On Efficient Query Processing of Stream Counts on the Cell Processor

Dina Thomas [#1], Rajesh Bordawekar [#2], Charu C. Aggarwal [#2], Philip S. Yu [#3]

[#1] *Stanford University*
Palo Alto, CA, USA
[1] dinathomas@gmail.com

[#2] *IBM T. J. Watson Research Center*
Hawthorne, NY, USA
[1] { bordaw, charu }@us.ibm.com

[#3] *University of Illinois at Chicago*
Chicago, IL, USA
[2] psyu@cs.uic.edu

***Abstract*— In recent years, the sketch-based technique has been presented as an effective method for counting stream items on processors with limited storage and processing capabilities, such as the network processors. In this paper, we examine the implementation of a sketch-based counting algorithm on the heterogeneous multi-core Cell processor. Like the network processors, the Cell also contains on-chip special processors with limited local memories. These special processors enable parallel processing of stream items using short-vector data-parallel (SIMD) operations. We demonstrate that the inaccuracies of the estimates computed by straightforward adaptations of current sketch-based counting approaches are exacerbated by increased inaccuracies in approximating counts of low frequency items, and by the inherent space limitations of the Cell processor. To address these concerns, we implement a sketch-based counting algorithm, FCM, that is specifically adapted for the Cell processor architecture. FCM incorporates novel capabilities for improving estimation accuracy using limited space by dynamically identifying low- and high-frequency stream items, and using a variable number of hash functions per item as determined by an item's current frequency phase. We experimentally demonstrate that with similar space consumption, FCM computes better frequency estimates of both the low- and high-frequency items than a naive parallelization of an existing stream counting algorithm. Using FCM as the kernel, our parallel algorithm is able to scale the overall performance linearly as well as improve the estimate accuracy as the number of processors is increased. Thus, this work demonstrates the importance of adapting the algorithm to the specifics of the underlying architecture.**

I. Introduction

Recent technological advances have led to a proliferation of data *streaming* applications that often process large quantities of data that can potentially grow without limit at a rapid rate [18]. A key data streaming application involves determining frequency statistics of the stream items in *real-time*. The number of items in a stream items and the item domain size can be very large. For example, in web click streams or phone call streams, the number of possible unique items (i.e., web pages or phone numbers) could easily range in the order of hundreds of millions or even billions. Obviously, to satisfy the memory and real-time execution constraints, the input stream data can not be stored in its entirety. Therefore, the counting applications employ algorithms that strive to maximize the computational performance while minimizing the memory usage.

A well-known memory-efficient technique for counting items from data streams uses probabilistic data structures [10], [8], [1], [2], [3] such as the sketches to condense the input stream data into a summary. The results for a query can then be extracted from these condensed summaries. While the sketch-based approach reduces the space complexity of the counting process, additional modifications are necessary for improving its computational performance. A traditional approach for accelerating computational performance involves partitioning the work across multiple processing entities and executing them in parallel. Recently, such partitioning approaches have become even more practical due to wide availability of systems that use *multi-core* processors [6], [5], [11] such as the Cell processor. The Cell is a heterogeneous multi-core processor with 9 cores, one controller (PPE), and 8 co-processors (SPEs), connected to on-chip memory via high-bandwidth, low-latency interconnection fabric. The Cell's 8 co-processors (SPEs) support a short-vector data-parallel (or Single-Instruction Multiple-Data (SIMD)) instruction set which can be used to accelerate vector computations (unfortunately, such architectures are not ideal for algorithms with lots of conditionals). Although the Cell was initially designed as a game processor, it is being increasingly used in blade servers for developing commercial and scientific applications [15].

Our need for a high-performance counting algorithm arose in the context of designing stream statistical tracking algorithms in the System S efforts at IBM[19]. System S is a distributed stream processing infrastructure designed to support applications that utilize different stream analysis algorithms working cooperatively on multiple source streams. The System S uses a variety of computing resources, including the Cell

processor, to implement various data analytics algorithms.

In this paper, we describe the design and implementation of a parallel algorithm for counting stream items on the Cell processor using a probabilistic sketch data structure. Although our algorithm is currently implemented only on the Cell processor, its features make it applicable to a wide range of multi-core processors with similar architectures, e.g., network processors [5], [11]. This class of the multi-core processors exhibit similar architectural traits: a heterogeneous processor with a single control processor mananging a set of specialized co-processors, each with limited local scratch memory. Parallel applications executing on such processors achieve scalability by dividing the work across multiple co-processors, where each processor is executing the same sequential program (known as the Single-Program Multiple-Data (SPMD) implementation). Our parallel counting algorithm also uses data partitioning to distribute stream ingestion and querying across multiple processors. Unfortunately, data partitioning strategy can lead to an uneven distribution of items across the processing cores. Therefore, items with even modestly high frequencies can sometimes have low frequencies on some processors. Current sketch-based counting algorithms, especially under limited memory, can not accurately estimate frequencies of low-frequency items. The errors in estimating low-frequency items can result in high aggregate errors in the final estimate. Our parallel implementation follows the SPMD programming model and uses a novel sequential counting algorithm, referred to as the Frequency-aware Counting (FCM) algorithm, as the sequential kernel. The FCM algorithm is an adaptation of the Cormode-Muthukrishnan Count-Min (CM) [4] algorithm and incorporates two novel extensions to address the low-frequency estimation problems due to data partitioning. Our implementation on the Cell processor has been optimized to exploit Cell's architectural cand programming capabilities. We have evaluated our implementation using Zipfian and Normally distributed datasets using FCM and CM as sequential kernels. We have demonstrated that in the parallel scenario, the FCM algorithm improves the estimation quality over the CM algorithm over the entire frequency range of the input data. Our experiments illustrate that simply partitioning data across multiple processors does not lead to an improvement in estimation quality as the number of processors is increased.

This work makes the following key contributions:

- **Implementation of the Parallel Algorithm on the Cell Processor:** The Cell processor exhibits a number of architectural constraints that make any Cell-based parallel implementation particularly difficult: The Cell processor has only one PPE for global handling of the incoming data, though the real computational capabilities reside in the SPEs. The relative weakness of the PPE implies that one cannot perform sophisticated tricks during the partitioning of the data. Also, high-performace SPEs also imply that one must limit the amount of communication among different SPEs as far as possible to achieve the maximum scalability. The real challenge in designing a scalable parallel stream counting algorithm arises in effectively adapting a sequential counting algorithm to the Cell, since any attempt to add sophistication to the partitioning of the data between SPEs inevitably leads to more work on the PPE side and more intra-processor communcation among the SPEs. In addition, each SPE has a small local store and the SPE codes need be developed using the SIMD instruction set. Our implementation uses two efficient data partitioning strategies: block-based and hash-based distribution, that do not burden the PPE. Also, the choice of the CM algorithm as the sequential kernel was critical as it can use simple data partitioning algorithms and the core algorithm can be executed over multiple SPEs without any inter-SPE communication.
- **Experimental Evaluation of Sketch-based Counting Algorithms:** The sketch-based counting algorithms are suitable for SIMD processors like the Cell as most of their key operations are non-branchy and commutative in nature. Thus, these operations can be accelerated using SIMD instructions. While a wide variety of sketch-based counting algorithms (such as the CM, CCFC, and the AMS sketch [1], [2], [4]) are available for adaptation, the relative *experimental* effectiveness of different algorithms is quite unknown. To the best of our knowledge, this work is the first to independently evaluate different sketch-based algorithms experimentally, and demonstrate that the CM sketch is actually superior to other algorithms from both the accuracy and space consumption. While this was necessary for the purpose of our choice of implementation, we note that this is an independent contribution which is likely to be useful to those interested in exploring sketch-based techniques for different stream counting applications.
- **Adaptation for Low-Frequency Items:** We have adapted the CM algorithm to address the inaccuracies caused by the low-frequency items in the parallel algorithm. We have implemented the adapted algorithm, FCM, on the SPEs of the Cell processor. FCM extends the CM algorithm in two aspects: First, it uses an additional space-conserving data structure to *dynamically* capture the *relative* frequency phase of items from an incoming data stream. Second, the FCM uses a variable number of hash functions for each item as per its current frequency phase (i.e., "high" or "low"). Our implementation has validated several key advantages of the CM algorithm that makes it suitable for the Cell processor. Its limited space consumption allows us to store the sketches in the small local scratch memories of the Cell SPEs and many of its key computations, e.g., hash value computations or computing the minimum values, can be easily accelerated using Cell's SIMD instruction set.

