A Study of SpMV Implementation Using MPI and OpenMP on Intel Many-Core Architecture

Fan Ye[1,2(✉)], Christophe Calvin[1], and Serge G. Petiton[2,3]

[1] CEA/DEN/DANS/DM2S, CEA Saclay, 91191 Gif-sur-Yvette Cedex, France
[2] Maison de la Simulation, USR3441,
Digiteo Labs Bât 565, 91191 Gif-sur-Yvette, France
fan.ye@cea.fr
[3] Laboratoire d'Informatique Fondamentale de Lille,
Université des Sciences et Technologies de Lille, 59650 Villeneuve d'Ascq, France

Abstract. The Sparse Matrix-Vector Multiplication (SpMV) is fundamental to a broad spectrum of scientific and engineering applications, such as many iterative numerical methods. The widely used Compressed Sparse Row (CSR) sparse matrix storage format was chosen to carry on this study for sustainability and reusability reasons.

We parallelized for Intel Many Integrated Core (MIC) architecture a vectorized SpMV kernel using MPI and OpenMP, both pure and hybrid versions of them. In comparison to pure models and vendor-supplied BLAS libraries across different mainstream architectures (CPU, GPU), the hybrid model exhibits a substantial improvement.

To further assess the behavior of hybrid model, we attribute the inadequacy of performances to vectorization rate, irregularity of non-zeros, and load balancing issue. A mathematical relationship between the first two factors and the performance is then proposed based on the experimental data.

1 Introduction

The SpMV is vital to scientific and engineering applications. It is the essential operation of many iterative linear and eigen solvers such as Conjugate Gradient (CG) and Generalized Minimum Residual (GMRES). In this paper, we take Intel Xeon Phi coprocessor as the underlying system for revealing some idiosyncrasies in an efficient SpMV implementation. A simplified way to view this many-core architecture is a chip-level SMP which offers remarkably high bandwidth. The prototype C0 codenamed Knights Corner (KNC) has 61 cores, each featuring a 512-bit wide vector unit and being capable of running up to 4 HW threads. These factors enable such single chip to yield over 1 TFlops double precision peak performance.

Due to sparse matrices' nature of irregularity, the memory subsystem often appears as the main bottleneck of SpMV's efficiency in terms of FLOPS (FLoating-point Operations Per Second). Furthermore, in a shared memory context with a large count of cores such as MIC, the scalability behavior is

M. Daydé et al. (Eds.): VECPAR 2014, LNCS 8969, pp. 43–56, 2015.
DOI: 10.1007/978-3-319-17353-5_4

not obvious which may depend on issues like data locality and access pattern. A common approach to address these problems is to propose a new sparse matrix storage format [10]. However, some certain techniques used in new formats may become less pertinent as the targeting architecture evolves. They may need to be adapted accordingly. Another potential downside of a new format is that it is hard to implement it in a large numerical package such as PETSc [1] or Trilinos [7], and thus be not easy to integrate or interface in large scientific applications. Both PETSc and Trilinos adopt the CSR (Compressed Row Storage) as the underlying sparse format. We want to study the SpMV kernel within the context of linear or eigen solvers, making the availability of these numerical packages prominent to us. As a result, we chose to use CSR format.

The preceding studies [4,5] hold pessimistic views of hybrid fashion compared to a unified MPI approach. The related literatures usually underline the importance of network performance in explaining the gap between different models. Therefore the high on-chip bandwidth of MIC drives us to investigate the potential benefit of using hybrid programming. We refer the hybrid execution here to a scenario where the coprocessor resources (cores and caches) are divided into several separate domains and each domain is governed by one MPI process and shared by a number of OpenMP threads.

To set an architectural baseline, we also perform the tests over the same matrix suite on dual Intel Sandy-Bridge octa-core processors, as well as the NVIDIA Tesla K20 GPU, using the vendor-supplied BLAS libraries.

The outline of this paper is structured as follows: the Sect. 2 details the architectural features of MIC, the Sect. 3 discusses different dimensions of parallelisms of a vectorized SpMV kernel, the Sect. 4 is devoted to the experimental environment and results, the Sect. 5 concentrates on the performance analysis and modeling, and the Sect. 6 concludes.

2 Architectural Overview of MIC

The Intel Xeon Phi coprocessor is x86-based many-core architecture. It has 61 cores connected via a 512-bit bidirectional ring interconnect. There are 8 memory controllers supporting up to 16 GDDR5 channels. The core's memory interface are 32-bit wide with two channels which sustains a total bandwidth 8.4 GB/s per core. The STREAM Triad benchmark achieves around 160 GB/s on this architecture with ECC turned on.

