A Case Study of OpenMP applied to Map/Reduce-style Computations

Keywords: OpenMP, map/reduce, reduction

As data analytics are growing in importance they are also quickly becoming one of the dominant application domains that require parallel processing. This paper investigates the applicability of OpenMP, the dominant shared-memory parallel programming model in high-performance computing, to the domain of data analytics. We contrast the performance and programmability of key data analytics benchmarks against Phoenix++, a state-of-the-art shared memory map/reduce programming system. Our study shows that OpenMP outperforms the Phoenix++ system by a large margin for several benchmarks. In other cases, however, the programming model is lacking support for this application domain.

1 Introduction

Data analytics (a.k.a. \Big Data") are increasing in importance as a means for business to improve their value proposition or to improve the e ciency of their operations. As a consequence of the sheer volume of data, data analytics are heavily dependent on parallel computing technology to complete data processing in a timely manner.

Numerous specialized programming models and runtime systems have been developed to support data analytics. Hadoop [2] and SPARK [22] implement the map/reduce model [6]. GraphLab [10], Giraph [1] and GraphX [20] implement the Pregel model [12]. Storm [3] supports streaming data. Each of these systems provides a paral-lel and distributed computing environment built up from scratch using threads and bare bones synchronization mechanisms. In contrast, the high-performance computing community designed programming models that simplify the development of systems like the ones cited above and that provide a good balance between performance and programming e ort. It is fair to ask if anything important was overseen during this decades-long research that precluded the use of these parallel programming languages in the construction of these data analytics frameworks.

This paper addresses the question whether HPC-oriented parallel programming models are viable in the data analytics domain. In particular, our study contrasts the performance and programmability of OpenMP [14] against Phoenix++ [17], a purpose-built shared-memory map/reduce runtime. The importance of these shared-memory programming models in the domain of data-analytics increases with the emergrence of in-memory data analytics architectures such as NumaQ [7]. To program against Phoenix++, the programmer needs to specify several key functions, i.e., the map, combine and reduce functions, and also select several container types used internally by the runtime. We have found that the programmer needs to understand the internals of Phoenix++ quite well in order to select the appropriate internal containers. Moreover, we conjecture that the overall tuning and programming e ort is such that the programming e ort is not much reduced in comparison to using a programming model like OpenMP.

We evaluate the performance and programmability of OpenMP for data analytics by implementing a number of commonly occurring map/reduce kernels in OpenMP. Experimental performance evaluation demonstrates that OpenMP can easily outperform Phoenix++ implementations of these kernels. The highest speedup observed was around 75% on 16 threads. We furthermore report on the complexity of writing these codes in OpenMP and the issues we have observed. One of the key programmability issues we encountered is the lack of support for user-de ned reductions in current compilers. Moreover, the OpenMP standard does not support parallel execution of the reduction operation, a feature that proves useful in this domain. This drives us to design the program and its data structures around an e cient way to perform the reduction.

In the remainder of this paper we will rst discuss related work (Section 2). Then we discuss the map/reduce programming model and the Phoenix++ implementation for shared memory systems (Section 3). We subsequently discuss the implementation of a number of map/reduce kernels in OpenMP (Section 4). Experimental evaluation

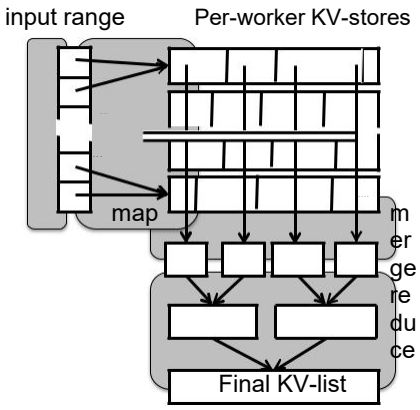

Figure 1: Schematic overview of Phoenix++ runtime system

demonstrates the performance bene ts that OpenMP bring (Section 5). We conclude the paper with summary remarks and pointers for future work (Section 6).

2 Related Work

Phoenix is a shared-memory map-reduce runtime system. Since its inception [16] it has been optimized for the Sun Niagara architecture [21] and subsequently reimplemented to avoid ine ciencies of having only key-value pairs available as a data representation [17].

