Queue Streaming Model, Part 2: Algorithms

Anup Zope, *Student Member, IEEE,* and Edward Luke, *Senior Member, IEEE,*

Abstract—In this paper we present the implementation of several common algorithms for the queue streaming model (QSM). These algorithms include structured regular computations represented by merge sort, unstructured irregular computations represented by sparse matrix dense vector multiplication, and dynamic computations represented by MapReduce. The analysis of these algorithms reveal architectural tradeoffs between memory system bottlenecks and algorithm design. The techniques described in this paper reveal a general software approach that could be used to construct more general irregular applications provided they can be transformed into a relational query form. Finally the paper demonstrates that the QSM can be used to design algorithms that enhance utilization of memory system resources by structuring concurrency and memory accesses such that system bandwidths are balanced and latency is amortized.

Index Terms—bridging model of computation, queue streaming, performance predictability and portability, QSM algorithm

1 Introduction

This paper is part two of the series of papers on queue streaming model (QSM). In part one, which we will henceforth denote by PART-1 [1], we described the QSM and presented a framework for the analysis and design of algorithms using the model. In the present paper, the merge sort, sparse matrix - dense vector multiplication (SpMV), and the MapReduce programming model [1] are expressed in the QSM and analyzed. These algorithms are backbones in a variety of application domains such as scientific computing, machine learning etc. They are chosen with the purpose of demonstrating the QSM usability for structured regular computations, irregular computations, and for computations that require dynamic load balancing. The analysis presented in this paper is useful for algorithm designers, but more importantly for hardware designers since it allows them to understand the impact of hardware parameters on an algorithm's performance. In addition to this, the cost analysis is useful for runtime schedulers that need to predetermine the computation schedule.

In Section 2, we provide QSM algorithm for merge sort and provide various optimization opportunities as advanced features from a higher level of QSM become available. In Section 3, we start with description of the sparse matrix - dense vector computation using relational algebra and its implementation on a random access machine (RAM). Then, we derive the QSM algorithm in steps, highlighting the general principles for transforming a scattered access computation to a QSM computation. In the process, we also developed the relationship between a traditional cache-based processor and a queue streaming processor (QSP) that demonstrates the equivalence between the two. In the section we also specified a number of optimization strategies for applications that consist of a chain of scattered access computations on various levels of the QSM. In Section 4 we demonstrate the application of the QSM to the MapReduce programming model.

- *Anup Zope is with the Center for Advanced Vehicular Systems, Mississippi State University, Starkville, USA*
- *Edward Luke is with the Department of Computer Science and Engineering, Mississippi State University, Starkville, USA*

Manuscript received (Date); revised (Date).

1. A. Zope and E. Luke, "Queue Streaming Model, Part 1: Theory," Submitted for review to TPDS

2 Merge Sort

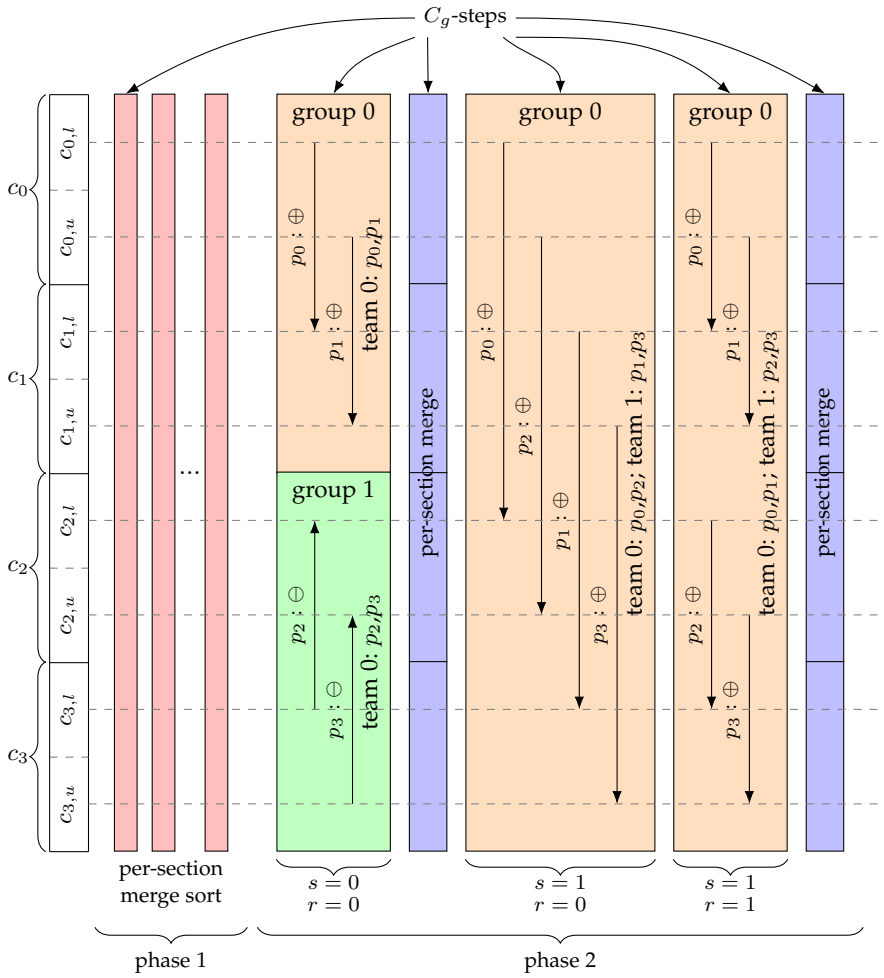
Let's assume that the cores in a QSP are labeled as $p_i; i = 0, ..., P-1$. Also assume that the number of cores, P, and the length of the input array, L, are powers of two and $L \gg P$. Let's also assume that the array elements are of type τ and the size of the type is $S(\tau)$.

If we imagine a hypothetical algorithm that sorts the array using a $\log_2 L$ pass single core algorithm that achieves full system bandwidth for each stage, the execution time of the sort is given by Equation 1. In reality, a single core will not be able to achieve the bandwidth of B_g so concurrent streaming will be needed to achieve this. In addition, potentially greater benefits could be accrued due to potential data reuse optimizations. As a result, this naive time is a good baseline to estimate the effectiveness of the sorting algorithm presented here.

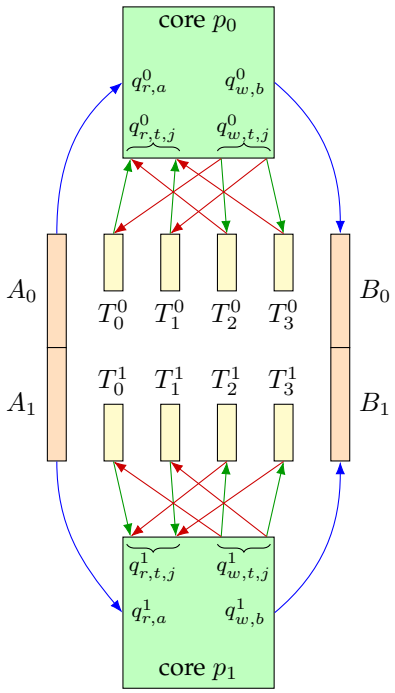
$$t_{\text{naive}} = \left(S(\tau) \times 2L \times \frac{1}{B_g} + o_g \right) \times \log_2 L \qquad (1)$$

Now we present the algorithm for the QSM in detail, discuss various optimization opportunities, and show how they improve the execution time of the hypothetical algorithm discussed in the last paragraph. The algorithm sorts the input array in following two phases. Figure 1a depicts these phases as a series of C_g-steps on a QSP with four cores.

1) *Phase 1*. Each core sorts a section of the input array using merge sort such that the output array contains P sorted sequences.
2) *Phase 2*. The P sorted sequences are merged to produce the final sorted array.



(a) Schematic of the merge sort algorithm for a QSP. In this figure, phase 2 is carried out using the bitonic sorting algorithm.



(b) Schematic of the queue streaming kernel for phase 1 of the merge sort algorithm on a level-0 QSP with two cores.

Fig. 1. Schematic of the merge sort algorithm for a QSP.

2.1 Phase 1

Let A and B be two arrays of type τ and length L allocated in the global memory. The values in A are unsorted. The objective of the phase 1 is to ensure that after its completion, section c_i of B (denoted as B_i) contains values in section c_i of A (denoted as A_i) in sorted order, where c_i is a range of indices defined as $c_i = \left[i \cdot \frac{L}{P}, (i+1) \cdot \frac{L}{P}\right)$. There are multiple ways to execute the phase 1 depending on the level of the QSM supported on a target processor. Let's see the variants one by one.

