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

Fan Ye[1,2], Christophe Calvin[1], Serge Petiton[2,3]

[1] CEA/DEN/DANS/DM2S, CEA Saclay, France
[2] LIFL, Université de Lille 1 Sciences et Technologies, France
[3] Maison de la Simulation, Gif-Sur-Yvette, France

Abstract. The Sparse Matrix-Vector Multiplication is the key operation in many iterative methods. The widely used CSR (Compressed Sparse Row)[7] format was chosen to carry on this study for sustainability and reusability reasons. We parallelized for Intel MIC architecture a vectorized SpMV kernel using respectively the pure and the hybrid MPI/OpenMP models. In comparison to pure models and vendor-supplied BLAS libraries across different mainstream architectures (CPU, GPU), the hybrid model exhibits a substantial improvement. It can help to promote the data locality and thread scalability. To further assess the performance, two indicators characterizing the nonzeros are proposed to model the performance.

1 Introduction

The Sparse Matrix-Vector multiplication (SpMV) is fundemental to a wide spectrum of scientific and engineering applications. We want to study the SpMV within the context of studying Krylov solvers, making the availability of large numerical packages such as PETSc or Trilinos prominent to us. As CSR is the commonly used sparse format in those packages, it is chosen to carry on this study.

The evolution of physical model and the increasing exigency for accuracy, reliability and computing speed lead the HPC community to exploit and efficiently use new architectures. In this context, heterogeneous supercomputers arise, enforcing the use of accelerators like GPU or MIC[4]. In this paper, we refer to the Intel Xeon Phi Coprocessor as the underlying system for revealing some idiosyncrasies in SpMV implementation. A simplified way to view the many cores in this architecture is as a chip-level SMP which offers remarkably high bandwidth. The prototype 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 irregular nature of sparse matrices, the memory subsystem often appears as the main bottleneck of SpMV. Furthermore, in a shared memory context

[4] Intel Many Integrated Core architecture

with a large count of cores, the scalability behavior is elusive which depends heavily on issues like data locality, access pattern, as well as the programming models. The preceding studies [2, 4] hold pessimistic views of hybrid fashion compared to a unified MPI approach. But the related literatures usually underline the importance of network performance in explaining the gap between different models. Thus the appealing on-chip bandwidth of MIC drives us to investigate the potential benefit of using hybrid programming. We define the hybrid model in the case of MIC architecture as one that hierarchically exploits the computational resources by the combination of MPI and OpenMP. The data distributed over processes can be exploited more locally than in a pure multithreading version.

As mentioned before, we use the Intel Xeon Phi Coprocessor as the principal test platform. To set an architectural baseline, we perform the tests over the same matrix suite on dual Intel Sandy-Bridge octo-core processors, as well as the NVIDIA Tesla K20 GPU, using the vendor-supplied BLAS libraries.

The outline of this paper is the following: in the next section we detail the SpMV implementation on MIC. Section 3 is devoted to experimental results with a focus on performance analysis and modeling. Section 4 concludes.

2 Sparse Matrix Vector Product Implementations for CSR Format

The CSR comprises of 3 arrays, *row_ptrs*, *col_inds*, and *vals*, representing respectively the row, column indexing information and nonzero values. Taking the standard CSR format as a starting point, we derived a vectorized kernel for SpMV.

2.1 Vectorized Kernel

For CSR, a natural way to parallelize the SpMV is to assign the subsets of rows to 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 [3]. The Alg. 1 delineates the SIMDized kernel that handles the row-wise multiplication. The *writemask* functions as a window ensuring only the lower portion of vector being operated when there're less than 8 nonzeros left in a row. The "8" in Alg. 1 implies 8 double precision floating numbers that occupy the 512-bits SIMD units in MIC.

2.2 Hierarchical Exploitation of Hardware Resources

The second dimension of parallelism is built upon the number of cores. Along with the hierarchical memory subsystem, these computational resources can be exploited by the execution units spawned and managed by the multiprocessing techniques. In this paper we demonstrate a spectrum where the two extremes

Algorithm 1 Row-wise vectorized kernel using CSR format (*row_ptrs, col_inds, vals*). "*reg_**" denotes vector graphic streaming SIMD extension register used by intrinsic functions.