Several studies have improved the scalability of Phoenix. TiledMR [4] improves memory locality by applying a blocking optimizing. Mao et al [13] stress the importance of huge page support and multi-core-aware memory allocators. Others have optimized the map/reduce for accelerators. Lu et al [11] optimize map-reduce for the Xeon Phi and attempt to apply vectorization in the map task and the computation of hash table indices. De Kruijf et al [9] and Ra que et al [15] optimize the map/reduce model for the Cell B.E. architecture.

While the map-reduce model is conceptually simple, a subtly unde ned aspect of map-reduce is the commutativity of reductions [19]. This aspect of the programming model is most often not documented, for instance in the Phoenix systems [16, 21, 17]. However, executing non-commutative reduction operations on a runtime system that assumes commutativity can lead to real program bugs [5] even in extensively tested programs [19]. OpenMP assumes reductions are commutative [14].

There has been e ort to use OpenMP style semantics for programming data-analytics and cloud-based applica-tions. OpenMR [18] implements OpenMP semantics on top of map-reduce runtime for cloud-based implementation. The motivation is to port OpenMP applications to the cloud as well as reduce the programming e ort. Jiang et al [8] introduce OpenMP annotations to a domain-speci c language for data-analytics, R, to facilitate the semi-automatic parallelization of R and thus reduce the parallel programming e ort.

3 Map-Reduce Programming Model

The map-reduce programming model is centered around the representation of data by key-value pairs. For instance, the links between internet sites may be represented by key-value pairs where the key is a source URL and the value is a list of target URLs. The data representation exposes high degrees of parallelism, as individual key-value pairs may be operated on independently.

Computations on key-value pairs consist, in essence, of a map function and a reduce function. The map function transforms a single input data item (typically a key-value pair) to a list of key-value pairs (which is possibly empty). The reduce function combines all values occurring for each key. Many computations t this model [6], or can be adjusted to t this model.

3.1 Phoenix++ Implementation

The Phoenix++ shared-memory map-reduce programming model consists of multiple steps: partition, map-and-combine, reduce, sort and merge (Figure 1). The partition step partitions the input data in chunks such that each

```
 1       int i ;
 2  #pragma omp parallel for
 3       for ( i =0; i < N; ++i ) f map(i); g
4
5
 6struct item t item; -
 7 #pragma omp parallel
 8 #pragma omp single
 9while( partition ( &item ) ) f
10  #pragma omp task
11            map(&item);
12       g
```

Figure 2: Generic OpenMP code structures for the map phase.

map task can operate on a single chunk. The input data may be a list of key-value pairs read from disk, but it may also be other data such as a set of HTML documents. The map-and-combine step further breaks the chunk of data apart and transforms it to a list of key-value pairs. The map function may apply a combine function, which performs an initial reduction step of the data. It has been observed that making an initial reduction is extremely important for performance as it reduces the intermediate data set size [17].

It is key to performance to store the intermediate key-value list in an appropriate format. A naive implemen-tation would hold these simply as lists. However, it is much more e cient to tune these to the application [17]. For instance, in the word count application the key is a string and the value is a count. As such, one should use a hash-map indexed by the key. In the histogram application, a xed-size histogram is computed. As such, the key is an integer lying in a xed range. In this case, the intermediate key-value list should be stored as an array of integers. For this reason, we say the map-and-combine step produces key-value data structures, rather than lists.

The output of the map-and-combine step is a set of key-value data structures, one for each worker thread. Let KV-list j = 0; : : : ; N 1 represent the key-value data structure for the j-th worker thread. These N key-value data structures are subsequently partitioned in M chunks such that each chunk with index i = 0; : : : ; M 1 in the intermediate key-value list j holds the same range of keys. All chunks i are then handed to worker thread N, which reduces those chunks by key. This way, the reduce step produces M key-value lists, each with distinct keys.

Finally, the resulting key-value lists are sorted by key (an optional step) and they are subsequently merged into a single key-value list.

Phoenix++ allows the programmer to specify a map function, the intermediate key-value data structure, a combine function for that data structure, the reduce function, a sort comparison function and a ag whether sorting is required.

3.2 OpenMP Facilities for Map/Reduce-Style Computations

Map/reduce, viewed as parallel pattern, is fairly easy to grasp and encode in a variety of parallel programming languages. OpenMP o ers multiple constructs to encode the map phase using parallel loops as illustrated in Figure 2. A parallel for loop applies when a large data set can be partitioned by considering the iteration domain of a for loop. Alternatively, if the partitioning requires a more complex evaluation, then task spawn construct inside a for loop may be more appropriate. An example encountered in our study is word count. Although the le contents are stored in an array, the boundaries of the partitions must be aligned with word boundaries, which is most easily achieved using the task construct.