2.1.1 Version 1: For the Level-0 QSM

This version performs n-way merge instead of the 2-way merge of the naive algorithm, except for the first pass. The reason for the 2-way merge of the first pass is given later in this section. Here, n is determined ahead of the execution from the number of queues available on each core. We also assume that the arithmetic intensity of the n-way comparison is less than the threshold of the arithmetic intensity for compute-boundedness on a given core. Therefore, each merge pass is bandwidth-bounded.

This algorithm requires $s = 0, ..., \lceil \log_n \left(\frac{L}{2P}\right) \rceil$ merge passes. Each pass is executed using a C_g-step. It also requires $2nP$ temporary arrays to store intermediate sorted sequences. They are denoted as $T^i_j; i = 0, ..., P-1; j = 0, ..., 2n-1$. Each array is of length $2n^{h-1}$, where h is the smallest integer such that $\frac{L}{P} \leq n^h$, that is, $h = \lceil \log_n \left(\frac{L}{P}\right) \rceil$.

For $s = 0$, core p_i configures and activates a C_g-queue, denoted as $q^i_{r,A}$ to read from A_i, and n C_g-queues, denoted as $q^i_{w,T,j}; j = 0, ..., n-1$, to write to the temporary arrays T^i_{j+n} in the initialization phase. In the computation phase, it performs a number of iterations. In each iteration, the core advances $q^i_{r,a}$ by two values, sorts them and pushes the sorted block of length 2 to the queue $q^i_{w,T,k\%n}$, where k is the iteration index and $\%$ is the modulo operator. Effectively, the $\%$ operation pushes the sorted blocks to the temporary write arrays in round robin order. Then, the core deactivates the queues. This initiates the core synchronization. In the first pass, we perform 2-way merge instead of n-way merge since the 2-way comparison does not require any storage. However, if a particular QSP has sufficient registers, it can use $n > 2$ for the first pass, where n depends on the number

of registers.

The next pass performs n-way merge to merge the length 2 blocks into blocks of length $2n$. In general, for $s = 1$ to $\lceil \log_n \left(\frac{L}{2P}\right) \rceil - 1$, the core configures read queues $q^i_{r,T,j}; j = 0, ..., n-1$ to read from the temporary arrays $T^i_{j+n\cdot(s\%2)}$ and write queues $q^i_{w,T,j}; j = 0, ..., n-1$ to write to the temporary arrays $T^i_{j+n\cdot((s+1)\%n)}$ in the initialization phase. This arrangement causes the temporary arrays to alternate their role between inputs and outputs of the C_g-steps as s advances. The schematic of the data flow for the phase 1 on a level-0 QSP is shown in Figure 1b. The arrows indicate the direction of the streaming data flow. The blue and green arrows indicate the flow paths when s is an even number. The blue and red arrows indicate the flow paths when s is an odd number. In each iteration of the computation phase, the core merges n blocks of length $2n^{s-1}$, one from each $q^i_{r,T,j}$, and streams out the resulting block of length $2n^s$ to $q^i_{w,T,k\%n}$, where k is the iteration index.

In the last pass, that is, $s = \lceil \log_n \left(\frac{L}{2P}\right) \rceil$, the core merges the n blocks of (maximum) length n^{h-1} from $q^i_{r,T,j}$ that are configured to read from $T^i_{j+n\cdot(s\%2)}$ and streams out the resulting block of length $\frac{L}{P}$ to $q^i_{w,B}$ that is configured to write to B_i. This completes the phase 1 of the sorting algorithm.

Each merge pass reads $S(\tau) \cdot L$ words of data and writes $S(\tau) \cdot L$ words of data to the global memory. Therefore, the execution time of the phase 1 is given Equation 2. The overhead o_g due to activation and deactivation of the queues is amortized if L is large.

$$t_1 = \left(S(\tau) \times 2L \times \frac{1}{B_g} + o_g\right) \times \left(\left\lceil \log_n \left(\frac{L}{2P}\right)\right\rceil + 1\right) \quad (2)$$

2.1.1.1 Architectural Implications: Though the output of this version is yet not fully sorted, comparison of Equation 1 and 2 shows that it obtains speed up by performing $n(> 2)$-way merge instead of the 2-way merge of the hypothetical algorithm. Larger the value of n, smaller is the number of passes required to form the $\frac{L}{P}$ sorted blocks. However, n has practical limitations. First, it is limited by the number of queues available on each core. The number of queues that can be supported with a given M0 buffer capacity is limited by the block sizes and bandwidths of the higher level of memories, and the corresponding queue scheduler service time (see PART-1, Appendix A). Also, the M0 capacity cannot increase arbitrarily since it also increases the latency of the look-back access. Second, n is also limited by the core's arithmetic throughput. Lower the throughput, closer is the n-way comparison to the limit of compute-boundedness. A compute-bounded n-way comparison for the merge operation performs poorly compared to a bandwidth-bounded n-way comparison as n grows since its growth function becomes linear in n. Therefore, it is necessary to choose n that makes the n-way comparison bandwidth-bounded. Finally, the term $\lceil \log_n \left(\frac{L}{2P}\right) \rceil$ indicates that as n increases, the rate of improvement in the execution time with respect to n reduces with a reciprocal of n. Therefore, a moderate number of queues are enough to ensure good speed up compared to the hypothetical version. This also justifies the small capacity of the M0 buffer as well as the assumption of bandwidth-boundedness of the merge passes.

Also, as the bandwidth to the global memory improves, the execution time improves since the passes over the global memory incur lower cost. This is evident from the term $\frac{1}{B_g}$ in Equation 2.

The hypothetical version assumes full system bandwidth at each stage when using only one core. However, in practice $B_c < B_g$. This algorithm uses concurrency ($P > 1$) to ensure full system bandwidth for each merge pass.

In general, as the block size and latency lowers, and the bandwidth grows of a the global memory the overhead o_g reduces (see PART-1, Appendix B). This improves the execution time of the sort. However, for large L, the overhead is amortized. Hence, the improvement may not be significant.

2.1.2 Version 2: For the Level-1 QSM

This version uses the local memory to form larger than n, let's say u, length blocks in the first pass. In practice u is limited by the size of the local memory, the memory footprint required for sorting a block of length u, and the maximum burst size allowed by a queue on the core which depends on the M0 buffer space available to the queue for hiding the latency. Since all these parameters are known in advance, u can be predetermined. The execution time estimation has to take into account that the first pass may become compute-bounded. Once the value of u is determined, each iteration of the first pass ($s = 0$) advances the queue $q^i_{r,A}$ by u values, stores them in the local memory, and streams out the sorted blocks to the queue $q^i_{w,T,k\%n}$, where k is the iteration index. The subsequent passes of the n-way merge work similar to the version 1. In this case, there are $\lceil \log_n \lceil \frac{L}{u\cdot P} \rceil \rceil$ such passes. If $t_{l,u}$ is the time required to bring the data in the local memory from the queue buffer, sort it and, write it to the queue buffer, the execution time of the phase 1 is given by Equation 3.

$$t_1 = \left\lceil \frac{L}{u \cdot P} \right\rceil \times t_{l,u} + o_g + \left(S(\tau) \times 2L \times \frac{1}{B_g} + o_g\right) \times \left\lceil \log_n \left\lceil \frac{L}{u \cdot P} \right\rceil \right\rceil \quad (3)$$

2.1.2.1 Architectural Implications: Equation 3 shows that as u increases, $t_{l,u}$ increases and $\lceil \frac{L}{u\cdot P} \rceil$ decreases. This implies that the time required for the first pass will increase since the growth rate of $t_{l,u}$ is $u \cdot \log u$ and the term $\lceil \frac{L}{u\cdot P} \rceil$ reduces linearly in steps. However, a larger u reduces the number of subsequent merge passes compared to the hypothetical algorithm (Equation 1) and the version 1 (Equation 2). In practice, u has limitations. First, it is limited by the size of the local memory, and the maximum allowed burst size for a queue (see explanation of $D_{i,\max}$ in PART-1, Appendix A), which is limited by the M0 buffer capacity. Since both the memories are closer to the core, increasing capacity of one memory reduces the space available for the other. This highlights that it may not be useful to increase the local memory size while sacrificing the M0 buffer size since it will limit the burst size ($D_{i,\max}$), which in turn will limit u. Second, the term $\lceil \log_n \lceil \frac{L}{u\cdot P} \rceil \rceil$ indicates that the returns from increasing u diminish

rapidly as the reduction in the number of merge passes slows down with increase in u. Therefore, a moderate value of u is practically beneficial. This also justifies the small capacity of the local memory.

2.1.3 Version 3: For the Level-3 QSM

Similar to the level-1 model analysis, we assume that a core can sort a block of length u stored in the local memory without requiring access to the M1 buffer or the global memory. In addition to this, the level-3 model allows us to form sorted blocks larger than u if we use the M1 buffer to store intermediate merge sequences.