There are two levels of cache memory. The level one cache has 32 KB instruction cache and 32 KB data cache. Associativity was increased to 8-way, with a 64 byte cache line. The bank width is 8 bytes. Data return is out-of-order. The L1 cache has a load-to-use latency of 1 cycle which allows an integer value loaded from the cache to be used in the next clock by an integer instruction. The L2 cache has a unified 512 KB capacity. Each core can access to all other L2 cache via the ring interconnection which makes a collective L2 cache size up to 32 MB. The L2 organization comprises 64 bytes per way with 8-way associativity, 1024 sets, 2 banks, 32 GB (35 bits) of cacheable address range and a raw latency of 11 clocks [8].

The vector processing units (VPU) of each core contains 32×512 bit SIMD registers which accommodates eight 64-bit values or sixteen 32-bit values. The VPU supports Fused Multiply-Add (FMA) operations which, for benchmarking purposes, counted as two floating operations. All these factors enable one single chip to yield over 1 TFlops double precision peak performance.

3 Sparse Matrix Vector Product Implementations for CSR Format

The CSR [11] comprises of 3 arrays, *row_ptrs*, *col_inds*, and *vals*, representing respectively the position of the first nonzero element of each row stored inside of *vals*, the column indices of every single nonzero element stored in *vals*, and the nonzero entries of the matrix in row-major order. Taking the standard CSR format as a starting point, we derived a vectorized kernel for SpMV.

Algorithm 1. Vectorized multiplication of the kth row (zero-based) of a matrix stored in CSR format (in *row_ptrs*, *col_inds*, and *vals*) with the vector x.

$reg\_y \leftarrow 0$
$start \leftarrow row\_ptrs[k]$
$end \leftarrow row\_ptrs[k+1]$
for $i = start$ to end **do**
  $writemask \leftarrow (end - i) > 8$? `0xff` : (`0xff` $\gg (8 - end + i)$)
  $reg\_ind \leftarrow load(writemask, \&col\_inds[i])$
  $reg\_val \leftarrow load(writemask, \&vals[i])$
  $reg\_x \leftarrow gather(writemask, reg\_ind, x)$
  $reg\_y \leftarrow fmadd(reg\_x, reg\_val, reg\_y, writemask)$
  $i = i + 8$
end for
$y[k] = reduce\_add(reg\_y)$

3.1 Vectorized Kernel

For CSR, a natural way to parallelize the SpMV is to assign the subsets of rows to different execution units. The elementary operation is then shrinked into the product of a compressed sparse vector with a dense vector. By using the SIMD instruction we insert at the lowest dimension a parallelism resulting from the vectorization. In this direction we propose the row-wise vectorized kernel for SpMV, which is similar to recent work on SpMV for MIC [10]. The Algorithm 1 delineates the SIMDized kernel that handles the row-wise multiplication. The *writemask* functions as a shifting window ensuring only the lower portion of vector being operated when there're less than 8 nonzero elements left in a row. The "8" in Algorithm 1 implies 8 double precision floating numbers which fill the 512-bits SIMD units in MIC. It is worth noting that, to ensure the correctness of results, the CSR used here must not be a simplified form for symmetric matrices.

3.2 Hierarchical Exploitation of Hardware Resources

The second dimension of parallelism is built upon the number of cores. Along with the hierarchical memory subsystem, these resources can be exploited by processes and threads spawned and managed by the multiprocessing techniques. In most cases, it is easier to implement with the pure model than the hybrid one. In this paper we discuss both pure and hybrid implementations. We expect to promote the efficiency of data access and alleviate the scaling pressure occurred in the pure model by mixing different approaches.

We define the SpMV process $y \leftarrow Ax + y$ as two phases:

1. The computing phase, where all elements of y should be calculated.
2. The communication phase, where y is copied to x.

The communication phase occurred usually in a iterative solver where the SpMV process needs to be repeated until the convergence of the solver. Because the memory space is unified for all threads, the communication phase of pure OpenMP can't be started before the termination of computing phase. However, with the participation of MPI, these two phases could be partially overlapped. In our study, we collect the computing phase timings corresponding to the slowest MPI process of each execution. These timing data were used to deduce the performance of SpMV. To obtain statistically meaningful results, we iterated 100 times for each measure of SpMV timing.

In terms of implementations, some conventional optimizations are applied to both pure and hybrid cases, such as software prefetching, and streaming stores. The rows of matrix are distributed so that each process receives the same number of nonzero elements. The minimal unit of partioning is one row. We also altered the number of processes and threads and attempted exhaustively all possible combinations of processes and threads to seek the best configuration of maximizing the performance for each test matrix. In particular, we don't take advantage of matrix symmetry to achieve better performance. All matrices are considered equally as non-symmetric ones. For the sake of better cache usage and to avoid oversubscription of threads, it is important to configure properly the processes' and threads' affinity. What's more, the highest numbered core of MIC (61th) should be left unused so it may process the interference from OS threads.