1: $reg_y \leftarrow 0$
2: $start \leftarrow row_ptrs[row]$
3: $end \leftarrow row_ptrs[row + 1]$
4: **for** $i = start$ to end **do**
5: $\quad writemask \leftarrow (end - i) > 8$? `0xff` : (`0xff` $\gg (8 - end + i))$
6: $\quad reg_ind \leftarrow load(writemask, \&col_inds[i])$
7: $\quad reg_val \leftarrow load(writemask, \&vals[i])$
8: $\quad reg_x \leftarrow gather(writemask, reg_ind, x)$
9: $\quad reg_y \leftarrow fmadd(reg_x, reg_val, reg_y, writemask)$
10: $\quad i = i + 8$
11: **end for**
12: $y[row] = reduce_add(reg_y)$

are respectively the pure OpenMP and the flat MPI. In between constitute the hybrid model that nests the OpenMP threads under the MPI processes. We define the SpMV process $y \leftarrow Ax$ as two phases. The computing phase, where all elements of y should be calculated. The communication phase, where y is copied to x. Because of the unified memory space, 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 this case, we collect the computing phase timings that correspond to the longest MPI process of each run. These timing data were used to deduce the performance of SpMV kernel.

In terms of implementations, we applied some conventional optimizations to all three cases: OpenMP, MPI, and hybrid MPI/OpenMP, such as software prefetching and streaming stores. In the presence of MPI, the rows of matrix are distributed in a way that each process MPI receives the same number of nonzero elements. The minimal unit of partioning is one row. We altered the number of processes and threads to seek the best configuration of maximizing the performance for each test matrix. To prevent the oversubscription of a certain area, the affinity factor was considered so as to properly bind the MPI processes and OpenMP threads to physical processing units. More specifically, the Intel® library provides additional environment variable to control the processes/threads' affinity. For threads, we set "*KMP_AFFINITY*" to "*granularity=thread,scatter*". For processes, according to the performance, we set "*I_MPI_PIN_DOMAIN*" to "*omp*" or "*auto*".

3 Experimental Results

3.1 Matrix Suite

In practice, we have 3 principles in selecting the test matrices [6]. Firstly, we favor the matrices that have been used in previous literatures. Secondly, the

Table 1. Main characteristics of test matrices (*nnz*: the number of nonzero entries, *nrow*: the number of rows)

Name	Dim (K)	nnz (M)	nnz/nrow
mixtank_new	29.957	1.995	66.597
mip1	66.463	10.353	155.768
rajat31	4690.002	20.316	4.332
nd6k	18.000	6.897	383.184
cage15	5154.859	99.199	19.244
crankseg_2	63.838	14.149	221.637
ns3Da	20.414	1.680	82.277
in-2004	1382.908	16.917	12.233
circuit5M	5558.326	59.524	10.709

Name	Dim (K)	nnz (M)	nnz/nrow
sme3Db	29.067	2.081	71.595
ldoor	952.203	46.522	48.858
Si41Ge41H72	185.639	15.011	80.863
pdb1HYS	36.417	4.345	119.306
bone010	986.703	71.666	72.632
dense8000	8	64.000	8000
pwtk	217.918	11.634	53.389
torso1	116.158	8.517	73.318

matrices should have a larger volume in memory than 30 MB which equals the aggregate L2 cache size of Xeon Phi, to neutralize the promotion in temporal locality induced by repeated runs of SpMV kernel. Lastly and most importantly, we require the matrices to be square to meet the premise of our study of Krylov eigensolver. We also include a dense 8000×8000 matrix (*dense8000*) expressed in CSR format. We outline the basic characteristics of 18 selected matrices in Tab. 3.1.

3.2 Experimental Environment

Different SpMV kernels were conducted and compared on various architectures, including pre-production of KNC C0, dual-socket Intel Xeon E5-2670 and NVIDIA K20 GPU. On MIC, SpMV kernels of pure OpenMP, hybrid, flat-MPI, MKL were tested. On CPU, MKL kernel was tested. On GPU, cuSPARSE kernel was tested. All tested vendor-supplied BLAS libraries adopted the CSR format.

3.3 OpenMP and MKL Performances

The pure multithreading programming model is a natural option for MIC architecture as it has been designed to a shared memory system. However, the streaming memory access pattern of SpMV makes the cores hard to run at full speed. Adding the number of threads helps to hide the core stall due to data miss. But the increase of virtual cores may lead to memory contention and network congestion, thus exhibits a poor scaling performance. At core level, the vectorization represent the first parallelism to sufficiently exploit the available performance. However, it may also burden more on the memory subsystem.

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 kernel. We measured the performances from 1 to 4 threads per core. For each matrix we plot in Fig. 1 the vertical bars of performance in which from top to down the performances corresponding to all threads

configuration (1, 2, 3 or 4 threads per core) are expressed in a descending order. From two figures we observe a similar behavior of both implementations on different matrices. None of them exhibit a better performance in average except that the MKL tends to have better performance with more threads per core.