The sorting is carried out in two stages - stage 1 and stage 2. The stage 1 requires one C_g-step and multiple nested C_1-steps. The stage 2 requires multiple C_g-steps. Let's assume that each C_1-step of the stage 1 performs m-way merge and each C_g-step of the stage 2 performs n-way merge.

The algorithm requires $2m$ temporary arrays per core, each of length m^{f-1}, allocated in the M1 buffer and $2n$ temporary arrays per core, each of length n^{h-1}, allocated in the global memory, in addition to the input and output arrays A and B. Here, f is the largest integer such that $m^f \leq \left\lfloor \frac{\Delta_1 - \Delta_{1,\text{latency}}}{2u \cdot P \cdot S(\tau)} \right\rfloor$, and h is the smallest integer such that $\frac{L}{P} \leq n^h$. In this case, the stage 1 forms sorted blocks of length $v = m^f$. Each block requires f merge passes, each represented by a C_1-step. Out of these f passes, the first and last pass access the global memory and the others access only the M1 buffer. Therefore, the effective bandwidth of the first and last pass is B_g, and that of the remaining passes is B_1 words per second. There are $\left\lceil \frac{L}{m^f \cdot P} \right\rceil$ blocks of length m^f in the input array section of length $\frac{L}{P}$. Therefore, there are in total $\left\lceil \frac{L}{m^f \cdot P} \right\rceil \times f$ C_1-steps in the stage 1. The stage 2 accesses only the data in the global memory. It requires $\left\lceil \log_n \left\lceil \frac{L}{v \cdot P} \right\rceil \right\rceil$ merge passes to form the final sorted sequence of length $\frac{L}{P}$ in the output array. Therefore, the execution time of the phase 1 on a level-3 QSM is given by Equation 4.

$$
\begin{aligned}
t_1 = & \left\lceil \frac{L}{u \cdot P} \right\rceil \times t_{l,u} + S(\tau) \times L \times \frac{1}{B_g} + o_g \\
& \qquad \text{... } C_g\text{-step of stage 1} \\
& + S(\tau) \times 2L \times (f-2) \times \frac{1}{B_1} + o_1 \times f \times \left\lceil \frac{L}{m^f \cdot P} \right\rceil \\
& \qquad \text{... } C_1\text{-steps of stage 1} \\
& + \left(S(\tau) \times 2L \times \frac{1}{B_g} + o_g \right) \times \left\lceil \log_n \left\lceil \frac{L}{m^f \cdot P} \right\rceil \right\rceil \\
& \qquad \text{... } C_g\text{-steps of stage 2}
\end{aligned} \tag{4}
$$

2.1.3.1 Architectural Implications: Similar to the version 2, this version improves the execution time over the version 1 and the hypothetical version by using the local memory to reduce the number of passes over the global memory. In addition to this, it improves the performance over the version 2 by forming larger blocks in the M1 buffer without requiring access to the global memory. Even though there are diminishing returns from increasing the block size (m^f) formed in the M1 buffer in terms of the reduction in the number of passes over the global memory data, the performance gain of this version is primarily due to the higher bandwidth of the M1 buffer. The higher the gap between the bandwidth of the M1 buffer and the global memory, the higher is the potential speed up of this version compared to the earlier versions. This indicates that processor designers need to provide much higher bandwidth for the M1 buffer compared to the global memory.

2.2 Phase 2

The phase 2 merges the P sorted blocks of length $\frac{L}{P}$ that the phase 1 produces. There are multiple ways to perform the merge.

2.2.1 Version 1: Using Merge Sort

This version of the phase 2 merges the P sorted sections from the phase 1 using exponentially reduced number of cores with each merge pass. This reduces the bandwidth when the number of cores in a pass falls below $p_{\text{crit}} = \left\lceil \frac{B_g}{B_c} \right\rceil$. However, the number of passes is only $\log_n P$ if each pass uses n-way merge. For $n = 2$, the execution time for this variant is given by Equation 5.

$$
t_2 = \sum_{i=1}^{\log_2 P} \left(\frac{S(\tau) \times 2L}{\min\left(B_g, \frac{P \cdot B_c}{2^i}\right)} + o_g \right) \tag{5}
$$

2.2.2 Version 2: Using Bitonic Merge

The lower bandwidth in the last passes of the version 1 slows down the computation. This version uses bitonic merge network [2] to ensure full system bandwidth for each merge pass. It also expects that the output from the phase 1 contains $\frac{P}{2}$ bitonic sequences such that the elements in section c_i of the output are in ascending order if i is even, in descending order otherwise. As mentioned previously, $c_i = \left[i \cdot \frac{L}{P}, (i+1) \cdot \frac{L}{P}\right)$ is a section of the array index space. In the bitonic network, all the P cores work concurrently on disjoint working sets to produce bitonic sequences of larger size. This is depicted in Figure 1a. This algorithm performs $\log_2 P$ stages: $s = 0, ..., \log_2 P - 1$. A stage s performs $s+1$ passes of the bitonic merge followed by a merge pass. The bitonic merge passes leave a bitonic sequence of length $\frac{L}{P}$ in each section c_i of their output array. This is due to the fact that each step of the bitonic merge on a bitonic sequence leaves sub-sequences of half length that are also bitonic. In the merge pass, each core p_i merges the sorted sections of the bitonic sequence in section c_i, effectively producing a larger bitonic sequence (compared to the last stage) in the output array.

Unlike the traditional bitonic merge network that operates in-place, this algorithm requires that the input and output array of each step are distinct memory locations due to the no-overlap requirement of the QSM. The merge pass requires two or three read queues that read in either forward or backward direction from the global memory and one write queue that writes to the global memory. The bitonic merge passes require two read streams and two write streams. Each core operates on disjoint portion of the arrays in each step. In addition, they also require synchronization after each step of the bitonic network as well as after each local merge step. Therefore, the merge step as well as each step of the bitonic network can be executed as a queue streaming kernel (see Figure 1a).

The phase 2 performs $\frac{\log_2 P(1+\log_2 P)}{2}$ bitonic merge passes and $\log_2 P$ merge passes. Each pass reads $S(\tau) \times L$ words from and writes $S(\tau) \times L$ words to the global memory. Therefore, the execution time of the phase 2 is given by Equation 6.

$$t_2 = \sum_{i=1}^{\log_2 P} \left[(i+1) \times \left(\frac{S(\tau) \times 2L}{B_g} + o_g \right) \right] \quad (6)$$

2.2.3 Version 3: Hybrid of Merge and Bitonic Merge

Though the number of passes in the version 1 are $\log_2 P$, it achieves lower bandwidth when the number of cores in a pass falls below p_{crit}. The version 2 ensures use of the concurrency to achieve full bandwidth for each pass. However, the number of passes grow as $\log_2^2 P$. The version 3 uses either version based on which will perform the merge step faster. In the early stages of merging, when the natural concurrency is high, version 1 will be superior, while under the right conditions of B_c and B_g, the higher bandwidth of the bitonic merge may allow for faster merging steps. The cost of the hybrid version is given by Equation 7.

$$t_2 = \sum_{i=1}^{\log_2 P} \min \left[\frac{S(\tau) \times 2L}{\min\left(B_g, \frac{P \cdot B_c}{2^i}\right)} + o_g, \right. \\ \left. (i+1) \times \left(\frac{S(\tau) \times 2L}{B_g} + o_g \right) \right] \quad (7)$$

2.2.4 Architectural Implications

There are several choices of algorithms for the phase 2. However, the optimal algorithm can be chosen quickly ahead of the execution. The factors that affect the choice are the architectural parameters such as P, B_c, B_g, and the number of queues. As the difference between B_c and B_g reduces, the version 1 becomes more efficient since most of the merge passes achieve full system bandwidth. However, as the gap grows, the version 1 performs poorly. In this case, the version 2 may be more efficient. However, to ensure minimum number of passes of the version 2, the phase 1 needs to use the least number of cores that attain full system bandwidth (i.e. $p = 2^{\lceil \log_2 p_{\text{crit}} \rceil}$). Finally, a hybridized version provides a method that an algorithm can adapt to the architectural parameters to achieve an optimal execution schedule. The predictability of the model performance becomes a significant factor in developing such adaptive algorithms.

A critical point to note in this analysis is the role of the core concurrency in saturating the memory bandwidth. Closer the per core bandwidth B_c to the system bandwidth B_g, lesser is the number of cores required to saturate the link between the queue scheduler and the global memory, resulting in lower execution time of the phase 2. However, a higher value of B_c requires that each core has a larger M0 buffer so that the queue scheduler can sustain a larger number of access requests from a single core. Therefore, the performance of the sorting algorithm is affected by the M0 buffer capacity. Since the M0 buffer, the local memory and the core are in close proximity to one another in a small space, processor designers need to obtain an optimal trade-off in the space usage by these three systems.