The most recent OpenMP 4.0 [14] standard introduced support for user-de ned reductions (UDRs), which allows to specify reductions of variables of a wide range of data types with little programming e ort. Unfortunately, few OpenMP compilers currently fully support user-de ned reductions. This strongly limits the programmability aspect of this study, although we can expect this situation to improve with the availability of user-de ned reductions. Hence the implementation and performance of the reduce phase in OpenMP depends on the data type of the reduction object. More importantly, complex OpenMP 4.0 UDRs may not be evaluated in parallel, a feature that is important for reductions on collections, which are common in data analytics workloads. For example, if each thread produces a same-sized array which must then be reduced element-wise, then UDRs allow to specify this but the execution of the reduction will be sequential. The fast way to reduce a set of arrays is, however, by assigning each section of the arrays to a thread and have all threads reduce their section in parallel. Reductions on more complex data structures such as hash tables are even harder to parallelise, even with UDR support, whereas a sequential approach results

in poor performance.

4 OpenMP Implementations

We have ported seven map/reduce benchmarks from the Phoenix++ system to OpenMP. We describe the main characteristics of these benchmarks and the main issues encountered in porting them.

4.1 Histogram

The histogram benchmark processes a bitmap image to compute the frequency counts of values (in the range of 0-255) for each of its RGB components. The map phase is parallelized using the OpenMP for work-sharing construct. Each thread is statically assigned a subset of the pixels in the image and computes a histogram over this subset. These per-thread results are then reduced to compute the histogram of the whole image. However, due to lack of OpenMP support for user-de ned reductions (UDR) in our compiler, we had to nd ways to reduce the results without using locks or critical sections (which incur signi cant execution time overhead). We de ned a shared array as large as the histogram array times the number of threads i.e. for a 24-bit image, (256 3) #threads bytes. During the map phase, each thread stores its results to the array assigned to it based on its thread id. Once the map phase is completed, the results are reduced in a second OpenMP for loop where each thread reduces a section of the histogram. E.g., for 16 threads, each thread reduces a slice of 16 3 values.

4.2 Linear Regression

Linear Regression computes the values a and b to de ne a line y= ax+b that best ts an input set of coordinates. Firstly, ve statistics are calculated (such as sum of squares) on the input coordinates. We have used the parallel for construct to distribute the work among the threads. The per-thread statistics are reduced using the reduction clause. Secondly, a and b are computed using the ve statistics collected in the rst step.

4.3 K-Means Clustering

This benchmark implements a clustering algorithm which groups input data points in k clusters. The assignment of a data point to a cluster is made based on its minimum distance to the cluster mean. The assignment algorithm is invoked iteratively until it converges, i.e., no further changes are made to the cluster assignment. As long as the assignment algorithm has not converged, the cluster means are also recalculated iteratively.

Both the assignment and mean calculation steps have been separately parallelized with the parallel for construct.

4.4 Word Count

The word count benchmark counts the frequency of occurrence of each word in a text le. This is a stereo-typical example of a map/reduce type benchmark. For the map phase, we have used OpenMP tasks. A team of threads is rst created with the OpenMP parallel construct. Then one of the threads is designated to iteratively calculate the input partitions and spawn the tasks for the other threads to work on. Each thread completes its word counting task for the assigned partition, and then becomes available to operate on another partition.

Here again we faced di culty due to the absence of UDR support. We thus de ned a vector of hash tables and each thread stored its results in separate hash tables. After all the threads have nished working, the results are sequentially reduced in a global hash table. Parallelizing this reduction in a similar way as histogram is challenging, due to the di culty of isolating slices in each of the hash tables that hold corresponding ranges of keys. Although it is not impossible to solve this issue, it clearly impacts the programmability of OpenMP for workloads like these.