This paper is organized as follows. In Section II, we present an overview of the Cell processor. Section III examines a number of issues in parallelizing the sketch-based counting algorithms and discusses our proposed parallel algorithm. Section IV presents an experimental evaluation of three sketch-

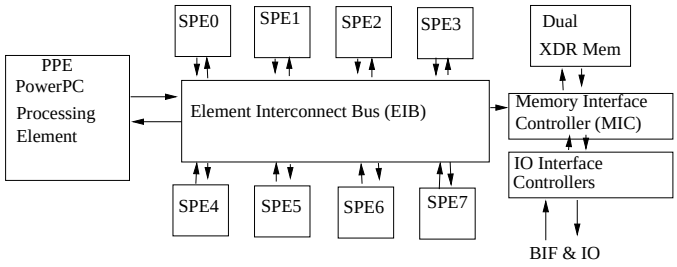

Fig. 1. Architecture of the Cell Processor.

based stream counting algorithms. Section V discusses FCM, our adaptation of the CM counting algorithm. Section VI presents details of our implementation of the parallel algorithm on the Cell processor. Section VII presents the experimental evaluation results. We discuss related work in Section VIII. The conclusions are summarized in Section IX.

II. Overview of the Cell Processor

The Cell processor is designed primarily for compute- and data-intensive streaming applications. It is a 64-bit single-chip heterogeneous multiprocessor with nine processing cores: one general-purpose processor, called the PPE and eight special purpose co-processors, called the SPEs. Both the PPE and the SPEs run at the same clock frequency. These processors are connected via a high-bandwidth internal bus and can access the shared, coherent main memory (Figure 1). The PPE is a PowerPC-based RISC core and runs the operating system to manage system resources of the entire multiprocessor. It has 32 KB instruction and data L1 caches, and a 512 KB integrated L2 cache. The PPE can support 2 simultaneous hardware threads. Each SPE is also a 64-bit RISC core natively supporting a short-vector 128-bit single-instruction multiple-data (SIMD) instruction set. The Cell SPE is a dual-issue, statically scheduled SIMD processor. Each SPE holds a 128-entry 128-bit register file to execute these SIMD instructions. The SPE SIMD instruction set can support multi-way (2,4,8, and 16) data parallelism. The SPE control unit can issue 2 instructions per cycle, in-order. Instead of caches, each SPE has a 256 KB private local store which is used to hold both instructions and data. The SPE load and store instructions manage the data transfer between the register file and the local store. Each SPE has dedicated single-precision floating point and integer vector units. Although designed primarily as a SIMD processor, the SPE can also execute scalar computations. However, as the SPEs lack branch prediction hardware, execution of scalar SPE code with conditionals is not optimal. Finally, there is no virtual-memory support on the SPEs and the only way to access the main memory from the SPEs is via explicit asynchronous direct memory access (DMA) commands. The DMA is also used for communicating among different SPE local stores [13]. The PPE and SPEs have different instruction-set architectures (ISAs) and the Cell compiler automatically generates the appropriately optimized native code.

The Cell's multi-core architecture can be exploited via a variety of parallel programming models [6]. A Pthreads like task-parallel programming model enables a host program executing on the PPE to spawn multiple threads which can execute different programs on different SPEs. Each SPE program can then use the SIMD instructions to exploit Cell's data parallel facilities. The SPE code performance can be further improved by using instruction-level parallelism via SPE's dual execution pipes. The Cell also supports a shared-memory programming model where multiple SPEs can share data from their local stores using DMAs over a distributed shared address space. Other supported programming models include function offloading and computation acceleration models [6]. Although the PPE is a 2-way SMT (simultaneous multi-threading) architecture, its small memory subsystem (32 KB L1 and 512 KB L2 caches) makes it substantially weaker processor. The PPE also acts as a controller for the SPE applications and assists in virtual memory management of the entire processor. In practice, to exploit the capabilities of the Cell processors, the programs should be designed such that they use the PPE only for coordinating purposes, not for heavy-duty computations.

Cell's architectural and programming features generate unique constraints on the design and implementation of any application. An ideal Cell application should be easily partitionable into components that can run on the PPE and the SPEs, with the PPE component executing the lightweight coordination functions. As there is no virtual-memory support, the SPE code should have a smaller memory footprint and should execute structured computations which could be easily vectorized using the short-vector SIMD instruction set, with as few branches as possible. Further, as there is no support for code overlays, the SPE binary code should be small enough to fit the local store. The Cell architecture has very high intra-SPE and I/O bandwidths, and is suited for applications that process high-bandwidth data. Fortunately, the stream counting application satisfies most of these constraints.

III. Parallel Stream Processing

Sketch-based probabilistic counting algorithms can be evaluated using three criteria: (1) Quality of results, (2) Space utilization, and (3) Time to store and query the stream datasets. The key goals of parallelizing such algorithms are to improve both the result quality, and times for stream ingestion and processing, while maintaining the space utilization. In any skech-based sequential algorithm, the most expensive operations are the update and querying of the sketch data structure as it is updated for every item in the stream. To achieve scalability, any parallel algorithm should try to acccelerate these operations.

While devising a parallel algorithm, we assume an abstract parallel machine consisting of p processing units sharing the main memory. In addition, each processing unit has private local memory. These processors can communicate either using shared data structures or via explicit messages. This abstract model captures the architecture of most current multi-core

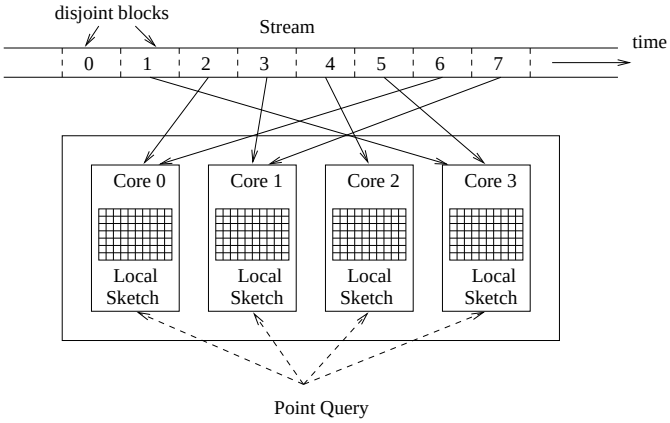

Fig. 2. Parallel stream counting via distributing stream items over multiple processing cores.

processors designed for stream data processing (such as the Cell [6] and the Network processors [5], [11]). Thus, our algorithm can be implemented on a wide variety of available software and hardware platforms.