3 Sparse Matrix - Dense Vector Multiplication (SpMV)

Consider a sparse matrix - dense vector multiplication $X^1 = AX^0$, where A is an $n \times n$ sparse matrix, X^0 and X^1 are $n \times 1$ dense vectors. If the matrix and vectors are expressed as sets of tuples: $X^1 = \{(i, x_i^1)\}$, $X^0 = \{(j, x_j^0)\}$, and $A = \{(i, j, a_{i,j}) | a_{i,j} \neq 0\}$, the product can be expressed as shown in Equation 8 and depicted as shown in Figure 2. The idea of expressing SpMV computation as a relational query is not new. See, for example, [3].

$$X^1 = \boldsymbol{\gamma}_{[i, \sum p_{i,j}^0]} \left(\boldsymbol{\pi}_{[i,j,p_{i,j}^0 \leftarrow a_{i,j} \cdot x_j^0]} \left(\bowtie_{[j]} (A, X^0) \right) \right) \quad (8)$$

Here,

1) $\boldsymbol{\gamma}_{[i, \sum p_{i,j}^0]}(P^0)$ is the group-by operator that groups the tuples in the set P^0 by the attribute i and reduces the attribute $p_{i,j}^0$ in each group using the summation operation to form the set $X^1 = \{(i, x_i^1)\}$.
2) $\boldsymbol{\pi}_{[i,j,p_{i,j}^0 \leftarrow a_{i,j} \cdot x_j^0]}(Q^0)$ is the projection operator that forms the product $p_{i,j}^0 = a_{i,j} \cdot x_j^0$ for each tuple in Q^0 and produces the set $P^0 = \{(i, j, p_{i,j}^0)\}$.
3) $\bowtie_{[j]} (A, X^0)$ is the equijoin operator that joins the tuples in the sets A and X^0 on j, and produces the set $Q^0 = \{(i, j, a_{i,j}, x_j^0)\}$.

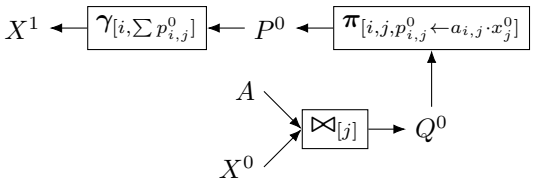


Fig. 2. Graphical representation of Equation 8.

The set notation used in the above description is sufficient to completely specify the computation. However, to perform the computation on a machine, it is necessary to also know the order in which the tuples are stored. Arrays enforce the order. Henceforth, we will use $X^1 = [(i, x_i^1)]$, $X^0 = [(j, x_j^0)]$, and $A = [(i, j, a_{i,j})]$ to represent arrays instead of sets. Note that A stores only the tuples for which $a_{i,j} \neq 0$. We will assume that X^1 stores a tuple (i, x_i^1) at index i and X^0 stores a tuple (j, x_j^0) at index j. The tuples of A are stored either in row-major or column-major order. The row-major ordered matrix is given by $A^r = \boldsymbol{\tau}_{[i,j]}(A)$, where $\boldsymbol{\tau}_{[i,j]}$ is the sorting operator that uses i as the primary sort key and j as the secondary sort key. Similarly, the column-major ordered matrix is given by $A^c = \boldsymbol{\tau}_{[j,i]}(A)$. We say that X^0 is j-ordered since it stores the tuples (j, x_j^0) in ascending order of j. Following the same terminology, X^1 is i-ordered, A^r is i-ordered, and A^c is j-ordered. Especially, A^r is (i, j)-ordered and A^c is (j, i)-ordered.

In this notation, i represents the row index space and j represents the column index space of the matrix. The non-zero structure of the matrix is given by the maps $M : i \rightarrow j = \boldsymbol{\pi}_{[i,j]}(A^r) = [(i, j)]$ which is i-ordered (specifically, (i, j)-ordered), and $M^{-1} : j \rightarrow i = \boldsymbol{\pi}_{[j,i]}(A^c) = [(i, j)]$ which is j-ordered (specifically, (j, i)-ordered). These maps specify the association between the indices in the two index

spaces. Also, note that $M^{-1} = \tau_{[j,i]}(M)$. Operators in Equation 8 that have their operands ordered by different index spaces require these maps to form the association between the elements of their operands. Also, the γ operator performs reduction using the $\sum$ operation. Hence, Equation 8 is an example of the *mapped reduction* operation.

3.1 Version 1: Using Scattered Access

Random access machine (RAM) allows independent access to any indexed location in an array. There are two ways to evaluate Equation 8 on the machine.

3.1.1 Case 1: Using A^r:

If we use A^r for the computation, A^r is i-ordered and X^0 is j-ordered in the expression $\bowtie_{[j]} (A^r, X^0)$. On a scattered access machine, the $\bowtie$ operator scans through A^r, and for each $(i, j, a_{i,j}) \in A^r$ it directly accesses x^0_j from X^0 using the index j. This operation generates streaming access to A^r and scattered access to X^0. The resulting tuples in Q^0 are i-ordered. It is scanned by the π operator to produce P^0. Since P^0 is already i-ordered, the γ operator requires only a scan over P^0 to produce X^1 in i-order.

3.1.2 Case 2: Using A^c

If we use A^c for the computation, both the operands of the $\bowtie$ operator are j-ordered. Hence, it requires only a scan over both the arrays. The subsequent intermediate results, Q^0 and P^0, are j-ordered. The γ operator scans over P^0 in j-order and adds the value $p^0_{i,j}$ to x^1_i in X^1 located at index i. This generates scattered access to X^1.

3.2 Version 2: Using Streaming Access

When a machine is allowed to perform only streaming access to the global memory, the $\bowtie$, π, and γ operators can only scan the tuples in each operands in the order in which they are stored. Then, Equation 8 is evaluated in one of the following three ways.

3.2.1 Case 1: Using A^r:

There are two ways to evaluate Equation 8 when A^r is available.

3.2.1.1 Case 1A: Transformation of A^r: It is shown in Figure 3 as a composition of a sorting and a mapped reduction phase. In this case, the tuples of A^r are sorted by j before passing them to the $\bowtie$ operator so that it requires only a scan over its operands. This yields R^0 in j-order. Then, the π operator scans over R^0 to produce Q^0 in j-order. Finally, the computation sorts Q^0 by i and then passes it to γ, which then performs a scan over P^0 to produce X^1 in the desired i-order. Note that the sorting operator (τ) requires multiple passes on a streaming machine as shown in Section 2.

3.2.1.2 Case 1B: Transformation of X^0: It is shown in Figure 4 as a composition of four phases. Instead of sorting A^r by j for the $\bowtie$ operator, we can also duplicate and sort X^0 to bring the duplicated tuples in (i, j)-order. This is achieved by the operation: $\tau_{[i,j]} \left(\bowtie_{[j]} \left(\tau_{[j,i]}(\pi_{[i,j]}(A^r)), X^0\right)\right) (= R^0$, let's say). Here, $\tau_{[j,i]} \left(\pi_{[i,j]}(A^r)\right)$ forms the (j, i)-ordered map $M^{-1} : j \rightarrow i$,

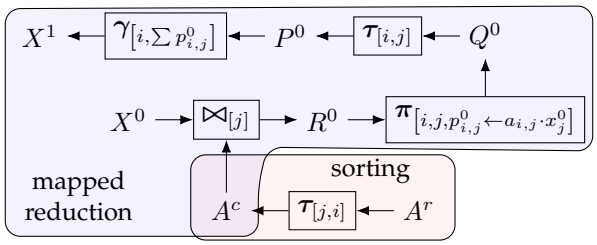


Fig. 3. Version 2-Case 1A of the SpMV computation that uses only streaming access.

which when joined with X^0 forms an array that contains duplicates of the values in X^0_j. Note that the $\bowtie_{[j]} (M^{-1}, X^0)$ operation requires only a scan over M^{-1} and X^0. Sorting the resulting tuples using $\tau_{[i,j]}$ brings them back in (i, j)-order. Since A^r is also (i, j)-ordered, the operation $\bowtie_{[j]} (A^r, R^0)$ requires only a scan over A^r and R^0 to produce Q^0 in i-order. After this, the π and γ operators scan their operands to produce X^1 in i-order.