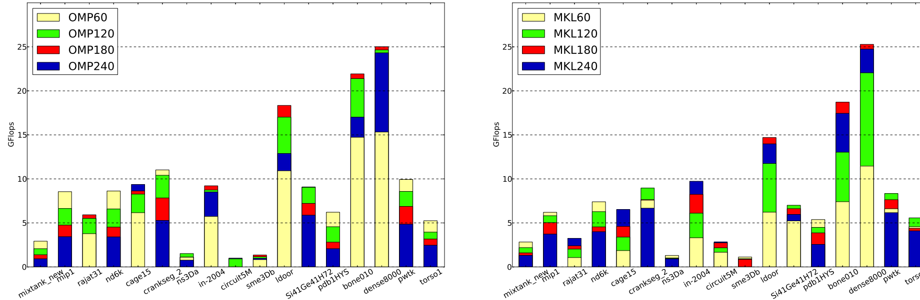

Fig. 1. Performances of the OpenMP (left) and MKL (right) SpMV kernel versions. For each matrix we plot the performances in GFlops according to the number of threads (60, 120, 180, 240)

3.4 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. The experiments were conducted using all possible combinations of processes and threads with careful pinnings. We plot the gain of hybrid model against pure OpenMP in Fig. 2. Over the entire matrix suite, the hybrid model exhibits a substantial performance improvement except in one case (*cage15*). The hybrid algorithm is shown in Alg. 2. We also tested the flat-MPI implementation. The results were not promising. Generally, more MPI processes better hide the computing latency, which implies also much higher communication cost, especially while gathering the results of y. For the sake of brevity, the results related to flat-MPI are not reported.

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

1: Distribute row blocks (rowptrs, colinds, vals) of A so that each MPI process receives approximately same number of nonzeros
2: Replicate x on all MPI processes, allocate y (same size of x) on all MPI processes
3: Apply locally the vectorized SpMV kernel with OMP multithreading
4: Gather the results from other MPI processes and update the local portion of y

---

3.5 Performances of SpMV kernel on various architectures

Finally, the performances of different SpMV kernels will be presented here. The Fig. 3 demonstrates the performances of hybrid model versus vendor-supplied BLAS libraries across various architectures. In most cases, the hybrid model obtained better performances. Since we adopted the CSR format for all architectures, the results can't reflect the potential of GPU. But it shows a path to better exploit the parallelisms on MIC architecture.

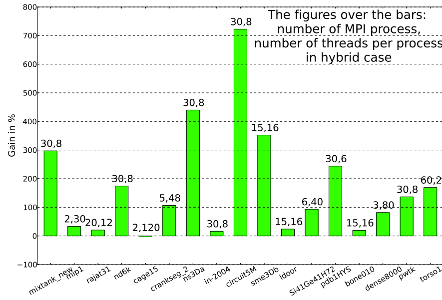

Fig. 2. Gain in percentage of hybrid MPI/OpenMP SpMV kernel against the pure OpenMP one, both on their best cases

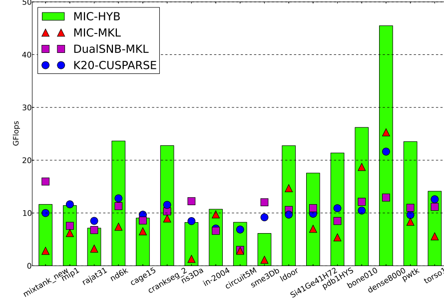

Fig. 3. Performances of the different SpMV kernels on various architectures (DualSNB in the legend refers to dual-socket Intel Xeon E5-2670)

3.6 Performances Analysis and Modeling

The experimental results reveal a considerable advantage of hybrid programming model over the pure ones. 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. For example, in Fig. 3, the hybrid performance of *dense8000* exceeds the sustainable performance limited by the achievable bandwidth[5]. It also mitigates the memory contention as each process keeps a copy of x and the rows are distributed deliberately to each process (see Alg. 2). Consequently, these data are spatially local to the process domain. By carefully binding the processes to physical cores, the data are stored uniformly in the memory space. Therefore it is more likely to achieve a higher aggregate bandwidth in the on-chip network. OpenMP often requires dynamic thread scheduling policies to obtain higher performance which makes it difficult to localize the data. In such a huge many-core system, a large thread pool is hard to manage due to the multithreading overheads. By using the hybrid model, each process is responsible for a small number of threads making it relatively easy

[5] Around 160 GB/s according to STREAM Triad benchmark.