4 Experimental Results

4.1 Matrix Suite

In practice, we have 3 principles in selecting the test matrices [6]. Firstly, we prefer the matrices that have been used in previous literatures. Secondly, the matrices should have a larger volume in memory than 30 MB, which is the aggregate L2 cache size of Xeon Phi, in order to neutralize the promotion in temporal locality induced by repeated runs of SpMV kernel. Lastly, our future study of eigensolvers requires the matrices to be square. We also include a dense 8000×8000 matrix (*dense8000*) expressed in CSR format. We outline the basic characteristics of 18 selected matrices in Table 1.

Table 1. List of main characteristics of test matrices. nnz is the number of nonzero elements. nrow is the square matrix dimension.

Name	Dim (K)	nnz (M)	nnz/nrow	Name	Dim (K)	nnz (M)	nnz/nrow
mixtank_new	29.957	1.995	66.597	sme3Db	29.067	2.081	71.595
mip1	66.463	10.353	155.768	ldoor	952.203	46.522	48.858
rajat31	4690.002	20.316	4.332	Si41Ge41H72	185.639	15.011	80.863
nd6k	18.000	6.897	383.184	pdb1HYS	36.417	4.345	119.306
cage15	5154.859	99.199	19.244	bone010	986.703	71.666	72.632
crankseg_2	63.838	14.149	221.637	dense8000	8	64.000	8000
ns3Da	20.414	1.680	82.277	pwtk	217.918	11.634	53.389
in-2004	1382.908	16.917	12.233	torso1	116.158	8.517	73.318
circuit5M	5558.326	59.524	10.709				

4.2 Experimental Environment

Different SpMV kernels were conducted and compared on various architectures.

- Intel MIC, pre-production of KNC prototype C0, 61 cores running at 1.2 GHz, 16 GB GDDR5 memory with ECC enabled, installed with MPSS v3.1.
- Dual-socket Intel Xeon E5-2670, 16 core running at 2.6 GHz, 64 GB DDR memory with ECC enabled.
- NVIDIA K20 GPU, 2496 cores running at 0.7 GHz, 5 GB GDDR5 memory with ECC enabled.

On MIC, SpMV kernels of pure OpenMP, hybrid MPI/OpenMP, MKL (Intel Math Kernel Library, v11.1) were tested. On CPU, only the MKL SpMV routine was tested. On GPU, cuSPARSE (NVIDIA CUDA Sparse Matrix library, v2.6) kernel was tested. All tested vendor-supplied BLAS libraries use the CSR sparse format.

4.3 OpenMP and MKL Performances

The SpMV is one of the challenging instances that is known to be memory bandwidth bound. Its streaming memory access pattern makes the cores hard to run at full speed. Adding the number of threads helps to hide the latency due to data miss. But the increase of virtual cores might leads to memory contention and network congestion, thus exhibits a poor scaling performance. At core level, the vectorization is necessary for improving performance. However, it also burdens more on memory subsystem for the vector instructions consume much more data than scalar ones. The load of data is less efficient for x than for *col_inds* or *vals*. A unified address of x for all cores may drive the problem even more severe. We implemented on MIC a multithreaded SpMV kernel using OpenMP. The MKL version was also tested as it is based on OpenMP threading environment therefore comparable to our implementation. We varied the number of threads

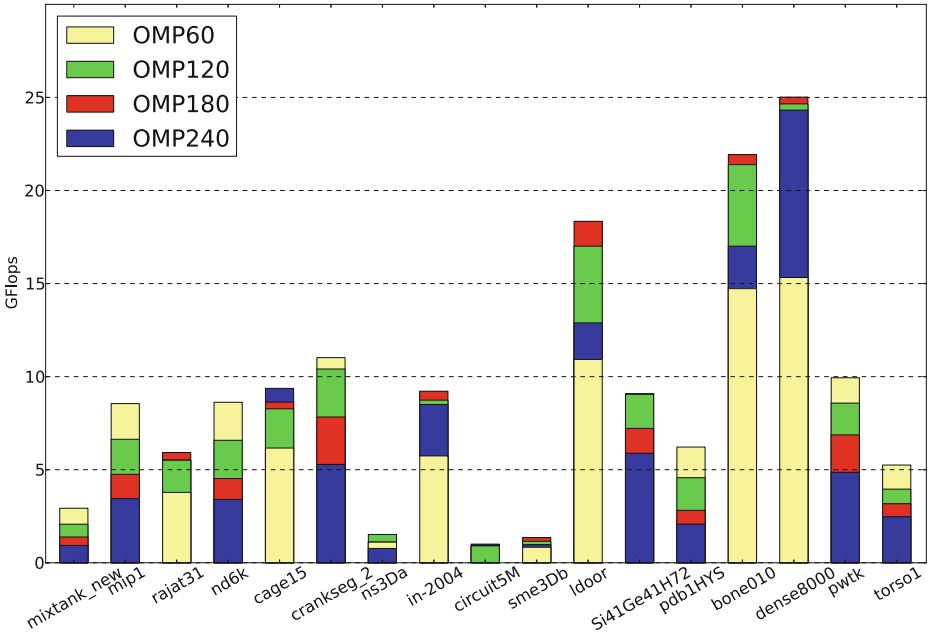

Fig. 1. Performances of OpenMP version of SpMV kernel. For each test matrix the performances are plotted in different colors depending on the number of threads used (Color figure online).