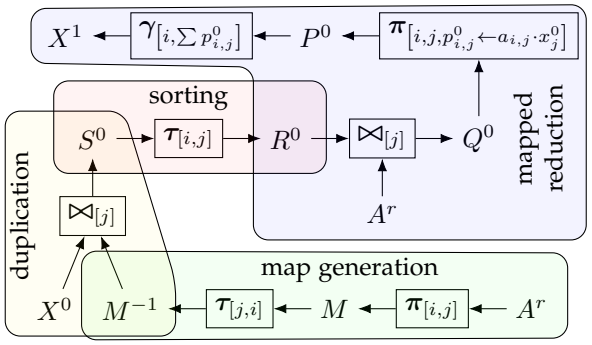


Fig. 4. Version 2-Case 1B of the SpMV computation that uses only streaming access.

3.2.2 Case 2: Using A^c:

The operations in this case are similar to the operations for the Case 1A except the sorting phase since $A^c = \tau_{j,i}(A^r)$ is already j-ordered.

3.3 Version 3: Using QSM

All the variants of the Version 2 of the SpMV computation can be executed on QSM since the queues provide sequential access to the data in the global memory. In this section we demonstrate optimization opportunities that QSM provides to the computation.

First, we take another look at the Version 1-Case 1 of the SpMV computation. On a scattered access machine, the operation $\bowtie_{[j]} (A^r, X^0)$ performs streaming access to A^r and scattered scattered access to X^0. On QSM, let's assume that it scans over both the operands even though they are ordered by different index spaces. Then, let's assume that for the tuple $(i, j, a_{i,j})$ at current scan position in A^r, the QSM computation ensures that the current scan position in X^0 is $j_f = i + \min(d_1, n - i)$. Here, d_1 is called as *look-ahead distance*. It is the maximum distance by which the scan position in X^0 is ahead of $j = i$ in the j index space of X^0. Then, the look-back of the queue that is configured to read

from X^0 contains tuples in the index range $w(i) = [j_r, j_f)$, where $j_r = \max(0, j_f - d_2)$. Here, d_2 is the *look-back distance* of the queue. $w(i)$ is called as *look-back window*. Note that, both j_f and j_r are monotonically increasing functions of i. Therefore, $w(i)$ slides ahead in X^0 as the computation scans over A^r in i-order. This maintains streaming access to X^0. The look-back window is depicted in Figure 5. Both the distances d_1 and d_2 are in terms of the number of values of the type of tuples in X^0, and they remain constant during the computation. Under this setting, the only X^0 tuples that are not in the look-back need duplication and sorting so that they can be supplied to the join operator via a separate queue.

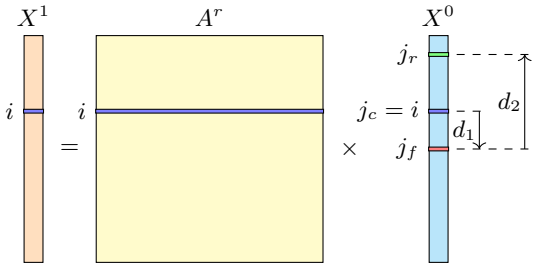


Fig. 5. Demonstration of the look-back window $w = [j_r, j_f)$ of the X^0 stream defined in terms of the current index i in A^r and X^1 streams. Here, d_1 is the look-ahead distance and d_2 is the look-back distance.

Similar to the Version 2-Case 1B, the QSM version duplicates and sorts values from X^0. However, instead of the map M^{-1}, it uses another map $D^{-1} \subset M^{-1}$ that reduces the amount of duplication due to the look-back access in X^0. To determine the map, we first augment A^r to contain tuples $(k, i, j, a_{i,j})$, where k is the index at which the tuple is stored in the array. Then, the duplication map is given by $D^{-1} = \boldsymbol{\tau}_{[j,k]}(\boldsymbol{\pi}_{[k,j]}(\boldsymbol{\sigma}_{[j \notin [j_r, j_f)]}(A^r)))$. See Figure 6. Here, $\boldsymbol{\sigma}_{[j \notin [j_r, j_f)]}$ is a selection operator that selects the tuples that fit the criterion $j \notin [j_r, j_f)$. Now, the duplication and sorting phases requires only streaming access over their operands. The resulting array R^0 is k-ordered. Next, the $\bowtie_{[j]}\ (A^r, X^0, R^0)$ operation generates Q^0 by streaming through A^r, X^0, and R^0. For each tuple $(k, i, j, a_{i,j}) \in A^r$, if $j \in [j_r, j_f)$, the value x_j^0 is read from the look-back window of X^0, otherwise it is read from R^0 to generate the tuples $(k, i, j, a_{i,j}, x_j^0)$ in Q^0. This operation maintains streaming access of the QSM to all the four operands. The tuples in Q^0 are k-ordered. Naturally, they are also i-ordered. Next operations are performed in the same manner as the Version 2-Case 1B.

3.3.1 Look-back Access to R^0

If the operation $\bowtie_{[j]}\ (A^r, X^0, R^0)$ uses look-back for the array R^0, further reduction in the cost of the duplication phase, and consequently, the sorting phase can be obtained. For this, the map D^{-1} needs to be processed to remove redundant duplication that accounts for the look-back window in the queued access to R^0, before performing the duplication operation $\bowtie_{[j]}\ (D^{-1}, X^0)$.

3.4 QSM vs cache-based processor:

The look-back window in QSM serves the same purpose as the cache in a cache-based processor. In the scattered access version, the cache system transparently serves the accesses to the global memory by automatically duplicating the data on a cache-miss. On a QSP, programs need to explicitly duplicate the data. It is essentially a process that gathers the look-back-missed references at one place and then sorts them according to the order in which they are referenced so that the sequence of computations generates streaming access to the duplicated data. In contrast to the scattered access version, which pays the latency cost for every cache-missed reference, the QSM version pays the latency cost only once for all the look-back-missed references. Therefore, the QSM version amortizes the latency. As a result, the algorithm's performance on a QSP with a certain look-back distance will likely be better than its performance on a cache-based processor with equivalent cache capacity. Another way to look at the relationship between the QSM and the cache-based processor is that the cache operates in temporal dimension by keeping track of time history of references while the queues and the look-back buffer operate in spatial dimension by keeping track of space history of the references. Consequently, the process of transforming a scattered access implementation to a QSM implementation described in this section is a transformation that brings the dispersed temporal references in a spatial order that amortizes their access latency.

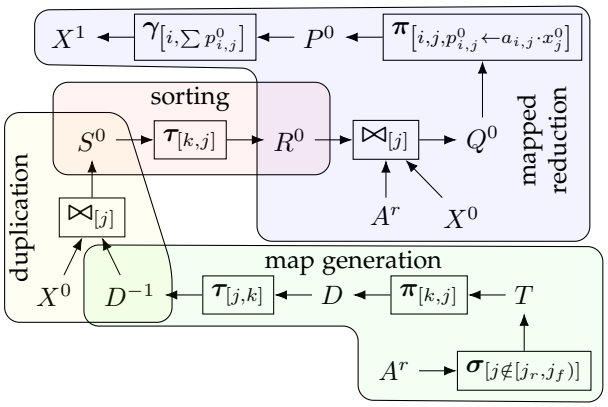


Fig. 6. Version 3 of the SpMV computation on a QSP.

3.5 Specific Adaptations

3.5.1 Multi-core execution:

Since the computation of each x_i^1 in X^1 is independent, multi-core execution of all the three versions of the SpMV computation is trivial. It requires partitioning of the i index space, which implies partitioning of X^1, A^r, X^0, and all the other intermediate arrays. The QSM implementation needs to take into account the partition boundaries of X^0 to determine the extents of the look-back window.

3.5.2 Storage optimizations:

There are several ways to reduce the space required for the storage of the arrays such as X^1, A^r etc. For example, there is no need to store the index i in the tuples (i, x_i^1) of X^1 since the index is implicit while scanning the array. Similarly, the repetition of index i in the tuples $(i, j, a_{i,j})$ of A^r can be eliminated by using an offset array $A^{r,o} = [(k_i)]$

and a value array $A^{r,v} = [(j, a_{i,j})|a_{i,j} \neq 0]$. This is usually referred as compressed-row-storage (CRS) format. Similarly, the array A^c can be split into $A^{c,o}$ and $A^{c,v}$ that represent the compressed-column-storage (CCS) format. In case of QSM, the index for accessing data from the look-back of a queue can be safely represented using a 16-bit or 8-bit integer depending on the range of the look-back indices. Further reduction in the storage space can be obtained by fusing operators as described in the next section.