4.5 String Match

String match takes as input a set of encrypted keys and a text le. The text le is then processed to see which set of words were originally encrypted to produce the encrypted keys. This benchmark is parallelized using OpenMP tasks (Figure 3). A single thread, from a team of threads, partitions the input le on word boundaries. It spawns a task to handle each partition independently. A reduction phase is not required for this benchmark.

```
1 int    splitter_pos  = 0;
2  #pragma omp parallel
3 f
4#pragma omp single
5       f
6             while( 1 ) f
7                   str_map_data_t  partition ;
8                   /  End of data reached. /
9                    if ( splitter_pos  >= keys_le_len )_
10                        break;
11                  /  Determine the nominal end point .  /
12                  int end = std::min( splitter_pos  + chunk_size,          keys_ le_len ) ;
13                  /  Move end point to next word break   /
14                  while(end < keysle_len_&& keys        _ le [end] != 'nn')
15                        end++;
16                  /  Set the start  of the next data.  /
17                   partition . keys = keys  le + splitter_pos ;
18                  partition . keys_len = end        splitter_pos ;
19                  /  Skip line  breaks (code skipped for  brevity ) .  /
20                   splitter_pos  = end;
21
22                  /  Spawn a task to do the real work  /
23                  #pragma omp task  rstprivate( partition )
24                  f
25                        /  Apply sequential algorithm on data /
26                  g
27            g/  end of while (1) /
28      g
29 g
```

Figure 3: OpenMP code for String Match

4.6 Matrix Multiply

It computes a matrix C which is a product of two input matrices A and B. We have parallelized a simple matrix multiplication algorithm with the parallel for construct and the collapse clause to increase the available parallelism. Each thread calculates a subset of elements C(i,j). Moreover, we swapped the order of the two inner loops to improve the data locality.

4.7 Principal Component Analysis

This benchmark implements two stages of the statistical Principal Component Analysis algorithm. It takes as input a matrix which is a collection of column vectors. In the rst stage, per-coordinate means are calculated along the rows and work is distributed among the threads with the loop scheduler. In the second stage, the covariance matrix is calculated along with a total sum of co-variance. This loop nest is parallelized using the parallel for loop with a reduction clause for the scalar sum of co-variance. The second loop nest exhibits load imbalance which we mitigated by changing the granularity of static loop scheduler.

5 Evaluation

We evaluated the OpenMP and Phoenix++ version 1.0 programs on a dual-socket Intel Xeon E5-2650 with 8 cores per socket and hyperthreading. The operating system is CentOS 7.0 and we use the Intel C/C++ compiler v. 14.0.0. We evaluate the programs on the small, medium and large data sets supplied with Phoenix++. We pin threads to cores to ensure at most one out of each pair of hyperthreads is used.

5.1 Analysis

Figures 4 and 5 show the speedup curves for the OpenMP and Phoenix++ implementations of the 7 map/reduce workloads for 3 inputs with di erent sizes. Speedups are normalized to the execution time of a purely sequential code.