(from 1 to 4 threads per core) while measuring the performances. For each matrix we plot in Figs. 1 and 2 the bars of performance among which from top to bottom the performances corresponding to different thread configuration (1, 2, 3 or 4 threads per core) are shown in a descending order. All bars started from 0 GFlops. The lower part of the bars may be covered by other bars with smaller magnitude except the smallest one.

From these two figures we observe a similar behavior of both implementations on different matrices. None of them performs better in average than the other one, except that the MKL tends to have better performance when using more threads per core. We argue that's because of its better thread scheduling and some low-level optimizations. Both implementations are nowhere near the theoretical or achievable peak performance of MIC architecture.

4.4 Hybrid MPI/OpenMP Performances

To better deal with the issue of thread scaling and alleviate the memory contention, we propose to implement the hybrid MPI/OpenMP SpMV kernel. We expect to promote the efficiency of multithreading, scaling and cache utilization. In this case, we evenly divide the cores' domain according to common resources (cores, caches) and place one MPI process for each subdomain. In each subdomain, we spawn the same number of threads. The experiments were conducted

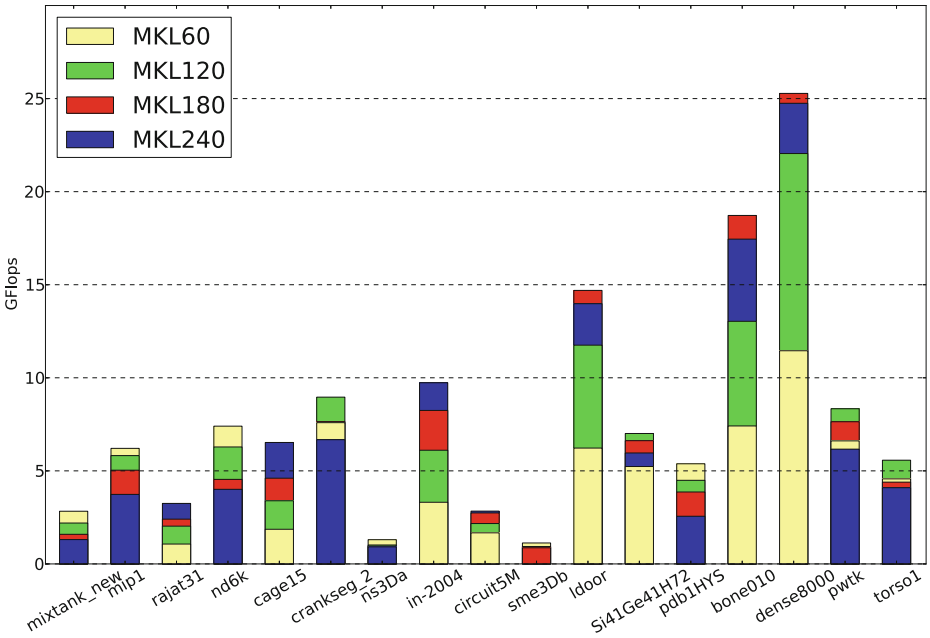

Fig. 2. Performances of MKL version of SpMV kernel. For each test matrix the performances are plotted in different colors depending on the number of threads used (Color figure online).

using all possible combinations of processes and threads with careful pinnings. Every subdomain governed by one MPI process is guaranteed to have the same and integer number of cores. And the highest numbered core of MIC is always free from application threads. All threads participate in the parallelization of vectorized SpMV kernel. Only the master thread manages the communication. The hybrid algorithm is described in Algorithm 2. We plot the gain of hybrid model against pure OpenMP in the Fig. 3. Over the entire matrix suite, the hybrid model exhibits a substantial performance improvement except in one case (*cage15*).

Algorithm 2. Hybrid MPI/OpenMP algorithm. Each MPI process accommodates the same number of OpenMP threads.