As the sketch-based counting problem is commutative in nature, it can be easily parallelized in the SPMD model (i.e., each processor executes the same program) using two approaches. The first approach would partition a sketch data structure across multiple processors. Each processor reads the **entire** stream and uses a sequential counting algorithm to update its local sketch for every item. At the query time, for an input item, **each** processor uses it local sketch to compute its local estimate. These local estimates would then be processed to compute the final result. Alternatively, each processor reads **disjoint** parts of the input stream and updates local sketches only for those items read by the processor. At the query time, all processors compute its local estimates which are then get aggregated to compute the final result. The first approach is clearly not scalable as it replicates work across participating processors, both during ingestion and quering phases. Hence, we use the second approach for developing our parallel algorithm. Figure III illustrates our parallel stream processing strategy.

The performance of the parallel algorithm depends on: (1) Stream data partitioning strategies and (2) The sequential algorithm. The data partitioning strategies divide the work across multiple processors by distributing the stream data. The data partitioning modifies statistical characteristics of the input data stream. As a consequence, a frequent stream item may appear as a *low frequency* item to some processors. This can lead to increased errors in computing per-processor local counts, in particular, when using limited memory. As the final result is computed by adding local counts, the data partitioning strategies also affect the extent of the error in the final result. Our algorithm uses two approaches for partitioning input streams:

- **Hash-based partitioning:** This approach uses a value-based partitioning strategy in which the coordinating processor hashes the input stream values into p buckets. The buckets are then distributed over different processor groups (e.g., 8 processors can be partitioned as 4 processor groups, each containing 2 processors, or 8 processor groups, each with an individual processor). Each processor within a processor group reads disjoint sections of a bucket and populates its local sketch. During the querying phase, a coordinating processor hashes the query item into a bucket. **Only those processors** that are associated with this bucket are queried. This value-based partitioning may lead to unequal distribution of data across the processor groups which can cause load imbalance. This is generally not an issue when the number of processor groups is small (Section VII). In contrast, this method of partitioning groups together all occurrences of items with a particular value. Thus, the *relative* frequency of items observed by a processor group is higher than in the input stream. Also, in general, the complexity of hashing may be non-trivial and in case of the Cell processor, may burden the PPE.
- **Block-based partitioning:** In this method, no pre-processing is performed on the input stream. As the stream is read by a coordinating processor, it is divided into equal disjoint chunks. Each chunk is sent to a processor in a round-robin fashion. Unlike the hash-based partitioning, block-based partitioning distributes work equally among the processors. Also, the coordinator needs to do less work as there is no need to compute hash values per item in the input stream. However, unlike hash-based partitioning, all processors are queried for each query. When the coordinating processor sums up the estimates from each unit, it also sums up the error returned by each unit. Hence the final error bound of this method is p times the error of a single processor, where p is the total number of processors. In case of hash-based partitioning, the final error bound is p' times the error of a single processor, where p' is the number of processors in the processor group.

The next Section addresses the issue of selecting the sequential counting kernel.

IV. Selecting the Sequential Counting Kernel

As our target multi-core processors have limited per-core memory (e.g., 256 KB for the Cell, 32 KB for the Intel IXP2800), it is necessary to select a sequential counting algorithm with the lowest space consumption. In spite of the wide variety of algorithms available, the literature on the relative *experimental behavior* of the different algorithms is relatively sparse. Therefore, our first step was to examine the different sketch-based algorithms for their suitability for adaptation to the Cell processor. Aside from the algorithmic structure of the particular method, one of our concerns was the raw effectiveness of the different methods in terms of their tradeoffs between accuracy and space requirements. For selecting the sequential counting kernel, we evaluated three well-known counting algorithms: AMS [1], Count Sketch

(CCFC) [2], and Count-Min (CM) [4]. All of these algorithms use some form of the probabilistic sketch data structure which gets updated using random values generated via multiple hash functions. However, they differ in the number of hash functions used, the size of the sketch and the way the sketch is updated (see Section VIII for more detailed comparison).

Algorithm	Hash range w	# Hash Functions d	Space
AMS	1	$2log(\frac{1}{\delta})$	$\frac{16}{\epsilon^2 d}$
CCFC	$\frac{8}{\epsilon^2}$	$log(\frac{n}{\delta})$	wd
CM	$\frac{e}{\epsilon}$	$log(\frac{1}{\delta})$	wd

TABLE I

COMPARING ALGORITHMIC CHARACTERISTICS OF THE THREE SELECTED STREAM COUNTING ALGORITHMS: AMS, CCFC, AND CM.

Table I presents an analytical comparison of the total space usage, the number of hash functions used, and the hash function ranges, between the AMS, CCFC, and CM algorithms. For comparison purposes, we use a pair of user-specified parameters, error bound, ϵ, and probability bound, δ. For the CCFC and AMS algorithms, the error is within a factor of ϵ with the probability of $1-\delta$, and for the CM algorithm, the error is within a factor of ϵ times the L1-norm of the sketch with the probability of $1-\delta$. As Table I illustrates, for a given ϵ and δ, for CM and CCFC algorithms, the space utilization is determined by the number of hash functions and the hash function range. Among the three algorithms, the CM algorithm is the most space-efficient and the space complexity of the other two algorithms is worse by an order of magnitude.

To experimentally compare these algorithms, we analyzed the implementations of these three algorithms from the Mass-DAL Public Codebank[1] against 16 MB (4M Integers) of Zipf ($\lambda = 1.1$) and Normal ($\sigma = 20$) data. We ran a point query using $\epsilon = 0.1$ and $\delta = 0.001$, to compute the frequency of a stream item against every member of the dataset and compared the results computed by the algorithms with the actual frequency. Figures 3 and 4 illustrate the results for the Normal ($\sigma = 20$) and Zipf ($\lambda = 1.1$) datasets. The X-axis represents the input data set sorted using the actual frequency and Y-axis represents the frequency calculated by various algorithms.

As these Figures illustrate, for both datasets, for the entire frequency range, the CCFC algorithm computes the best approximations while the AMS algorithm performs the worst. The quality of approximation by the CM algorithm improves as the item frequency increases. However, the CCFC algorithm requires far more space ($log\frac{n}{\delta}\frac{8}{\epsilon^2}$) than the CM algorithm ($log\frac{1}{\delta}\frac{e}{\epsilon}$) and its space consumption increases quadratically ($O(\frac{1}{\epsilon^2})$) as the error parameter ϵ is reduced. In our experiments, the CM sketch required only 1536 bytes whereas the CCFC sketch used 51816 bytes and the AMS sketch used 83816 bytes. Furthermore, the CCFC performance degraded substantially when run with space comparable to the CM

[1]www.cs.rutgers.edu/∼muthu/massdal.html.

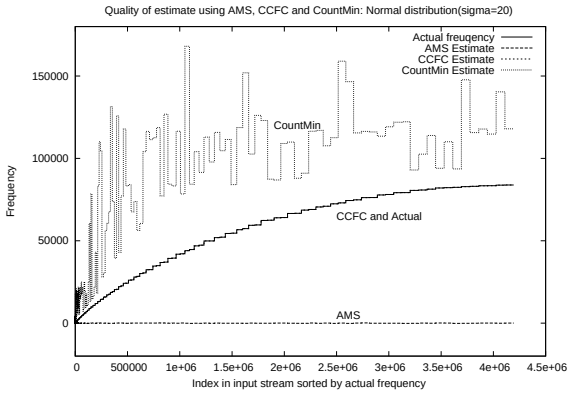

Fig. 3. Evaluation of the three selected Counting Algorithms on a 16 MB Normal ($\sigma = 20$) dataset.

algorithm (for the Normal ($\sigma = 20$) data, the average error per unique items increased from 0 to 30693, and for the Zipf ($\lambda = 1.1$) data, from 150 to 7100). Among the three algorithms, the CM algorithm provides the best accuracy while consuming the lowest space and its space consumption increases linearly as the error parameter ϵ is reduced (unlike CCFC, where the space consumption increased quadratically). Therefore, we decided to use the CM algorithm as the basis of our sequential kernel.