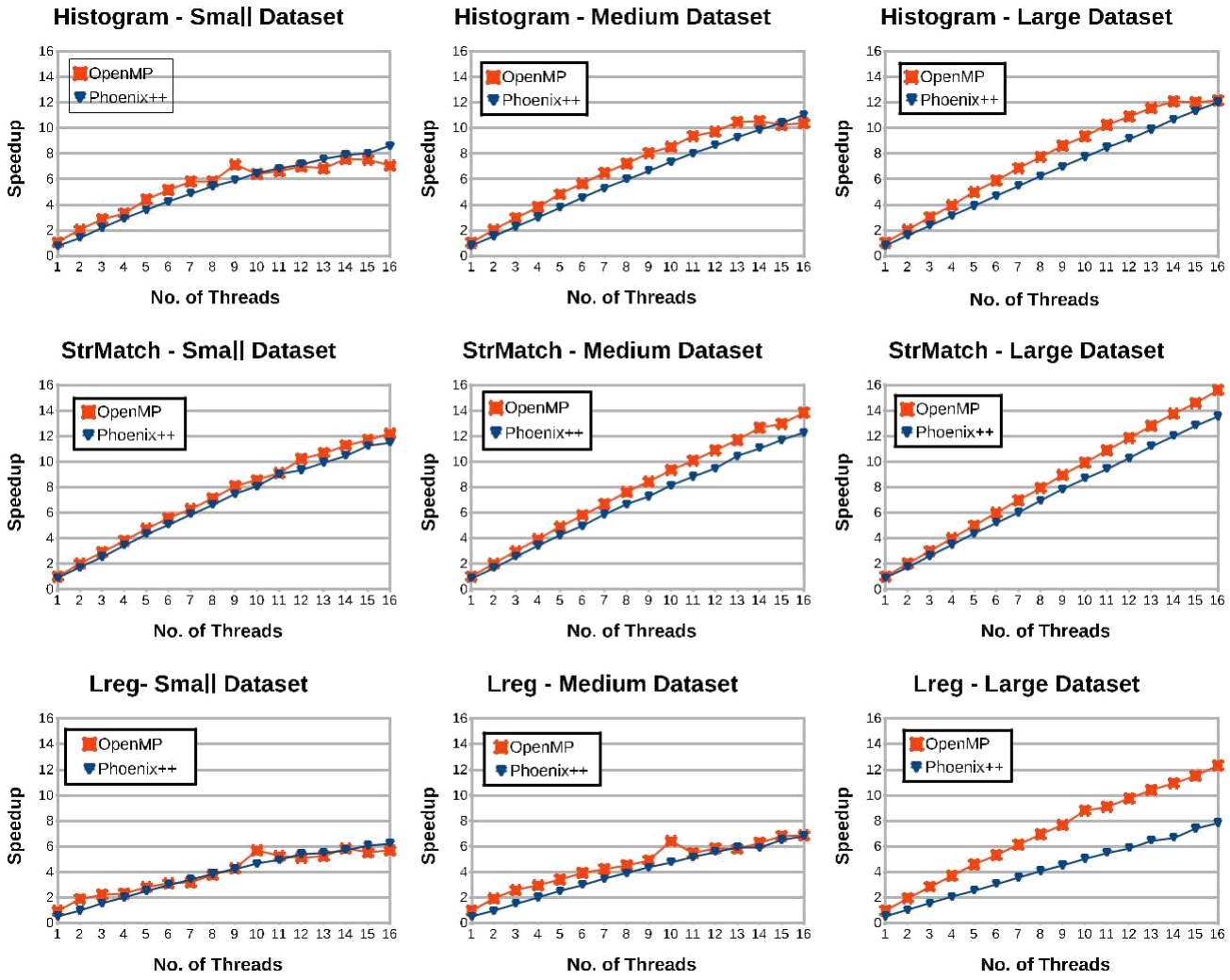

Figure 4: Speedup obtained with the OpenMP implementations of the benchmarks in comparison against the Phoenix++ implementations. Benchmarks with low computational intensity.

Figure 4 shows the performance of benchmarks with low computational intensity, i.e., they perform few opera-tions per byte transferred from memory. The OpenMP implementation of histogram performs similar to Phoenix++. For string match, OpenMP is again similar to Phoenix++ except on the large input where the OpenMP code gains 15% advantage. For linear regression, the Phoenix++ code scales to an 8-fold speedup at best, while the OpenMP code gains up to 12x. This is possible due to the higher e ciency of the OpenMP code, which does not use gener-alized data structures to mimic the emission of key-value pairs in the map task, or mimic a reduction of key-value pairs.

Two benchmarks with high computational intensity, namely kmeans and pca, perform markedly better with OpenMP than with Phoenix++ (Figure 5). In the case of k-means this is due to a memory allocator issue in Phoenix++, which can be solved by substituting for a better multi-core-aware memory allocator. PCA has load imbalance in the iterations of its outer loop. The Phoenix++ runtime cannot deal with this by itself and also o ers no controls to the programmer. In contrast, the OpenMP API allows us to x the load imbalance through setting the granularity of tasks for the static loop scheduler, which results in a 75% speedup.

Phoenix++ obtains excellent scalability on matrix multiply. While we did not obtain good speedups in our implementation of matrix multiply, we assume this can be xed with su cient locality optimization. Note however that the Phoenix++ implementation is quite straightforward and does not exhibit any speci c locality optimization.

Finally, word count shows bad scalability when implemented in OpenMP. While the map phase is trivially parallel using task parallelism, word count requires a reduction of the per-thread hash tables holding word counts. Parallelizing that reduction is hard but is key to obtaining good performance.