Distribute row blocks (rowptrs, colinds, vals) of A so that each MPI process receives approximately same number of nonzero elements
Replicate x on all MPI processes, allocate y (same size of x) on all MPI processes
Apply locally the vectorized SpMV kernel using OpenMP multithreading with "guided" scheduling
Gather the results from other MPI processes and update the local portion of y

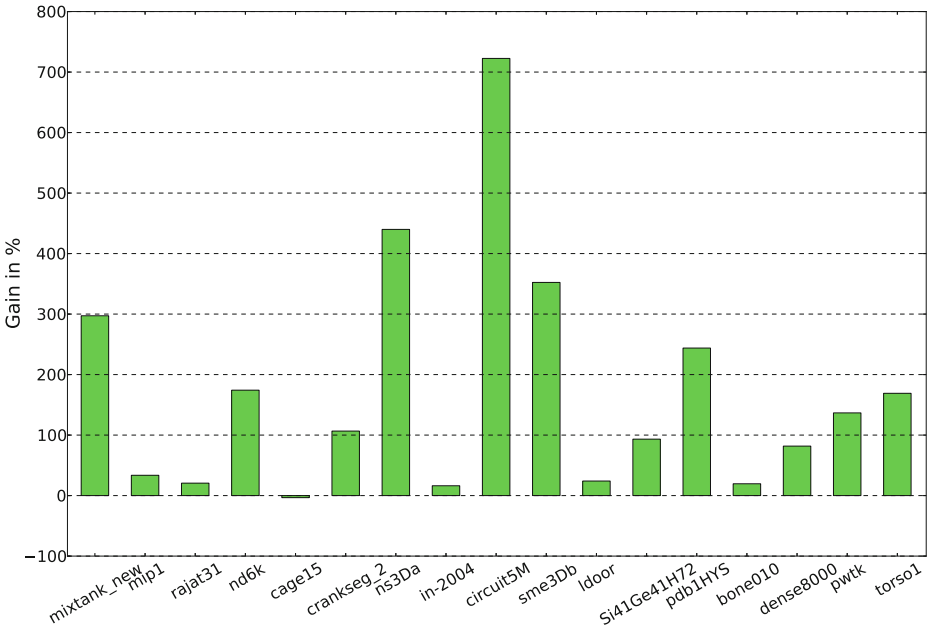

Fig. 3. Gain in percentage of hybrid MPI/OpenMP SpMV kernel against the pure OpenMP one.

4.5 Performances of SpMV Kernel on Various Architectures

Finally, the performances of different SpMV kernels will be presented here. The Fig. 4 delineates the performances of hybrid model versus vendor-supplied BLAS libraries across a variety of architectures. In most cases, the hybrid model outruns the other ones. Since we used the CSR format for all architectures, the results do not represent the inherent capacity of some architecture such as GPU. But it shows a path to better exploit the MIC architecture. We notice in some cases that CPU still achieved better performances. We will try to understand this phenomenon in Sect. 5.

5 Performance Analysis and Modeling

The experimental results reveal a considerable advantage of hybrid model over the pure ones. However, not being able to determine in advance the optimal combination of MPI processes and OpenMP threads invalidates this approach simply because the best results are irreproducible. As a consequence, it is imperative to devise a method to sketch the behavior of the machine. We will discuss qualitatively the reasons of performance improvement and the primary performance restraining factors, from where we develop a mathematical relationship that quantifies the effects of different factors. The effectiveness of the deduced model will be verified at the end of this section.

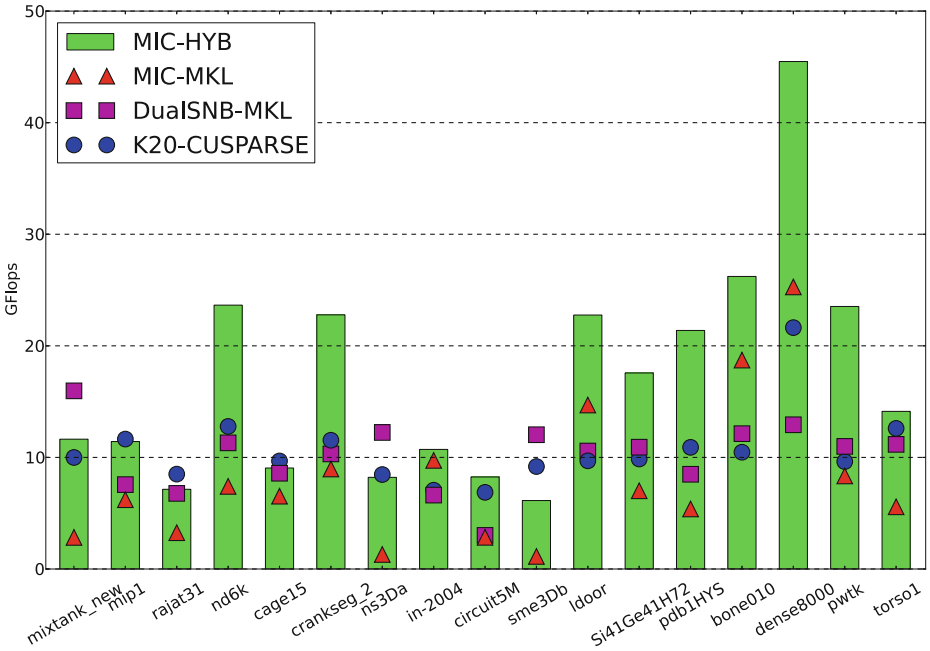

Fig. 4. Performances of different SpMV kernels on various architectures.