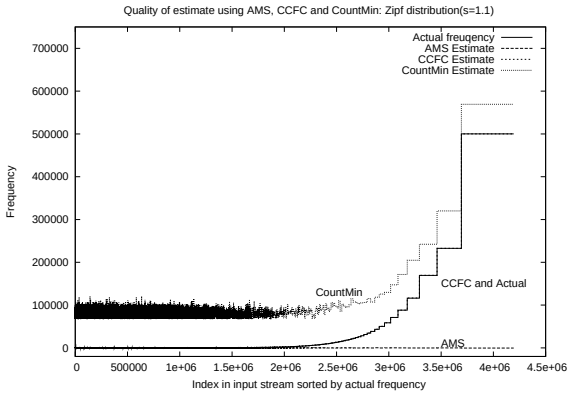

Fig. 4. Evaluation of the three selected Counting Algorithms on a 16 MB Zipf ($\lambda = 1.1$) dataset.

V. FCM: A FREQUENCY-AWARE COUNT-MIN ALGORITHM

Unfortunately, Figures 4 and 3 illustrate that the accuracy of estimates computed by the CM algorithm decreases for the low-frequency items. Thus, even with the CM algorithm, our inability to full control the distribution of items to different processors can exacerbate the errors caused by uneven distribution of items across processors. To alleviate this problem, we

make some adaptations to the CM algorithm which makes the behavior of the algorithm more *frequency-aware*. An important part of the reason that we have to modify the underlying sequential kernel is that the weak PPE on the Cell Processor limits our ability to add too much sophistication in the global distribution phase. Therefore, more intelligence needs to be added to the algorithms used on the SPEs.

Given an error bound ϵ and a probability bound δ, FCM uses the same sized sketch data structure as the CM algorithm. The FCM sketch is a set of d uniform pairwise independent hash functions, each with the range w (Figure 5). We use the universal hash functions for computing hash values (i.e., hash(x)= $(a \cdot x + b) mod(P)$, where a and b are constants and P must be a prime number, such as $2^{31} - 1$). In contrast to CM, the FCM sketch uses *variable* number of hash functions for an item based on its current *frequency phase*. An item, when deemed as a high-frequency item, uses *fewer* hash functions than a low-frequency item. To reduce the number of collisions further, a subset of the d hash functions is updated per item. The subset is chosen as a hash value of the item using two additional hash functions: first one is used to compute an initial offset into the d rows and the other computes a gap between consecutive hash tables, and the subset is chosen in a round-robin fashion. This approach differs from the CM sketch where every item updates all rows of the sketch beginning with the first row. For example, Figure 5 illustrates the ingestion of a high-frequency item i_h and a low-frequency item i_l into the FCM sketch. Both i_h and i_l have different initial offsets and gaps. The low-frequency item i_l uses more hash functions (6) than the item high-frequency i_h (3). Even with these schemes, there may be a collision between the two items, as illustrated in Figure 5.

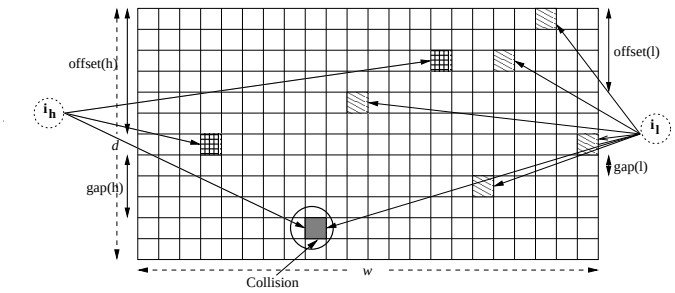

Fig. 5. Frequency-aware updating of the FCM sketch. High-frequency items update fewer hash tables than the low-frequency items.

A. Misra-Gries Frequency Counter

As the frequency of a stream item can dynamically change over time, FCM determines the frequency phase (i.e., high or low) of a stream item over the section of the stream processed so far. For dynamically detecting relative frequency phase of a stream item, we employ a space-efficient counter based on the Misra-Gries (MG) algorithm [17]. The MG algorithm approximates the set of k heavy hitter items from an online stream using $k \cdot log(k)$ space. At any point in the stream, the algorithm tracks those items whose frequency is more than

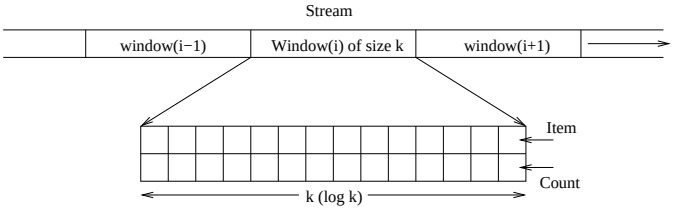

Fig. 6. Dynamic Frequency-phase Detection using the Misra-Gries(MG) Counter.

$1/k$ for some constant k, and returns an over-approximation of the set of items that satisfies this condition.

In our implementation, we use a list of $< item, count >$ pairs, called the MG counter, to keep track of counts of unique items (Figure 6). The input stream is divided into *windows* of size k. When an item is being processed, we first check if it is in the counter. If the item is in the list, its count is incremented. Otherwise, a new entry is inserted with an initial count depending on the index i of the window being read. After $i \cdot k$ items have been processed, that is, after the i^{th} window, any item with count less than $(i + 1)$ has frequency less than $1/k$ and so is deleted from the list. While processing the $(i + 1)^{th}$ window, if a new entry is observed then its initial count is set at i, which is the maximum number of occurrences of the item, after which it could have been deleted from the list. This initial count ensures that an entry whose frequency is greater than $1/k$ is not missed. FCM classifies an item as an high-frequency item if it is present in the MG counter. This classification is then used to differentially fill in the FCM sketch. We also use an initial count value as a threshold for preventing items from early sections of the stream being mislabelled as high-frequency items.

B. Sketch Updating and Querying

Figure 7 presents the FCM algorithm for processing a query to compute frequency of an item i (a *point* query). Given an item i, we first check if i is present in the MG counter (Step I1). If the item is in the counter, we treat it as an high frequency item and choose a fraction, d_h, of the d hash tables to fill in. Otherwise, if the item is not in the counter we choose a larger fraction, d_l ($d_l \geq d_h$), of the d hash tables to fill in (Step I2a). We use the value of the item to be inserted for computing the offset and gap (Step I2b). These values are computed using uniform and pairwise independent hash functions, each with range $[1...d]$. Given an item, we first calculate its offset and gap, and in a round robin fashion, identify d_h or d_l hash tables to fill in. To minimize self-collisions during the round-robin updating of the hash tables, we choose a prime value for d. For each chosen hash table, the item is hashed into a bucket i, $0 < i < w$, and the count in the bucket is incremented by 1 (Step I3).