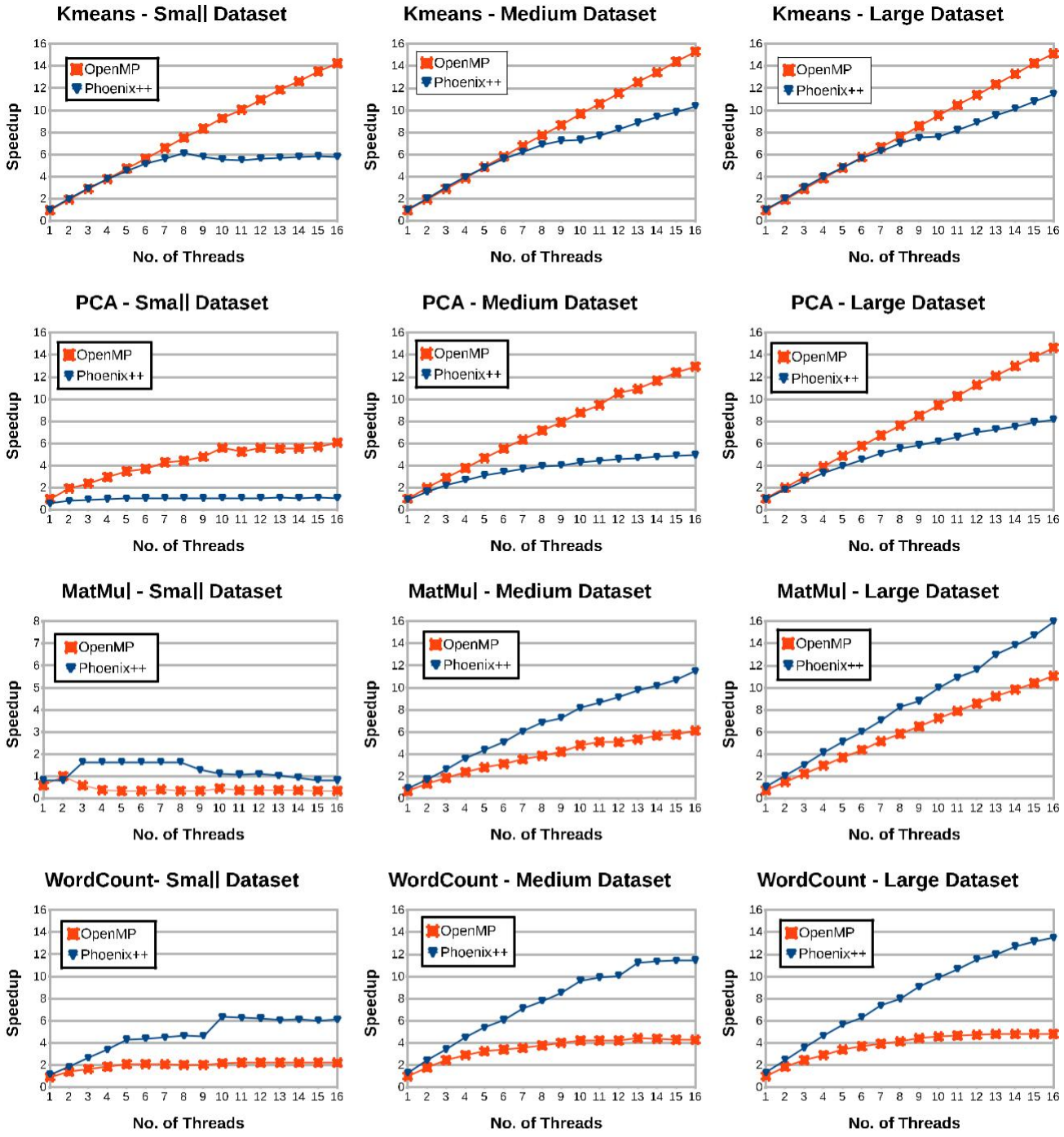

Figure 5: Speedup obtained with the OpenMP implementations of the benchmarks in comparison against the Phoenix++ implementations. Benchmarks with high computational intensity.

5.2 Coding Style Comparison

We present a comparsion of histogram benchmark implementation in both OpenMP and Phoenix++ (Table 1) to understand the programming e ort required. Phoenix++ library speci es default map, reduce and split function along with di erent options for containers and combiners. In most of the cases, the programmers will need to override the default map and split function to specify their own map function and distribute the work accordingly. Histogram code overrides the default map function to emit the distinct range of keys (line 6-10) for R, G and B components of the image . The selection of container depends on the cardinalty of keys. The histogram implementation below

uses an array container (line 2) since the number of distinct keys (or cardinality) is xed, in this case to 768. The de nition of MapReduce class also depends on whether the the keys need to be sorted in a particular order or not. From di erent combiners provided with Phoenix++ library, histogram uses the sum combiner since the values for a particular key need to be simply added.