3.5.3 Operator fusion:

Even though the dependency between the operators imply sequential execution, the implementations do not need to do so. For example, all operators in the operation $X^1 = \boldsymbol{\gamma}_{[i,\sum p^0_{i,j}]}\left(\boldsymbol{\pi}_{[i,j,p^0_{i,j} \leftarrow a_{i,j} \cdot x^0_j]}\left(\bowtie_{[j]}(A^r, R^0)\right)\right)$ shown in Figure 4 can be fused in a single operator as demonstrated in Figure 7. Operator fusion also saves storage space by eliminating intermediate arrays such as P^0 and Q^0 in this example.

for all $i \in [0, n)$ **do**
  $k_1 = A^{r,o}[i]$
  $k_2 = A^{r,o}[i+1]$
  $t = 0$
  for all $k \in [k_1, k_2)$ **do**
    $e = A^{r,v}[k].a_{i,j}$
    $v = R^0[k].x^0_j$
    $t = t + e * v$
  end for
  $X^1[i] = t$
end for

Fig. 7. Demonstration of operator fusion in the computation $X^1 = \boldsymbol{\gamma}_{[i,\sum p^0_{i,j}]}\left(\boldsymbol{\pi}_{[i,j,p^0_{i,j} \leftarrow a_{i,j} \cdot x^0_j]}\left(\bowtie_{[j]}(A^r, R^0)\right)\right)$ of the SpMV computation in Figure 4.

Operator fusion can also be performed for the mapped reduction shown in Figure 3. Here, the $\bowtie_{[j]}(A^c, X^0)$ operation duplicates x^0_j and produces tuples $(i, j, a_{i,j}, x^0_j)$ in R^0. These tuples are then projected to form $p^0_{i,j} = a_{i,j} \cdot x^0_j$. Further, the $\boldsymbol{\tau}$ and the $\boldsymbol{\gamma}$ operators reduce the $p^0_{i,j}$ values targeted for the same index i using the $\sum$ operation to produce the final value of x^1_i. In this case, the duplication, projection, sorting, and reduction operations can be fused into a multi-pass radix partitioning operation. This type of fusion is demonstrated by the work in [4]. If implemented using their **`milk`** OpenMP extension, it would fuse the duplication and projection, that is, the $\boldsymbol{\pi}_{[i,j,p^0_{i,j} \leftarrow a_{i,j} \cdot x^0_j]}\left(\bowtie_{[j]}(A^c, X^0)\right)$ operation, with the first pass of the radix partitioning. In addition to this, the OpenMP extension fuses sorting ($\boldsymbol{\tau}_{i,j}$) and reduction ($\boldsymbol{\gamma}_{[i,sump^0_{i,j}]}$) in each partitioning pass whenever possible within the spatial bounds of the cache. To achieve this, each partitioning pass distributes the projected values to appropriate bucket in cache resident containers that allow reduction on the values with the same index i before bursting the container's data to the global memory. An alternative approach to optimize the mapped reduction is to delay the duplication as much as possible to later passes of radix partitioning, and perform the reduction in the later passes when most of the projected values with the same index i are localized in the cache. This transformation reduces the cumulative bandwidth cost of the radix partitioning since the delayed duplication reduces the I/O volume. This transformation is demonstrated in [5].

3.5.4 Optimization for static applications:

The scattered access pattern of the Version 1 depends on the order in which the tuples of X^1 and X^0 are stored as well as the non-zero structure of the matrix A. Many scientific computations produce a sparse matrix that does not change its non-zero structure as long as their computational grid does not change its topology. For these cases, which we will refer to as *static irregular*, it is possible to improve the spatial locality of the scattered accesses using a space filling curve order [6], [7] or the reverse Cuthill-McKee order [8], [9]. These orderings improve the cache hit rate of the scattered accesses which reduces the latency cost of the computation. Since this is a low flop/byte computation, the reordering significantly improves the flop rate of the computation.

For the streaming and QSM versions, the static structure of the sparse matrix provides further opportunities to improve the performance. First, both the versions need to generate the duplication map (M^{-1} in case of the Version 2-Case 1B and D^{-1} in case of the Version 3) only once. Since it can be reused several times later, these algorithms incur only the cost of duplication and sorting each time the values x^0_j in X^0 change. Second, the spatial locality improvement also results in lower duplication and sorting cost for the QSM version since most of the references are served through the look-back of X^0 after the reordering. Section 3.6 shows the savings in the bandwidth cost when a locality improving reordering is used for the QSM SpMV algorithm.

3.5.5 Optimization of the duplication phase:

The duplication phase in Version 2 and 3 requires a scan over all the tuples of X^0 even though some of the values are not duplicated. For the sparse matrices in static irregular computations, the duplication phase can be further optimized by reordering the j index space such that the tuples in X^0 that need duplication are gathered in one part and the remaining tuples are gathered in the other part. The partitioning also implies reordering of the i index space which implies reordering of the tuples of X^1 and A^r, since X^1 is often consumed by subsequent SpMV computation (i.e. $X^2 = A^r \cdot X^1$). However, once reordered, the duplication, sorting and computation schedule can be reused several times, thus amortizing the cost of reordering.

3.5.6 Operator slicing:

Both the duplication and sorting phases of the QSM version (Figure 6) are bandwidth bounded. When a level-3 QSP is available, these phases can be carried out in the M1 buffer to take advantage of its higher bandwidth. If these phases take larger space than the capacity of the M1 buffer, the subsequent equijoin operator, $Q^0 = \bowtie_{[j]}(A^r, X^0, R^0)$, needs to be sliced. This is done by slicing its input and output arrays so that the space required to generate each slice of R^0 is within the capacity of the M1 buffer. Then, a QSP executes one C_g-step for the equijoin operator that contains several nested C_1-steps for the duplication and sorting phases for each slice of R^0.

3.5.7 Clustering for the level-3 QSM:

In a large queue streaming application, there may be a chain of scattered access computations, each implemented as a queue streaming kernel. Each kernel consumes streams produced by the preceding kernel and produces streams that are consumed by the succeeding kernels. When level-3 QSM is available, the M1 buffer can be used to store the intermediate streams of a cluster of kernels. Since the M1 buffer has higher bandwidth than the global memory, the grouping results in a lower execution time of the cluster. If the intermediate streams are larger than the capacity of the M1 buffer, the input and output streams of the cluster can be sliced such that the intermediate streams fit in the M1 buffer.

3.5.8 Generalization of the SpMV computation:

Many algorithms in scientific simulation code have performance characteristics similar to the SpMV computation such as low computational intensity and scattered data access. The transformations and optimization techniques discussed in this section are applicable to all these cases. In [5], we presented the case of gradient computation and experimentally demonstrated a speed up of $\approx$ 1.45 of a level-0 QSM implementation over the traditional scattered access implementation optimized using Hilbert space filling curve order [6], [7]. The QSM transformation amortizes the latency of the scattered access by grouping and reordering the references. Since latency fundamentally lags bandwidth [10], [11], a QSM implementation for other scattered access algorithms would also show speed up.

3.6 Cost Analysis

Each operator in the SpMV computation is I/O bounded due to low flop/byte ratio. Therefore, their cost depends on the I/O volume as well as the type of access (scattered or streaming). For the QSM implementation, the cost is bandwidth bounded since each operator performs streaming access to its operands. In this section, we analyze the bandwidth cost of the $Q^0 = \bowtie_{[j]} (A^r, X^0, R^0)$ operation in the QSM implementation (Figure 6).

The bandwidth cost of the operation is proportional to its I/O volume which is equal to the length of A^r, R^0, X^0, and Q^0. While the length of A^r, X^0, and Q^0 is invariant of the QSP design parameters, the length of R^0 depends on the look-back distance available to a read queue on the processor as well as locality of references to X^0 presented by the map M. With no look-back, the duplication is required for each individual reference to each tuple of X^0. The total number of such references is equal to the length of A^r. This gives us the worst case bandwidth cost of the operation. On the other hand, if a QSP has infinite look-back, there is no need for duplication since the entire X^0 array can be loaded in the queue buffer and accessed with look-back. This case corresponds to the best-case bandwidth cost of the operation. On a QSP with a finite, non-zero look-back, the bandwidth cost is between these two extremes. Figure 8 shows the percentage increase in the cost with respect to the best-case cost for various values of the look-back distance (d_2) and the ratio of the look-ahead distance to the look-back distance ($\frac{d_1}{d_2}$). The graph also shows the worst-case cost with respect to the best-case cost. The graph uses dashed lines and solid lines for the cases when the computation uses look-back in R^0 and when it does not, respectively. The plot is experimentally derived from a matrix based on a tetrahedral mesh consisting of $\approx$ 5.5 million cells that are ordered with the Hilbert space filling curve [6], [7]. Note that the worst-case cost is also the cost of the join operation in Version 2, Case 1B algorithm that does not use look-back. Also, note that the worst-case cost is independent of the locality of X^0 references. On the other hand, the cost saving in case of the QSM version is due to the combined effect of the locality improvement as well as the look-back access. Figure 8 shows that with sufficient look-back, the bandwidth growth is a small fraction of the best-case cost and much lower than the worst-case cost. This also implies that the duplication and sorting phases also require a small amount of time despite several passes required over the duplicated data.