Once the data stream has been analyzed and a sketch has been constructed, we can query the sketch to answer point query estimates. Given a query item q, using the same hash functions, we compute its offset and gap, and using the same round robin fashion, d_h hash tables are chosen from the

(Stream Ingestion Phase) For every item i in a stream:
I1. Update the MG Counter to determine its frequency phase
I2a. Determine the # of hash functions, d_h or d_l, using the freqency phase
I2b. Calculate the offset and gap using the item value
I3. Use d_h or d_l hash functions to increment the buckets in the sketch.
(Query Processing Phase) For the item q in a point query:
Q1. Calculate the offset, gap for the item
Q2. Compute an estimate from the FCM sketch by minimizing values over d_h buckets

Fig. 7. Frequency-aware Counting Algorithm for a point query.

computed offset (Step Q1). For each hash table, the appropriate bucket i is selected using the corresponding hash function employed during ingestion. The value in the bucket i is then returned as an estimate of the count of the item. Among the d_h counts obtained, we return the *minimum* value as the final count estimate for the item q from the sketch (Step Q2). This approach differs from the CM sketch where the final result is computed as the minimum of d counts without using any offset or gap. The choice of d_h prevents estimates for high-frequency items from being under-estimated due to collisions with low-frequency items. Further, as for every ingested item, at least d_h hash tables are used, one would never get a zero count for an inserted item.

As results from Section VII demonstrate, the FCM adaptation consistently computes more accurate estimates than CM. In the context of parallel stream counting, a key advantage of the FCM algorithm is its ability to control the collisions between high- and low-frequency items in the sketch by changing the values of d_h and d_l. Such fine tuning is not possible with straightforward adaptations of the CM algorithm.

VI. Details of the Cell Implementation

Our implementation of the parallel counting algorithm uses the master-slave approach using the Cell's task-parallel programming model. This approach also used for programming network processors [5], [11]. Our implementation divides the work between the PPE- and SPE-based components. All SPEs executed the same code in the SPMD fashion. The sequential PPE code acts as the coordinating processors. It loads the SPE code into individual local stores and then invokes the sequential FCM kernels on one or more SPEs. The PPE code reads the stream data and partitions it using either block- or hash-based partitioning schemes. At the query processing time, the PPE propagates the query to the participating SPEs, collects the results from these SPEs and returns the final result.

Each participating SPE code executes the core FCM counting kernel. The SPE kernel first initializes key data structures, e.g., the FCM Sketch, and the data buffers in the local store. All locally allocated memory is 128-byte aligned. Once initialized, each SPE instance fetches its portion of the stream data via DMA from the main memory. As a single DMA can fetch only 16 KB of data at every invocation, the core ingestion algorithm (Figure 7:I1–I3) had to be *strip-mined* (the main loop partitioned into 2 nested loops). Further, the DMA memory accesses use the double-buffering approach (i.e., issue two asychronous DMA requests and work with the one that completes first) to hide the memory latencies. The SPE implementation of the FCM uses the native 128-bit SIMD instruction set to accelerate a variety of key computational functions. The FCM algorithm, for every item, computes $d+2$ hash values. Hence, the most critical function in the SPE code is the calculation of the hash value. Specifically the computations of the universal hash function (i.e., hash(x)=$(a.x + b)mod(P)$, where P is a prime number) was implemented using the SPE SIMD instrisics. Although, the input data was represented as 32-bit integers, we used 64-bit double-precision floating point representation while computing the hash values. The hash-value computation used SIMDized implementation of the double-precision modulo function. Since the SIMD instructions use 128-bit registers, we could compute hash-values of two items simultaneously.

```
#define MOD 2147483647
Vector hash31_spe(
      unsigned int a, unsigned int b,
      unsigned int x, unsigned int y,
      unsigned int number)
{
1 vector double result;
2 unsigned int lresult;
3 vector double aV,bV,xV,dV,num;
4 aV=(vector double) spu_splats((double)a);
5 bV=(vector double) spu_splats((double)b);
6 xV=(vector double) spu_insert((double)x, xV,
0);
7 xV=(vector double) spu_insert((double)y, xV,
1);
8 dV=(vector double) spu_splats((double)MOD);
9 num=(vector double) spu_splats((double)number);
10 result=(vector double)spu_madd(aV,xV,bV);
11 result=(vector double)_fmod_v(result,dV);
12 result=(vector double)_fmod_v(result,num);
13 return(result);
}
14 x_pos = (unsigned int) spu_extract(result,
0)
15 y_pos = (unsigned int) spu_extract(result,
1)
```

Fig. 8. A code fragment illustrating the SIMD implementation of $(((a.x + b)mod(P))mod(w))$, where w is the sketch width.

Figure 8 presents a code fragment that uses SIMD instructions to compute the hash values for two integer items x and y. To prevent overflow problems, we used double-precision numbers to store the results. For these items, the function `hash31_spe()` returns the corresponding positions in the

range $[1..number]$ as doubles. The function uses a variety of SPE SIMD intrinsic functions on a special 128-bit data type, `vector`. The `spu_splats()` intrinsic function populates all cells of a typed vector with the same specified value. For example, in line 4, the vector `aV` stores two 64-bit double precision numbers of value a. Lines 6 and 7 insert the items x and y into the vector xv at positions 0 and 1, respectively. Line 10 performs the multiply-and-add computation, $a.z+b$, on the two items x and y using 3 double vectors and using a single SPE intrinsic `spu_madd()` and stores result in the `result` vector. Lines 11 and 12 perform the `modulo` operations on the two vectors. Finally, line 13 returns result vector. Lines 14 and 15 extract the appropriate values and assign them to the corresponding integer variables. Note that the SIMD intrinsic functions, in every invocation, perform the same operation on more than one item (in this case, 2 doubles).

We also used the SIMD instructions to implement other operations in the ingestion phase. The Cell SPE has 2 execution pipes and can process two instructions simultaneously and improve the cycles per instruction count (CPI). However, to effectively exploit the dual-issue features, we had to extensively unroll the SPE code. During the querying phase, the PPE multicasted a point query to the participating SPEs. The SPEs accessed their local data structures to compute local counts, and sent them back to the PPE. Hence, our parallel counting algorithm did not require any inter-SPE communication during stream ingestion or querying.

While implementing the parallel algorithm on the Cell was relatively straight-forward, optimizing the per-SPE code was non-trivial. Our performance analysis showed that performance of the overall parallel algorithm was affected not by the DMA costs or by the cost of data partitioning on the PPE, but by the SPE computations. The SPE implementation of the FCM algorithm had two keys performance issues, both caused by the use of double-precision arithmatic. First issue was the performance of the hash function, specifically, the cost of the modulo function. We addressed the problem by using an optimized implementation of the modulo function which does not use the floating point division. The second was related to the cost of scalar convertion of a float to an integer (note that the FCM algorithm used 2 hash functions to compute the offset and gap into the FCM sketch. While the hash values are double-precision floats, the offset and gaps are integers.) This conversion causes excessive memory stalls that affects the overall performance. We addressed this issue by improving the dual-issue rate via instruction-level parallelism by further unrolling the inner loop (i.e., operations on the 16 KB data).