In case of OpenMP, the choice of container to store historgam depends on how the reduction is to be performed. Since there are xed number of keys within a known range, a global array of the size of histogram (i.e., 768) times the number of available threads is de ned. OpenMP for construct (line 12) parallelizes the map phase by distributing the iterations of the loop among available worker threads. Each thread then updates the respective histogram buckets based on its id (line 14-17). Due to absence of UDR support for non-scalars, programmer needs to write how the reduction is to be performed on the array. For this benchmark, a separate global array histo has been used to store the results of reduction performed on histo shared array (line 22-25).

Table 1: Coding Style Comparison for histogram benchmark implementation in Phoenix++ (left) and OpenMP(right)

```
1 /       de ning a MapReduce class with a sum combiner and
          default sorting order by keys /
2  class HistogramMR : public MapReduceSort<HistogramMR,
          pixel, intptr t, uint64 t, array container <intptr t ,
          uint64 t , sum combiner,768>>
3  f
4   public :
5       /  overriding default map function /
6         void map(data type const& pix, map container& out) const
                f
7             emit intermediate (out,    pix .b, 1);
8             emit intermediate (out,    pix .g+256, 1);
9             emit intermediate (out,    pix . r+512, 1);
10       g
11          /  default reduce function from library  to be used /
12 g;
13
14 / inside main() function /
15
16  std :: vector<HistogramMR::keyval> result;
17  HistogramMR mapReduce = new HistogramMR();
18
19 /       calling map reduce on input image with image data bytes in
          bitmap[] /
20 mapReduce   >run((pixel )bitmap, num pixels, result);
```

```
1  int num of threads;
2  uint64 t  histo shared ;
3 #pragma omp parallel
4  f
5        / map phase / 6
#pragma omp single
7       f
8             num of threads= omp get num threads();
9             histo shared = new uint64 t [768 num of threads];
10       g
11        int id = omp get thread num();
12 #pragma omp for
13       for ( long i =0; i < num pixels; i ++ ) f
14            pixel  pix = ( pixel  ) &(bitmap[3 i]) ;
15            histo shared [ id 768+ (size t) pix  >b]++;
16            histo shared [ id 768+(256+((size t) pix      >g))]++;
17            histo shared [ id 768+(512+((size t) pix      >r))]++;
18       g
19
20       /  reduce phase /
21 #pragma omp for
22       for ( int j=0; j< 768; j++)
23            for ( int k=0; k<num of threads; k++)
24                 histo [ j]+=histo shared[ j+k 768];
25 g
26 delete []     histo shared ;
```

5.3 Implications to OpenMP

The implementation of most of the benchmarks was straightforward and lead to the results shown fairly easily. Obtaining excellent performance was easy especially in those cases where the reduction variable consisted of a small number of scalars. Whenever the reduction variable became more complex (e.g., an array in histogram or a hash table in word count), much of the programming e ort became focused on how to e ciently perform the reduction, which required parallel execution of the combine step of the reduction. The Intel compiler we have used currently does not support user de ned reductions (UDR). We expect that UDR support will simplify the programming e ort substantially. However, it is unlikely that UDRs will deliver su cient performance as the OpenMP speci cation does not allow parallel execution of a reduction, e.g., OpenMP pragma's within combine functions are disallowed [14]. This is a potential area of improvement for OpenMP.

The matrix multiplication problem demonstrates that OpenMP may require substantially higher e ort than Phoenix++ to tune the performance of an application. Even though it is evident what parallelism is present in matrix multiply, exploiting this in OpenMP requires signi cant e ort, while a straightforward implementation in Phoenix++ gives fairly good results.

6 Conclusion

This paper has evaluated the performance and programmability of OpenMP when applied to data analytics, an increasingly important computing domain. Our experience with applying OpenMP to map/reduce workloads shows that the programming e ort can be quite high, especially in relation to making the evaluation of the reduction step e cient. For most benchmarks, however, OpenMP outperforms Phoenix++, a state-of-the-art shared-memory map/reduce runtime.

To simplify the programming of these workloads, OpenMP will need to support much more powerful reduction types and support parallel execution of the reduction. User-de ned reductions, currently unavailable to us, promise ease of programming but parallel execution of reductions is not supported.

7 Acknowledgment

This work is supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the ASAP project, grant agreement no. 619706, and by the United Kingdom EPSRC under grant agreement EP/L027402/1.