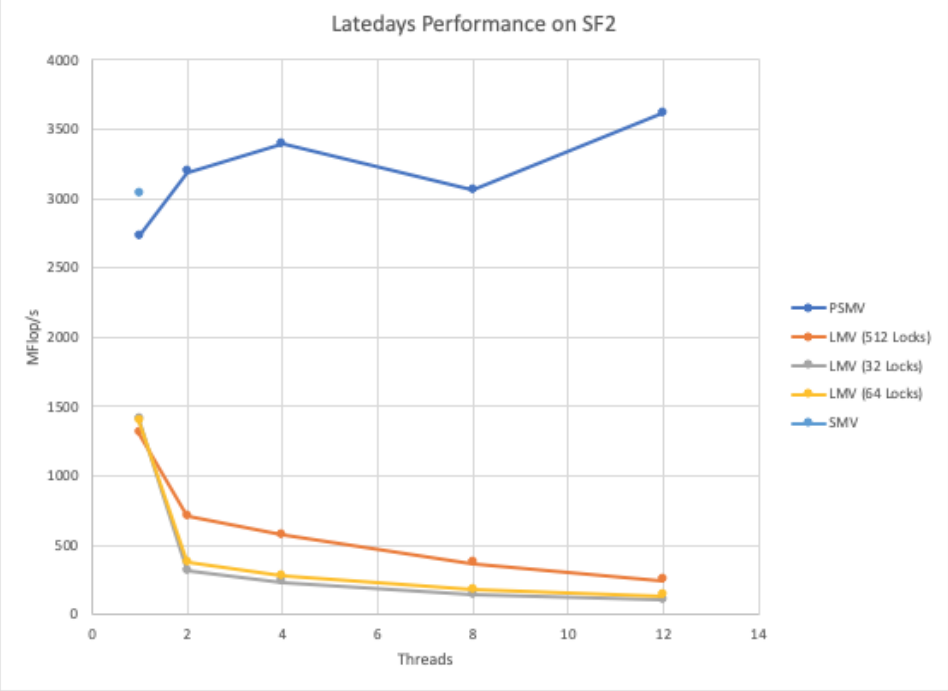


Figure 3: Performance of SMV, PSMV, and LMV using sf2 meshes on latedays clusters

Figure 4: Performance of SMV, PSMV, and LMV using sf5 meshes on latedays clusters

Both figure 3 and 4 show that the performance of LMV decreases as the number of threads are increased. Our implementation shows an opposite performance pattern of increasing throughput as the number of threads increase. Moreover, our implementation is able to perform roughly 3000 million more floating point operations a second compared to the lock implementation with 12 threads. This is due to the benefits gained of having no idle-time due to locking as well as an increase of write-locality to the output vector across the threads. Since the sf5 and sf2 tests differ by an order of magnitude in size, these results show that the trend is present over varying problem size.

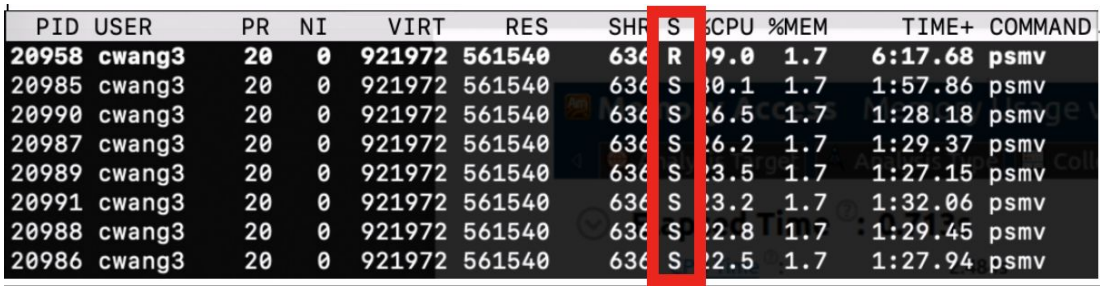
One observation we made is that running our implementation with one thread results in performance worse than the sequential implementation. This is due to our implementation having worse read locality while having the same write locality since both of the symmetric elements A_{ij} and A_{ji} are added to the schedule in each iteration of preprocessing the symmetric matrix A. When accessing the elements of the schedule formed, the one thread reads from nonzero elements that are opposite of each other in the symmetric matrix and writes irregularly to the output vector w. However, as the number of threads increase, the throughput of our

implementation surpasses that of the sequential implementation. This is due to the fact that each thread now computes only its exclusive partition of the output vector *w*, which gives higher write locality.