VII. Experimental Evaluation

We have evaluated the parallel counting algorithm on the Cell using two sketch-based counting kernels: FCM and CM. We evaluated the implementations using a stream of 4M integer items with Zipf ($\lambda = 1.1, 1.5$) and Normal ($\sigma = 10, 20$) distributions under the error bound $\epsilon = 0.087$, and the probability bound $\delta = 0.0002$. Based on these parameters,

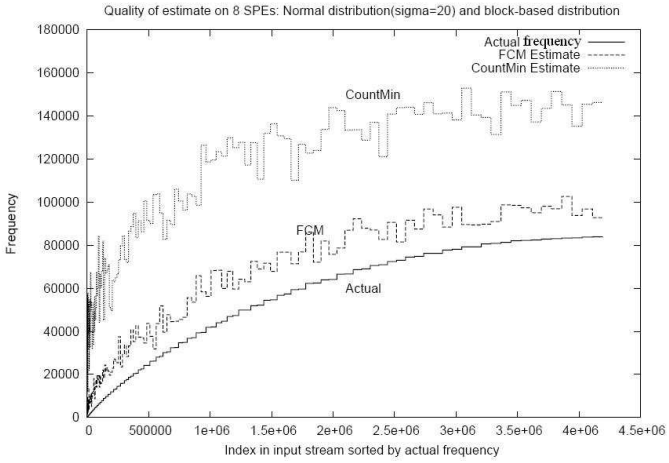

Fig. 9. Quality of estimating a Normal ($\sigma = 20$) input data using 8 SPEs via block-based data partitioning over 4M items.

for both the CM and FCM algorithms, we used a sketch with 17 hash functions (i.e., $d = 17$) with the range of 31 (i.e., $w = 31$). We used the same universal hash functions for updating the FCM and CM sketches. For these experiments, d_h was set to $\frac{d}{2}$ and d_l was set to $\frac{4}{5}d$.[2] The FCM algorithm also used a MG frequency counter with the stream window size $k = 8$ (corresponding number of entries in the MG frequency counter was $klogk = 24$). Note that the additional space required by the MG frequency counter was substantially less than the CM or FCM sketch. For the FCM algorithm, the per SPE memory consumption was around 110 KB (note that, depending on the size of the binary code, the SPE local store allows only upto 115 KB of user space), which included the space of two data buffers, the sketch, and the frequency counter, etc. The CM algorithm consumed slightly less SPE local space as it did not use the MG frequency counter.

We ran our experiments on a 2.1 GHz Cell-based blade with a single Cell Broadband Engine(BE) Processor, with a single PPE and 8 SPEs. We scaled the number of SPEs from 1 and 8, and partitioned the data using block-based partitioning with 16 KB blocks and hash-based partitioning over 8 processor groups (each processor group had a single member). For the hash-based partitioning, we used the last three bits of the item value as the function for partitioning the input data set into 8 buckets. Each bucket was then assigned to a SPE. For evaluation purposes, we use **normalized** error per unique items computed over a window of 1024 items.

Figures 9, 10, 11, and 12 illustrate the estimates for the Normal and Zipf datasets using the parallel algorithm over 8 SPEs. Each experiment is run using the CM and FCM algorithms with the block-based and hash-based data partitioning. As these graphs demonstrate, in most cases, the FCM algorithm substantially improves the estimation quality over the CM algorithm for the *entire frequency range*. The

[2]To a priori determine optimal d_h and d_l for an input data distribution is an open problem and is currently being investigated.

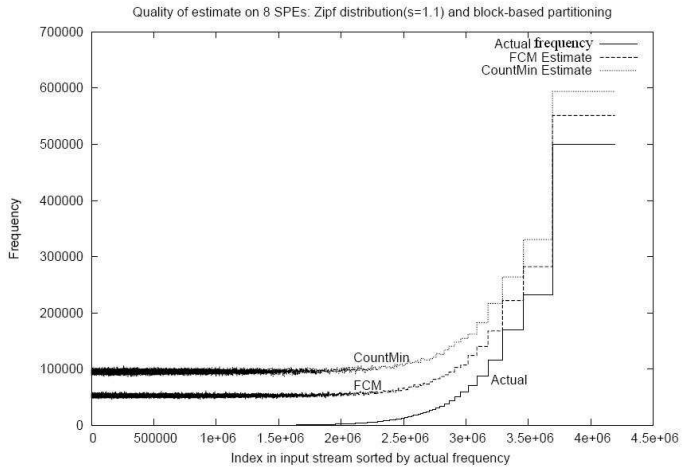

Fig. 10. Quality of estimating a Zipf ($\lambda = 1.1$) input data using 8 SPEs via block-based data partitioning over 4M items.

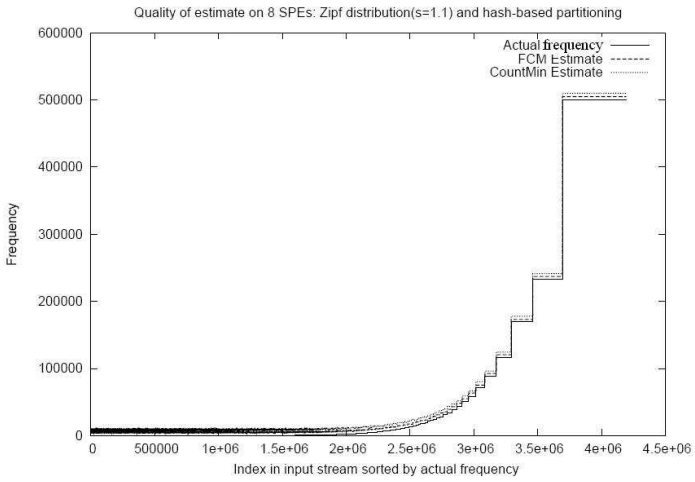

Fig. 11. Quality of estimating a Zipf ($\lambda = 1.1$) input data using 8 SPEs via hash-based data partitioning. The normalized estimation error was reduced from 2.13 (for CM) to 1.13 (for FCM).

exception being the low-tailed Normal ($\sigma = 10, 20$) datasets (Figure 12), where FCM and CM estimates are very accurate due to fewer collisions among low- and high-frequency items. A key point to note is that both CM and FCM algorithms are using the same space for storing their sketches. Therefore, the improvement is mainly due to FCM's better estimation of low-frequency items. The reduced number of collisions among high- and low-frequency items in the FCM algorithm also improves the estimates of high-frequency items.

Further, for both FCM and CM algorithms, the overall quality of estimation improves when the stream data is partitioned using the hash-based partitioning scheme. There are two reasons for this improvement. First, when the stream data is partitioned using a hash-based partitioning scheme, all occurrences of a particular value are grouped together. Thus the *relative* frequency of the items observed by a processor group increases. As the CM and FCM algorithms both provide

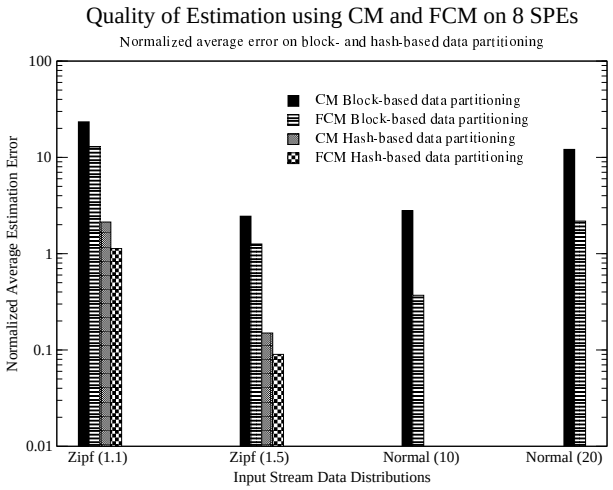

Fig. 12. Comparison of estimation quality for Zipf and Normal data distributions using block- and hash-based data partitioning. Note the logarithmic Y-axis. Errors in estimating the Normal datasets using hash-based data partitioning too small for representation.

good estimation of high-frequency items, the per-processor estimates improve substantially (This effect was particularly prominent for the Normal datasets as displayed in Figure 12). Second, as the data is partitioned only across a subset of processors, the error in the final estimate is bound by the number of processors in the processor group (in our experiment, one processor). The combination of improved local estimates and aggregation over a smaller number of processors leads to substantial improvement in the final estimate. Figures 13 and 14 compare the frequency estimation for the hash-based data partitioning while using the FCM algorithm on 1 SPE and 8 SPEs. In both cases, the frequency estimation improves substantially as the number of SPEs is increased to 8. These results further demonstrate the benefits of using hash-based data partitioning for our parallel algorithm.