5.1 Performance Analysis

First thing to understand is the performance improvement due to the mixture of MPI and OpenMP. We argue that's mainly because of the promotion of data locality and thread scalability. The promoted data locality improves the data reusability in terms of better cache utilization. It also mitigates the memory contention. More specifically, the vector x is replicated, thus avoiding contention when large number of threads read elements of x. This would also be possible in pure OpenMP via thread-private variables. However, that means the replication of x has to be made on all threads. The memory usage would be varied if number of threads changes. In the case of hybrid model, the x is only replicated on each process and shared by threads belonging to that process which creates us a higher degree of flexibility. In addition, the rows of matrix A is distributed to different memory regions. Therefore, these data are spatially local to the process domain. By carefully binding the processes to the physical cores, the data are stored uniformly in the memory space. Therefore it is more likely to generate a higher aggregate bandwidth in the on-chip ring network.

The scaling factor should also be considered. In a large many-core system, the multithreading overheads such as loop scheduling overheads may not be linear when the number of threads grows. However, in hybrid model, each process keeps a relatively small number of threads making it easier to scale.

There are other potential advantages for hybrid model as well. For example, it is straightforward to implement it in a numerical software environment such as Trilinos, where the underlying MPI/OpenMP modules are already encapsulated and ready to use.

In spite of some performance improvement, the hybrid SpMV kernel still performs poorly in some cases compared to other implementations. The poor performances are likely to occurred in matrices having a low average number of nonzero elements[1], as seen in Table 1 for *rajat31*, *cage15*, *in-2004*, *circuit5M*. However, these are not the only matrices behaving badly. The performances of *ns3Da* and *sme3Db* are not promising either whereas they have decent average numbers of nonzero elements. Further research shows that their nonzero elements are distributed dispersedly and sparsely along the rows, which may lead to poor vectorization efficiency. Since each thread performs the vectorized multiplication between two arrays at a time, small number of nonzeros per row makes the vector instruction overheads significant compared to the whole execution time, thus inducing a low vectorization rate. In general, low vectorization rate shakes the foundation of producing high performance. Though this reasoning does not applied well in *ns3Da* and *sme3Db*. The nonzero elements of these two matrices are not only numerous but also uniformly spreaded. The vectorization should be well conducted unless certain operation described in Algorithm 1 decelerates the computation. The most probable explanation would be that the *gather* instruction appeared in line 8 of Algorithm 1 cancels out the high vectorization rate because of its long latency. The irregularity of nonzero elements makes the load of x inefficient, thus causing the unexpected cache misses and eventually bad performance.

Besides these two factors, there should be one more concern linked to the message passing programming, which is the load balancing issue. In the hybrid model, the processes are independent and the last terminated process determines the global performance. In our case, this issue is connected to the row partitioning policy. Using a dynamic instead of static row distributing strategy may improve load balancing. We will include this study in our future works.

5.2 Performance Modeling

Definition 1. *For a given matrix, let the nnz be the number of nonzero elements. If t is the execution time of SpMV computing phase defined in Sect. 3.2, then the performance P of a hybrid SpMV kernel is defined as*

$$P = \frac{2\ nnz}{t_{max}}$$

where t_{max} is the execution time of the last terminated MPI process.

[1] The average number of nonzero elements is defined as the quotient of total number of nonzero elements over the row dimension.

If nnz_{glob} is the total number of nonzero elements in a given matrix, then the global performance P_{glob} is given by

$$P_{glob} = \frac{2\ nnz_{glob}}{t_{max}}$$

To properly model the performance, we want to seperate the load balancing factor from the vectorization rate and nonzero elements' irregularity. Since the t_{max} is the time of the slowest MPI process, it is more accurate to obtain its local performance P_{local} by applying the same formula:

$$P_{local} = \frac{2\ nnz_{local}}{t_{max}} = \frac{nnz_{local}}{nnz_{glob}} P_{glob}$$

where nnz_{local} is the number of nonzero elements of the row block assigned to the slowest process. We measured the execution time of the slowest process with its rank recorded. The ranking information helps to identify the row blocks assigned to the processes. Since thread is the minimal execution unit which performs vector instructions, the per thread performance is more meaningful for characterizing the indicators discussed in the last subsection.