When looking at the performance trend of our implementation, we noticed that it does not scale well as the number of threads increased. By running our implementation on the GHC machines, we were able to gain some insights into why the program is behaving the way it is.

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
20958	cwang3	20	0	921972	561540	636	R	98.3	1.7	6:20.66	psmv
20985	cwang3	20	0	921972	561540	636	R	29.7	1.7	1:58.76	psmv
20987	cwang3	20	0	921972	561540	636	R	26.4	1.7	1:30.17	psmv
20990	cwang3	20	0	921972	561540	636	R	26.4	1.7	1:28.98	psmv
20989	cwang3	20	0	921972	561540	636	R	23.1	1.7	1:27.85	psmv
20991	cwang3	20	0	921972	561540	636	R	23.1	1.7	1:32.76	psmv
20988	cwang3	20	0	921972	561540	636	R	22.4	1.7	1:30.13	psmv
20986	cwang3	20	0	921972	561540	636	R	22.1	1.7	1:28.61	psmv

Figure 5: CPU usage when running our implementation with 8 threads on GHC



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
20958	cwang3	20	0	921972	561540	636	R	99.0	1.7	6:17.68	psmv
20985	cwang3	20	0	921972	561540	636	S	30.1	1.7	1:57.86	psmv
20990	cwang3	20	0	921972	561540	636	S	26.5	1.7	1:28.18	psmv
20987	cwang3	20	0	921972	561540	636	S	26.2	1.7	1:29.37	psmv
20989	cwang3	20	0	921972	561540	636	S	23.5	1.7	1:27.15	psmv
20991	cwang3	20	0	921972	561540	636	S	23.2	1.7	1:32.06	psmv
20988	cwang3	20	0	921972	561540	636	S	22.8	1.7	1:29.45	psmv
20986	cwang3	20	0	921972	561540	636	S	22.5	1.7	1:27.94	psmv

Figure 6: PID status when running our implementation with 8 threads on GHC

From Figure 5, we can see that the percent of CPU usage for all the worker threads are only at roughly 25 percent. Moreover, from Figure 6 it is evident that the worker threads alternate their statuses between sleep and running. We believe the lack of CPU utilization and extra idle time are likely the reason why our implementation does not scale well as the number of threads increase. However, we were not able to identify why the worker threads are consistently waiting for the master thread. We speculate that the wait time is due to the thread synchronization after each iteration.

Another observation we made is that there is a drop in the performance for our implementation when we run the program using 8 threads on the sf2 input mesh. We believe this is due to the nature of the Latedays system and that only 6 threads can be assigned to the same

processor. Therefore, with 8 threads two of the threads must be assigned to two cores on the second Xeon processor. Moreover, when submitting jobs to the Lateday clusters, threads are assigned to consecutive cores.

The results from running LMV with different number of locks show a decreasing trend as the number of threads are increased. In the Spark98 technical report, it was mentioned that this pattern occurs for small numbers of locks as the number of threads increases “due to excessive contention for locks” (O'Hallaron 12). However, in the technical report LMV shows an increasing throughput trend for LMV using 512 locks (O'Hallaron 13). This differs from LMV's performance on latedays where increasing the number of thread decreases throughput when using 512 locks.

We attempted to test the different implementations on GHC machines, but the extreme fluctuation in results made the data far too difficult to collect. Latedays displayed far more consistent test results, and so it proved to be a good choice for our project. Overall, our project shows that a shared memory lock free scheduling-based approach to partitioning the sparse-symmetric matrix has significant situational benefit over a shared memory lock-based approach.

Reference

O'Hallaron, David R. “Spark98: Sparse Matrix Kernels for Shared Memory and ...” *Spark98*, 8 Oct. 1997, www.cs.cmu.edu/~quake-papers/spark98.pdf.

O'Hallaron, David R. “Sparse Matrix Kernels for Shared Memory and Message Passing Systems.” *Spark98*, Apr. 1998, www.cs.cmu.edu/~quake/spark98.html.

Division of Work

Equal work was performed by both project members.

Our work can be found at : https://github.com/charleswang3/Spark98_SMVP