To further examine the effect of hash-based partitioning on the estimation errors, we measured the normalized estimation error for both block- and hash-based partitioning for both Zipf and Normal datasets while scaling the number of SPEs from 1 to 8 (Figures 15 and 16). As Figure 15 illustrates, for the block-based partitioning, the normalized estimation error does not decrease as the number of SPEs is scaled up from 1 to 8. In fact, in one case, the error *increases* as the number of SPEs is increased. On the contrary, while using hash-based partitioning (Figure 16), the normalized error decreases significantly as the number of SPEs is increased (for the Normal dataset, after 2 SPEs, the error is too small to be represented on the graph.). Unlike the hash-based partitioning, the block-based partitioning does not group together items of the same value. Hence, the quality of local estimate is not as high as that for the high-frequency items (i.e., local errors are high). Also, the final result is calculated via aggregating local results over all SPEs in the system. Therefore, the local errors get accumulated, resulting in a significant degradation in estimation quality.

Finally, Figure 17 illustrates the scalability of the parallel

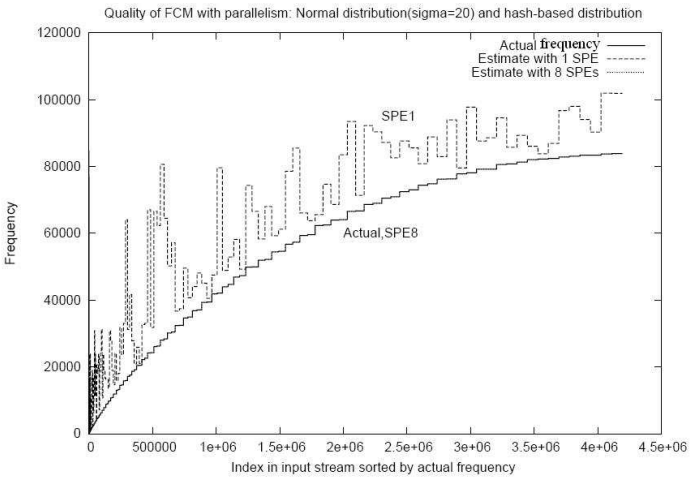

Fig. 13. Comparison of the quality of estimation of a Normal ($\sigma = 20$) data using hash-based data partitioning over 1 and 8 SPEs.

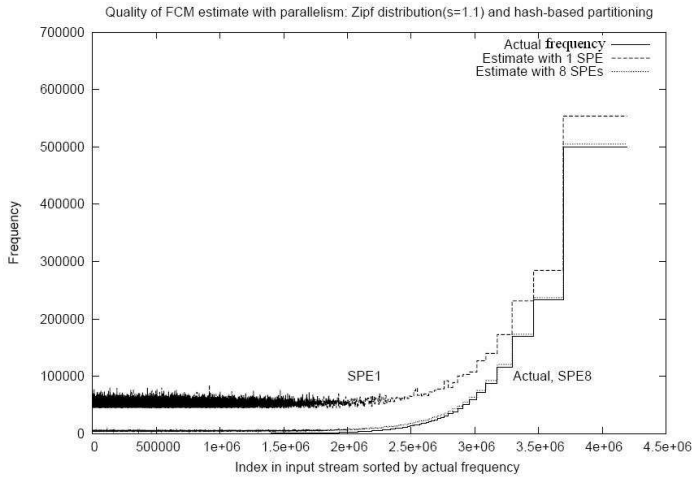

Fig. 14. Comparison of the quality of estimation of a Zipf ($\lambda = 1.1$) data using hash-based data partitioning over 1 and 8 SPEs.

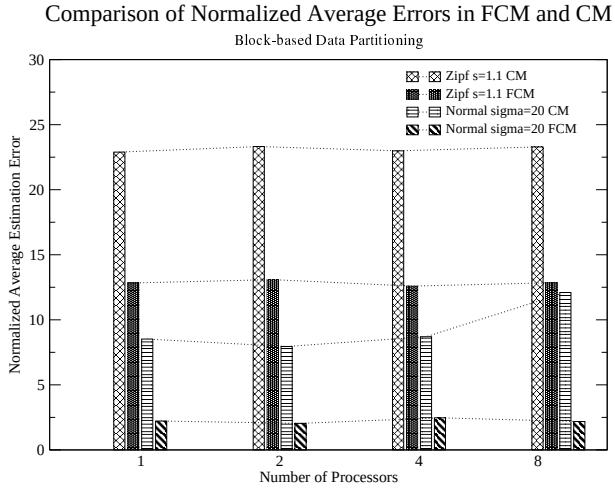

Fig. 15. Comparison of errors in FCM and CM using block-based data partitioning from Normal ($\sigma = 20$) and Zipf ($\lambda = 1.1$) datasets. The number of SPEs is increased from 1 to 8.

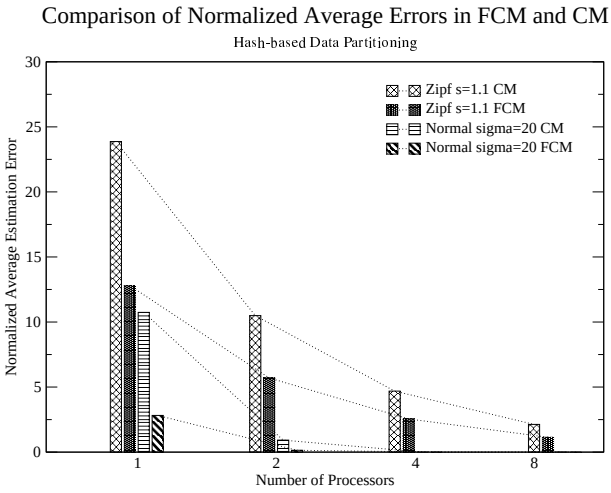

Fig. 16. Comparison of errors in FCM and CM using hash-based partitioning as the number of SPEs is increased from 1 to 8. Errors in estimating Normal ($\sigma = 20$) datasets on 4 and 8 SPEs are too small for representation.

algorithm under different runtime configurations. We use the execution time of the algorithm for ingesting 16 MB data stream on a single SPE as a baseline and compute the relative performance for 2, 4, and 8 SPEs. As shown in Figure 17, in all cases, the parallel algorithm demonstrates linear scalability as the number of SPEs was increased. The hash-based data partitioning has very similar performance to the block-based data partitioning. In both cases, the cost of processing a point query was insignificant. We also did not find any significant performance overheads due to the additional PPE-side processing or unbalanced data partitioning across the SPEs.

The results presented in this section conclusively demonstrate that FCM's key feature, namely, the frequency-aware sketch processing using the Misra-Gries Counter, improved the estimation quality over the CM algorithm for the *entire frequency range* under *similar space utilization* by reducing the collisions between high- and low-frequency items during stream ingestion. In addition, FCM's use of a certain subset (d_h, Figure 7) of hash tables for querying prevented the underestimation of high-frequency items. While our parallel algorithm achieved linear scalability under both hash- and block-based partitioning, these results also illustrate that simply partitioning data across multiple processors does not lead to an improvement in estimation quality as the number of processors is increased. To achieve performance scalability and estimation improvement, one needs to use hash-based data partitioning. Hash-based partitioning improved local count estimates and reduced the error in computing the final estimate.