Definition 2. *If P is the aggregate performance of n_{thd} number of threads, then the per-thread performance is estimated as*

$$P_{thd} = \frac{P}{n_{thd}}$$

Assume the n_{thd} is the number of threads spawned within the slowest process, then the local per-thread performance is

$$P_{thd} = \frac{P_{local}}{n_{thd}}$$

In this context, two indicators are proposed to quantify the SIMD efficiency as well as the impact of nonzero elements' dispersion. The first one is the average number of nonzero elements. There are at least three features helping to set up the functional relationship between this indicator and the per-thread performance. All these features are discussed without the interference of the second indicator.

1. If the number of nonzero elements equals to 0, the performance should also be 0. However, as the average number of nonzero elements starts from 0, the impact of vector instruction overheads might diminish rapidly.
2. Bigger the average number of nonzero elements is, less amplification of performance is gained.
3. The performance should have an upper bound as the number of nonzero elements is extremely large.

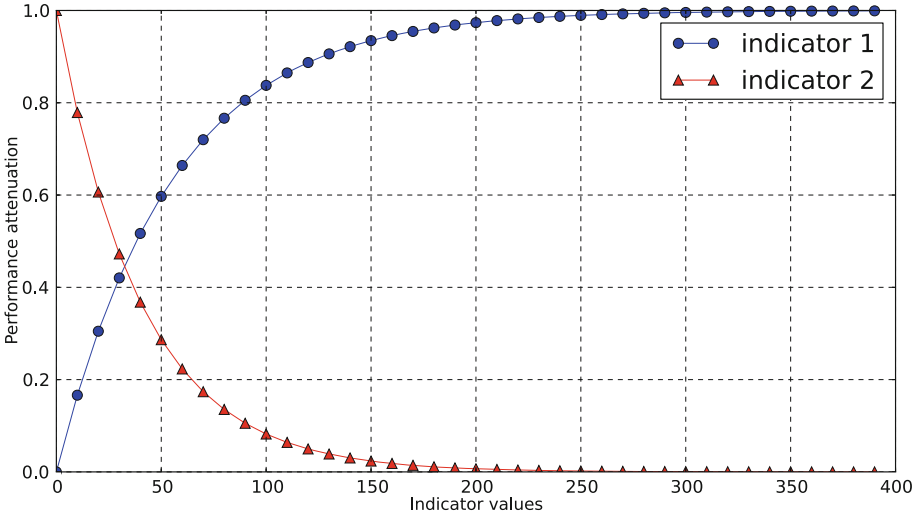

Fig. 5. The relationships between two indicators and the performance. The first indicator is the average number of nonzero elements per row. The second indicator is the average number of occurrences when the distance between any pair of contiguous nonzero elements within a row is greater than 2.

Such relationship could be delineated by the blue curve in Fig. 5. With more nonzero elements in a row we have generally better performance, if given a fixed level of nonzero elements' dispersion. The second indicator should then be able to describe the "level of dispersion". A direct solution is to estimate the cache misses. However, the cache behavior in modern architecture depends on, including but not limited to, cache capacity, cache associativity, cache line width, cache levels, and replacement policy. It is highly unpredictable using a low-cost model. Considering its complexity, this method is less practical. We are searching for a convenient and simple approach to establish the second indicator. It turns out that a simple trait of matrix, which based on the distances between each pair of contiguous nonzero elements in a row, is an effective indicator. It refers to the average number of occurrences when such distance is greater than 2. Similar to the first indicator, it averages over all studied rows. The red convex decreasing curve with triangle markers in Fig. 5 depicts the attenuation caused by the second indicator to the performance. According to the graphs of two indicators, we give the functional form of regression model in Eq. 1, where $\hat{P}_{thd}$ is the estimated per-thread performance, $\overline{nnz}$ is the first indicator and the $\overline{d}$ is the second indicator.

$$\hat{P}_{thd}(\overline{nnz}, \overline{d}) = \alpha \left[1 - \exp\left(-\frac{\overline{nnz}}{\epsilon_1}\right)\right] \exp\left(-\frac{\overline{d}}{\epsilon_2}\right) \tag{1}$$

The experimental data were collected from the slowest processes of different matrices listed in Table 1. These data were processed to obtain P_{local}, n_{thd}, $\overline{nnz}_{local}$, $\overline{d}_{local}$ for the use of regression analysis.