Part of the reason for the small increase in the bandwidth cost of the QSM version is the low dimensionality of the support graph of A. With appropriate reordering of the rows and columns of the matrix, the spread of the references to X^0 becomes compact. As the dimensionality grows, the spread grows. This increases the duplication due to reduced look-back hit rate. In this case, the Version 2-Case 1B algorithm may perform better than the Version 3 algorithm if used in conjunction with the appropriate optimizations demonstrated in [4], [5], [12]. Note that with the increase in dimensionality, the latency cost of the scattered access version also grows due to reduced cache hits.

For scattered access version on a cache-based processor, there is no need for explicit duplication and sorting since the cache automatically copies the cache-missed references from DRAM to the cache. In this case, the best-case and worst-case costs correspond to the scenario where the cache is of infinite capacity and zero capacity, respectively. However, the cost in terms of the wall clock time in the best-case and worst-case scenario would be higher than the best-case and worst-case costs of the QSM version since the cache misses due to scattered access to X^0 would incur full DRAM latency. Therefore, the extra cost of duplication and sorting in the QSM algorithm will be less than the cumulative latency cost in terms of the wall clock time.

Instead of the QSM version, we can also use the Version 2-Case 2 algorithm that uses A^c instead of A^r. This version does not require duplication and sorting. However, it requires sorting of the output of $\boldsymbol{\pi}$ (Q^0 in Figure 3) before passing it to $\boldsymbol{\gamma}$ for reduction. The sorting happens over the entire Q^0. For one time computations or for cases where the non-zero structure of A changes frequently, this may be acceptable since it avoids the map generation. However, in case of static computations, the cost of sorting will accumulate over several SpMV computations. In this situation, the QSM version is beneficial, for two reasons. First, the map generation is required only once. Hence, its cost is amortized. Second, the duplication and sorting is required over a small portion of X^0. This reduces the cumulative cost of sorting over the reuses.

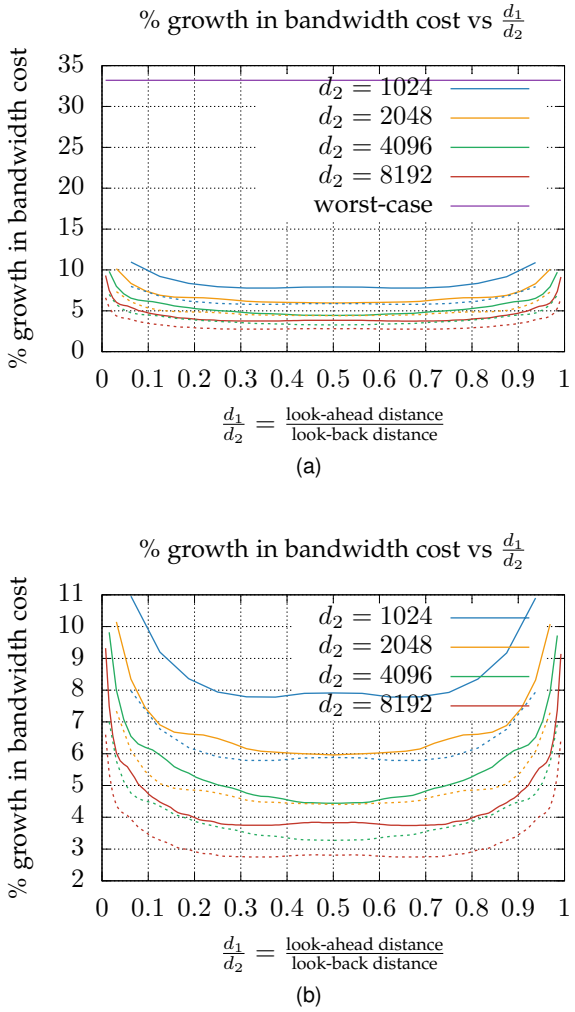


Fig. 8. Percentage of growth in the bandwidth cost of the join operation in the QSM implementation of the SpMV computation using a matrix derived from tetrahedral mesh of ≈ 5.5 million cells on a QSP with 16 cores.

3.7 Architectural Implications

A larger look-back distance corresponds with a lower cost in the QSM version. Therefore, a larger M0 buffer is desirable. However, being a resource closer to the cores, the M0 capacity is limited. Also, increasing M0 capacity increases the cost of the look-back access. Therefore, the look-back distance is finite.

The M0 buffer space required to hide the latency grows with the block size of the immediately next level of memory. On the level-1 QSM, this level of memory is the global memory. However, for higher capacity, it is necessary that the global memory has larger block sizes. In this case, the M1 buffer in the level-3 QSM increases the look-back distance since it has much smaller block size compared to the global memory.

In case of the slicing optimization, higher the bandwidth and larger the capacity of the M1 buffer, better is the performance of the copy an duplication phases.

If multi-pass radix partitioning algorithm is used for the equijoin or the group-by operator (as demonstrated by the examples of operator fusion transformation), the number of partitioning buckets depends on the number of queues available on each core. Larger the number of cores, larger are the buckets in each partitioning pass, and smaller is the number of passes. However, the number of queues is limited by the M0 buffer capacity as explained in Section 2. Similar to the sorting, the rate of improvement in the execution time of the radix partitioning reduces with increasing number of buckets. Hence, a moderate number of queues and the M0 buffer capacity are enough to obtain good speed up.

4 MapReduce

In MapReduce [1] programming model, user supplies a map and a reduce function. The reduce operation is typically associative and commutative. First, the framework applies the map function to every key-value pair in an input array to generate a list of intermediate key-value pairs. This is called as the map phase. Next, the framework applies the reduce function to all the intermediate pairs with the same key to reduce their values. This is called as the reduce phase. This phase generates the output key-value pairs. Finally, the merge phase sorts the output key-value pairs by the key. The merge phase is optional.

Talbot *et al.* [13] presented a general characterization of a MapReduce workload based on three dimensions of the map and reduce operations - the map task to intermediate key distribution, the number of values per key, and the computation per task with respect to the framework overhead. The key to improve performance of a MapReduce operation on a shared-memory parallel processor is to choose a correct implementation of the container that stores the intermediate key-value pairs and use combiners on as many intermediate key-value pairs as possible to compact the intermediate data before it is written to the memory. Various MapReduce frameworks propose refinements to the crude description of the algorithm given above to obtain speed up on a multi-core and many-core processor, or a GPGPU [13], [14], [15], [16], [17], [18]. The MapReduce model satisfies the requirements for a QSM since each map and reduce task is data independent. Depending on the characteristics of the intermediate key-value data, and the map and reduce operation, both the map and reduce phases present load imbalance if performed using a static schedule since the time required for each map and reduce task is variable. Therefore, we need to use a dynamic schedule. In this section, we demonstrate the use of the level-2 and level-3 model for the MapReduce operation. We assume that the reduce operation is associative and commutative.

The map phase is straightforward. On a level-2 model, it requires the shared queues to form and execute a dynamic schedule. Each core picks the next available chunk of the input key-value pairs from the centralized work-queue and emits the intermediate key-value pairs to an exclusive write queue. Optionally, the core can store these pairs in the local memory, sort them and perform reduce operation on the values of the same key before streaming out the intermediate key-value data to the global memory.

The reduce phase is carried out using merge sort algorithm combined with the reduce operation. Each two-way merge takes two sorted blocks of the intermediate key-value pairs and merges them. As keys from the two blocks are

being compared, it also checks for equality of the keys and performs reduce operation if the keys are equal. As a result, the resulting merged block may be of length smaller than the sum of the lengths of the input blocks. Since this is a combination of the merge and the reduce operation, we call it merge-reduce operation. Since the length of the blocks to merge-reduce may be variable, this phase also needs a dynamic schedule. At the end of the merge-reduce passes, the final output is a sequence of the output key-value pairs sorted by the key. If a level-3 model is available, the M1 buffer could be used to perform merge-reduce operations on the intermediate key-value pairs stored in the buffer before streaming it out to the global memory. This reduces the total cost of the algorithm due to reduction in the volume of the global memory data access.

This scheme will produce load imbalance in the last few merge-reduce passes as the number of blocks to merge-reduce drops. However, since the intermediate data is already compressed, these passes will require a much lower volume of the read/write data from the global memory. Therefore, the idle cores will not severely degrade the performance.

4.1 Execution Time on a Level-2 Model

For this analysis, we assume that the dynamic schedule is optimal and it completely hides the overhead of activation and deactivation of the exclusive queues using parallel slackness. Let N_1 be the number of input keys and N_2 be the number of output keys.