It is important to note that even under hash-based partitioning, the FCM algorithm computed far better estimates then the CM algorithm. One possible reason is that we had to use very simple hash functions so as to avoid overloading of the PPE. In general, it is difficult to identify a hash function that is computationally simple and preserves the statistical characteristics of the original data even when the data is distributed. Our experiments with hash-based partitioning also used the

processor groups of size 1. When the number of processors is large (e.g., 64), the hash-based partitioning would use non-singular processor groups, and within each processor group, the data could be again partitioned in a block-based manner. In such cases, the issues related to collisions between high- and low-frequency become prevelant again. Hence, the FCM algorithm would be more effective than the CM algorithm even under hash-based data partitioning.

VIII. Related Work

The problem of synopsis construction has always been considered an important one in the context of data streams. A broad overview of synopsis construction methods in data streams may be found in [9].

The sketch based method was first proposed in [1] as an application of random projection techniques for space-efficient determination of aggregate characteristics of the stream such as frequency moments and frequent items. The broad idea of the original sketch technique is that a stream can be viewed as a very high dimensional vector whose dimensionality is determined by the number of items in the stream, and a random projection [12] of this stream can be used to determine aggregate functions of the frequencies of the individual items. A pseudo-random number generator is used to dynamically generate the components of the random projection vector rather than explicitly storing them [1]. The sketch based method is particularly useful for the space-constrained scenario (such as a cell processor) since its space-requirements are logarithmic in terms of the number of distinct items in the data stream. Subsequently a variety of other sketch based methods were proposed which vary from the random projection idea for sketch computation. These include hash-based structures such as that proposed in [2] (Count Sketch), and the count-min sketch [3]. Some the sketch based structures such as the count-min sketch [3], [4] are particularly suited to tracking frequent items in the streams, whereas others such as random projection based sketches [1] are more suited to aggregate computations such as determining the frequency moments. However, both sketch based structures are generally better suited to estimating the behavior of items with high frequency, and are not particularly well suited to lower frequency items or determining the number of distinct elements.

The MG counter approach is similar to the Sticky Sampling and Lossy Counting algorithms proposed by Manku and Motwani [14]. However, their aim was to detect items whose true frequencies *exceed* a certain threshold. In contrast, we are interested in determining the current *relative* frequency phase of an item. Further, the space complexity of the Manku-Motwani algorithms is worse than that of the MG counter. Estan and Varghese [7] propose a counter-based scheme to detect top-k heavy hitters (*elephants*) based on a threshold value. The Estan and Varghese algorithm neglects low-frequency items (*mice*); FCM's key goal is to detect and correctly estimate low-frequency items. They uses a conservative update method to increment multiple counters to avoid false positions and negatives. This approach is not applicable when a single

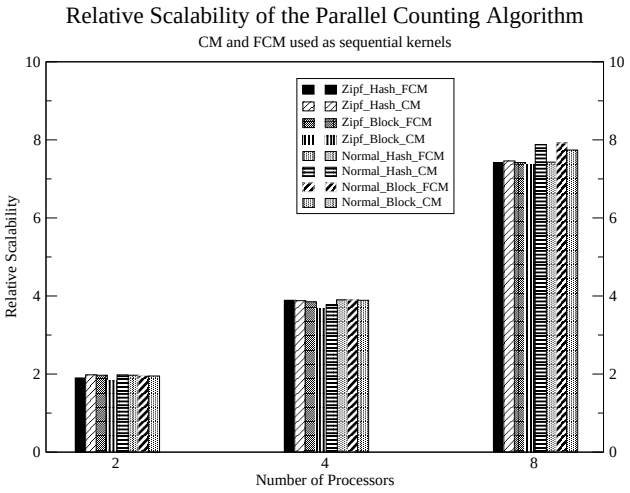

Fig. 17. Linear scalability of the Parallel Counting Algorithm as compared to the single SPE execution time for ingesting data stream.

counter is used (as in the case of the FCM algorithm). The space complexity of their scheme is also significantly larger. The *Stream-Summary* data structure proposed by Metwally et al [16] uses steps similar to those used for updating the MG counter. However, its space complexity is a function of user-defined error ϵ and the range of the input data, which is significantly larger than our MG counter. The Count Sketch [2] algorithm first proposed the idea of varying the number of counters updated per item as a way for reducing collisions between high- and low-frequency items. Unlike our proposal, this idea does not use an item's current frequency state (i.e., high- or low-frequency) to determine the number counters to be updated.

IX. Conclusions and Future Work

In this paper, we investigated the problem of counting items from data streams using the modern heterogeneous multi-core Cell processor. We proposed a new sketch-based counting algorithm, FCM, which improves over competitive algorithms by improving estimates of low-frequency items using similar space utilization. FCM updates the sketch data structure using different number of hash tables for low- and high-frequency items. We used the FCM algorithm as the kernel in our implementation of a parallel counting algorithm on the Cell processor. The experimental results demonstrate that in the FCM kernel provides better quality estimates over the CM algorithm both for the high- and low-frequency items. Further, our parallel algorithm improves both performance and quality as the number of processors is scaled up. Although the FCM algorithm is evaluated using the Cell processor, it can be implemented on similar multi-core processors [5], [11] or using software libraries like Pthreads.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *ACM Symposium on Theory of Computing*, pages 20–29, 1996.

[2] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002*, pages 693–703, 2002.

[3] G. Cormode and M. Muthukrishnan. An Improved Data Stream Summary: the Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1), April 2005.

[4] G. Cormode and S. Muthukrishnan. What's not and what's not: Tracking Most Frequent Items Dynamically. *ACM Transactions on Database Systems*, 30(1):249–278, March 2005.

[5] Intel Corporation. Intel IXP Network Processors. Online Documentation, www.intel.com, 2000.

[6] I.B.M. Corporation. Cell Processor Technical Documentation. IBM Developerworks, 2006.

[7] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.

[8] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1982.

[9] M. Garofalakis and J. Gehrke. Querying and Mining Data Streams: You only get one look. In *VLDB Conference*, 2002.

[10] P. B. Gibbons and Y. Matias. Synopsis Data Structures for Massive Data Sets. In *Proceedings of Symposium on Discrete Algorithms*, 1999.

[11] Cisco Systems, Inc. Parallel Express Forwarding on the Cisco 10000 Series. White Paper, 2006.

[12] W. Johnson and J. Lindenstrauss. Extensions of Lipshitz Mapping into Hilbert Space. *Comtemporary Mathematics*, 26:189–206, 1984.

[13] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Interconnection Network: Built for speed. *IEEE Micro*, 26(3), 2006.

[14] G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proceedings of the 28th VLDB Conference*, pages 346–357, 2002.

[15] Mercury Computer Systems, Inc. *Cell Broadband Engine Processor Products*, 2006.

[16] A. Metwally, D. Agrawal, and A. E. Abbadi. An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams. *ACM Transactions on Database Systems*, 31(3):1095–1133, 2006.

[17] J. Misra and D. Gries. Finding Repeated Elements. *Science of Computer Programming*, 2:143–152, 1982.

[18] S. Muthukrishnan. Data streams: Algorithms and Applications. Technical report, Computer Sciences Department, Rutgers University, 2005.

[19] K.-L. Wu, P. S. Yu, B. Gedik, K. Hildrum, C. C. Aggarwal, E. Bouillet, W. Fan, D. George, X. Gu, G. Luo, and H. Wang. Challenges and Experience in Modeling and Prototyping a Multi-Modal Stream Prototyping Application on System S. In *VLDB Conference*, pages 1185–1196, 2007.