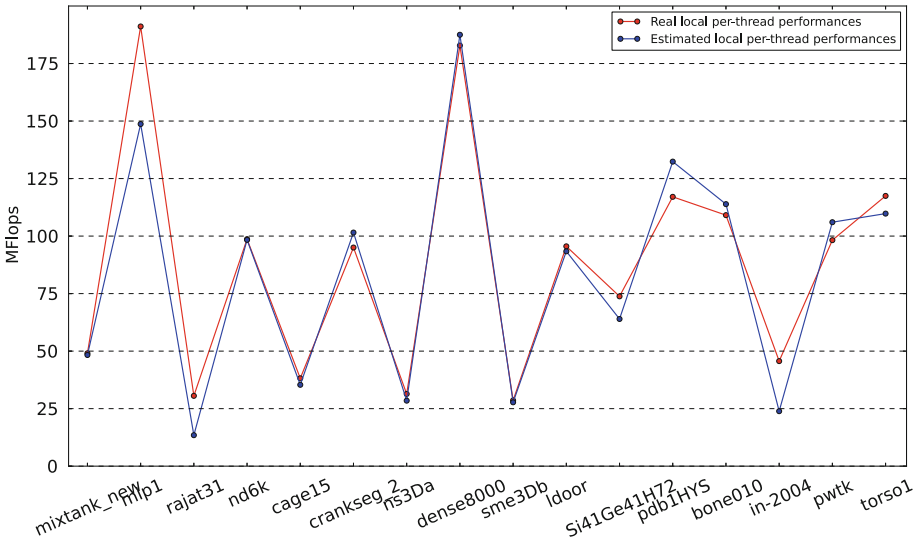

Fig. 6. The real and the estimated local per-thread performances over a set of test matrices.

The Fig. 6 draws the real and the estimated local per-thread performances over a set of test matrices. The local performance were collected from the slowest process of each execution. It is based on this set of data that we obtained the following coefficients. The estimated values are:

$$\widehat{\alpha} = 187.5,\ \widehat{\epsilon_1} = 55,\ \widehat{\epsilon_2} = 40$$

where the $\widehat{\alpha}$ corresponds to the per-thread performance (in MFlops) of *dense8000*. Considering its large average number of nonzero elements per row and continuous nonzero elements, it should execute almost optimally.

6 Conclusions

The SpMV is the key kernel that constitutes the main process in many iterative numerical methods. In this paper, we investigate two programming models, the pure OpenMP and the hybrid MPI/OpenMP. Starting from a vectorized CSR SpMV kernel, we proposed different ways of parallelizing it. A set of evaluations on various mainstream architectures (Intel Dual-Socket Sandy Bridge, NVIDIA K20 GPU) was conducted by using not only our own implementations but also the vendor supplied BLAS libraries. The results suggest that the hybrid MPI/OpenMP model is very promising on Intel MIC architecture. It can help to reduce the scaling overheads and promote data locality compared to the pure models, thus improving substantially the performance.

In order to better understand the performance of hybrid model, we identified 3 performance indicators, namely the average number of nonzero elements,

the average number of occurrences when the distance between any two contiguous nonzero elements within a row is greater than 2, along with the load balancing. We studied the impacts of the first two indicators within the last terminated process and came up with a regression model based on the experimental data. We also estimated the regression coefficients and obtained good fitting results.

References

1. Balay, S., Brown, J., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Web page (2013). http://www.mcs.anl.gov/petsc
2. Berrendorf, R., Nieken, G.: Performance characteristics for OpenMP constructs on different parallel computer architectures. Concurrency Pract. Exp. **12**(12), 1261–1273 (2000)
3. Bull, J.M.: Measuring synchronisation and scheduling overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, pp. 99–105 (1999)
4. Cappello, F., Etiemble, D.: MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Supercomputing 2000. IEEE Computer Society, Washington, DC (2000). http://dl.acm.org/citation.cfm?id=370049.370071
5. Chow, E., Hysom, D.: Assessing performance of hybrid MPI/OpenMP programs on SMP clusters. Technical report, Lawrence Livermore National Laboratory (2001)
6. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Trans. Math. Softw **38**(1), 1:1–1:25 (2011). http://doi.acm.org/10.1145/2049662.2049663
7. Heroux, M., Bartlett, R., Hoekstra, V.H.R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., Williams, A.: An overview of trilinos. Technical report, SAND2003-2927, Sandia National Laboratories (2003)
8. Intel: Intel Xeon Phi Coprocessor System Software Developers Guide. Technical report (2012)
9. Kourtis, K., Goumas, G., Koziris, N.: Exploiting compression opportunities to improve SpMxV performance on shared memory systems. ACM Trans. Architec. Code Optim. **7**(3), 16:1–16:31 (2010)
10. Liu, X., Smelyanskiy, M., Chow, E., Dubey, P.: Efficient sparse matrix-vector multiplication on x86-based many-core processors. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS 2013, pp. 273–282. ACM, New York (2013). http://doi.acm.org/10.1145/2464996.2465013
11. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia (2003)
12. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC 2007, pp. 38:1–38:12. ACM, New York (2007). http://doi.acm.org/10.1145/1362622.1362674