The map operation emits intermediate key-value pairs from for each input key-value pair. The transformation could be compute-bounded (for example, in case of the k-means clustering). Even if it is not compute bounded, the map phase is more likely to be compute-bounded due to the local memory sorting and reduction performed by the cores before writing the sorted block to the global memory. Therefore, we assume that the map phase is compute-bounded. If a single map task takes, on an average, t_m units of time including the time for local memory sort, the total time required for the map phase is $t_m \times N_1 + (o_s + o_g)$, where N_1 is the number of input key-value pairs and the overhead is due to the activation and deactivation of the shared queue and initial activation and final deactivation of the exclusive queues of an execution context.

For the analysis of the merge-reduce passes, we assume that the map phase produces intermediate key-value pairs in an array in which the keys are distributed randomly and uniformly across the entire length of the array, and the array contains sorted blocks of length u. We also assume that the reduction operation is an inexpensive associative and commutative operation (e.g. multiplication, addition etc.) that makes each merge-reduce pass bandwidth-bounded. If each key appears on an average r times in the output of the map phase, the total compression ratio of all the merge-reduce passes is r. This compression is achieved in multiple passes. Since the keys are distributed uniformly and randomly, we can assume that each $i = 0, ..., s-1$ merge-reduce pass achieves a uniform compression ratio of r_a such that $r = r_a^s$. Each i^{th} pass consumes blocks of length $(2r_a)^i u$ and produces blocks of length $(2r_a)^{i+1} u$ using $\frac{1}{2^{i+1}} \left\lceil \frac{rN_2}{u} \right\rceil$ merge-reduce operations. Therefore, the total number of passes is $s = \left\lceil \log_2 \left\lceil \frac{rN_2}{u} \right\rceil \right\rceil$. Initial passes achieve full system bandwidth while the last few passes achieve lower bandwidth due to reduced concurrency. The effective bandwidth of the i^{th} pass is $B_{p,i} = \min\left(B_g, \min\left(P, \frac{1}{2^{i+1}} \left\lceil \frac{rN_2}{u} \right\rceil\right) \cdot B_c\right)$. Therefore, the execution time of the merge-reduce passes is $N_2 \times S(\tau) \times \sum_{i=0}^{s-1} \left(\frac{r_a^{s-i} + r_a^{s-i-1}}{B_{p,i}}\right) + s \times (o_s + o_g)$, where $S(\tau)$ is the size in bytes of an intermediate key-value pair. Therefore, the total time of the MapReduce operation is given by Equation 9.

$$
\begin{aligned}
t =& t_m \times N_1 + (o_s + o_g) \\
&+ N_2 \times S(\tau) \times \sum_{i=0}^{s-1} \left(\frac{r_a^{s-i} + r_a^{s-i-1}}{B_{p,i}}\right) + s \times (o_s + o_g)
\end{aligned}
\tag{9}
$$

4.2 Architectural Implications

As u increases, the number of passes of the merge-reduce phase decreases. Similar to the phase 1, version 2 of the sorting algorithm, u is limited by the size of the local memory and the maximum allowed burst size for a queue. Therefore, processor designers need to choose the size of the local memory and the M0 buffer carefully not to squander the valuable chip area. See Section 2 for the explanation.

Similar to the sorting algorithm, the execution time of a level-3 MapReduce computation improves with improvement in the M1 buffer bandwidth. Therefore, a high bandwidth M1 buffer is essential to gain performance.

The dynamic scheduling policy determines its effectiveness for hiding the overhead. However, larger the overhead $(o_g + o_s)$, larger is the smallest chunk size that the schedule can generate. A larger chunk size may produce load imbalance. Also, the block size of the global memory and the service time of the queue schedule affects the overhead. Therefore, these parameters also determine the effectiveness of the dynamic schedule.

5 Conclusion

The QSM fills the void of prescriptive access semantics in current processor architectures and programming models that heavily depend on reactive mechanisms for minimizing data access latency costs. The examples in this paper demonstrate how the QSM fulfills the objectives it envisioned in PART-1 about performance predictability, portability and latency-amortization of shared memory parallel algorithms. Even though the model appears too restrictive at first glance, the examples in this paper show a number of ways to use the QSM for a variety of algorithms and obtain exact execution time of an algorithm that reflects the effect that architectural choices have on algorithm performance.

While the cost analysis requires precise knowledge of a QSP's design parameters, it is beneficial for two reasons. First, the architectural parameters in the execution time estimation allow hardware designers to readily see the relationship between design decisions and an algorithm's performance. This is especially important when the attainable transistor density is near saturation, which will compel

processor designers to judiciously use the available chip area. Second, the explicit dependence on the QSP design parameters makes the cost prediction accurate and portable. Predictability is beneficial for algorithm designers since they do not require run time performance tuning as long as the algorithm conforms to the QSM. The performance predictability the model provides is an essential ingredient to the the design of runtime schedulers since they can reliably predict performance of scheduling options without concern about subtle interactions between memory access patterns and complex and reactive memory systems that are a common cause for performance surprises in modern architectures.

A QSP shares many characteristics with a cache-based processor. An important difference between them is that in QSP, the components (such as queue buffer, memory controller etc.) prioritize spatial locality over the temporal locality. As a result, a cache-based processor can be morphed into a QSP and vice versa with minimal hardware changes. This also implies that a queue streaming mode can coexist with the usual fine-grained, random-access, multi-core computing mode on the same hardware as long as a program uses only one of these modes at a time.

ACKNOWLEDGMENTS

The authors would like to thank the Center for Advanced Vehicular Systems and the High Performance Computing Collaboratory of the Mississippi State University for providing the necessary resources. This work was partially supported by the Department of Defense (DoD) High Performance Computing Modernization Program (HPCMP).

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.

[3] V. Kotlyar, K. Pingali, and P. V. Stodghill, "Compiling parallel code for sparse matrix applications," in *Supercomputing, ACM/IEEE 1997 Conference*. IEEE, 1997, pp. 10–10.

[4] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with `milk`," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation - PACT '16*, September 2016. [Online]. Available: https://doi.org/10.1145/2967938.2967948

[5] A. Zope and E. Luke, "A block streaming model for irregular applications," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 753–762.

[6] M. Bader, *Space-filling curves: an introduction with applications in scientific computing*. Springer Science & Business Media, 2012, vol. 9.

[7] D. Hilbert, "Ueber die stetige abbildung einer line auf ein flächenstück," *Mathematische Annalen*, vol. 38, no. 3, pp. 459–460, 1891.

[8] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*. ACM, 1969, pp. 157–172.

[9] E. Cuthill, "Several strategies for reducing the bandwidth of matrices," in *Sparse Matrices and Their Applications*. Springer, 1972, pp. 157–166.

[10] D. A. Patterson, "Latency lags bandwidth," *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[11] J. D. McCalpin, "Memory bandwidth and system balance in HPC systems," Invited talk, Supercomputing 2016, Salt Lake City, Utah, 2016.

[12] S. Manegold, P. Boncz, and M. Kersten, "Optimizing main-memory join on modern hardware," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 4, pp. 709–730, 2002.

[13] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: modular MapReduce for shared-memory systems," in *Proceedings of the second international workshop on MapReduce and its applications*. ACM, 2011, pp. 9–16.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Parallel Architectures and Compilation Techniques (PACT), 2008 International Conference on*. IEEE, 2008, pp. 260–269.

[15] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing MapReduce for multicore architectures," in *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*. Citeseer, 2010.

[16] W. Jiang, V. T. Ravi, and G. Agrawal, "A map-reduce system with an alternate api for multi-core environments," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE, 2010, pp. 84–93.

[17] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 523–534.

[18] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh, "Optimizing the MapReduce framework on Intel Xeon Phi coprocessor," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 125–130.

Anup Zope Anup Zope received his undergraduate degree in mechanical engineering from College of Engineering, Pune, India in 2005 and MS degree in computational engineering from Mississippi State University, USA in 2015. Currently, he is a PhD candidate of computational engineering and graduate research assistant at Center for Advanced Vehicular Systems, Mississippi State University, USA. His research interests include parallel algorithm design and analysis, performance tuning and measurement, computer architecture, and scientific computing. He is a student member of the IEEE.

Edward Luke Edward Luke received a BS degree in electrical engineering from Mississippi State University in 1987 and his MS and Ph.D. in computational engineering from Mississippi State University in 1993 and 1999 respectively. He is currently a professor in the department of computer science and engineering at Mississippi State. He is the developer of the auto-parallelizing Loci framework and the CHEM multi-physics computational fluids solver that is widely used in the aerospace community.