to scale. However, the hybrid SpMV still performs poorly compared to other implementations in some cases. We attribute the poor performance to three main factors: vectorization rate, nonzeros dispersion rate, and load balancing. The first factor focuses on the quantity of nonzeros within a row while the second cares whether these nonzeros are close to each other. A large amount of nonzeros amortizes the vectorization overheads and thus raise the vectorization rate. The Alg. 1 collects the corresponding x entries through the *gather* instruction. More the nonzeros are dispersed, more likely the cache misses would be encountered. The dispersive nonzeros also make the loaded cache line inefficient. We model the per-thread hybrid SpMV performance by giving a mathematical relationship based on the first two factors. Two indicators are proposed to quantify their effects. The first one $\overline{nnz}$ is the average number of nonzeros per row. The second one $\overline{d}$ is the average number of occurrences when the distance between any pair of contiguous nonzero elements within a row is greater than 2. Assuming the relationship between these two indicators and the performance is depicted in Fig. 4, we choose the functional form as proposed in Eq. (1).

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

The per-thread performance is defined in Eq. (2).

$$P_{thd} = \frac{2\ nnz_{total}}{t \cdot n_{thd}} \tag{2}$$

where t is the execution time, nnz_{total} is the total number of nonzeros within involved rows. Specifically, we consider the slowest MPI process so as to leave aside the load balancing factor. Taking the rows assigned to this process as samples, we deduce the corresponding values of two indicators. Using these data and Eq. (1) to perform the regression analysis, we procure the following estimated parameters. The results are shown in Fig. 5.

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

4 Conclusion

The SpMV is essential to many iterative numerical methods. The emerging many-core architecture permits higher performance but also complicates the way achieving that. In this paper, we investigate three programming models that consist of pure OpenMP, flat MPI, and hybrid MPI/OpenMP. 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 with regard to pure models, therefore improve substantially the performance. It is also straightforward to implement hybrid MPI/OpenMP in a numerical software environment such as Trilinos, where the underlying MPI/OpenMP modules are already encapsulated and ready to use. To better understand the

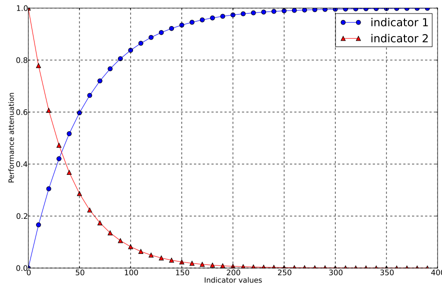

Fig. 4. The presumed relationships between two indicators (respectively $\overline{nnz}$ and $\overline{d}$) and the performance.

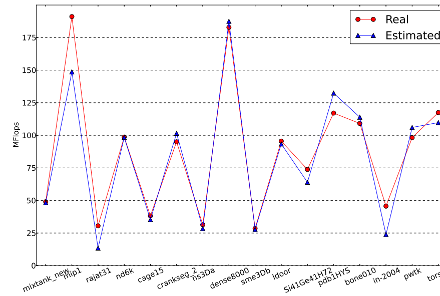

Fig. 5. The real and the estimated per-thread performances of the slowest MPI process over a set of test matrices.

performances of hybrid model, we identified 3 performance indicators, namely the average number of nonzeros, the average number of occurrences when the distance between any two contiguous nonzeros within a row is greater than 2, along with the load balancing. We studied the impacts of the first two indicators inside of the slowest process and devised a regression model based on the analysis. We then estimated the regression coefficients using the experimental results. The deduced model succeeded to predict the performances using very limited information about the matrix. This model can also be instructive for SpMV optimization. For example, an alternative row partitioning policy based on this model may ameliorate the quality of load balancing. We will include this in our future work.

References

1. Kourtis, K., Goumas, G., Koziris, N.: Exploiting Compression Opportunities to Improve SpMxV Performance on Shared Memory Systems. ACM Trans. Architec. Code Optim. 7, 3, Article 16 (2010)
2. Chow, E., Hysom, D.: Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters. Tech. Report UCRL-JC-143957, LLNL, Livermore, CA (2001)
3. Liu, X., Chow, E., Smelyanskiy, M., Dubey, P.: Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors. In ICS'13, Eugene, Oregon, USA (2013)
4. Cappello, F., Etiemble, D.: MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In Supercomputing 2000 (2000)
5. Bull, J. M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In First European Workshop on OpenMP 2000, Edinburgh, UK (2000)
6. Davis, T. A., Hu, Y.: The University of Florida Sparse Matrix Collection http://www.cise.ufl.edu/research/sparse/matrices/
7. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia, PA (2003)