Performance Limitations for Sparse Matrix-Vector Multiplications on Current Multi-Core Environments

Gerald Schubert, Georg Hager, and Holger Fehske

Abstract The increasing importance of multi-core processors calls for a reevaluation of established numerical algorithms in view of their ability to profit from this new hardware concept. In order to optimize the existent algorithms, a detailed knowledge of the different performance-limiting factors is mandatory. In this contribution we investigate sparse matrix-vector multiplications, which are the dominant operation in many sparse eigenvalue solvers. Two conceptually different storage schemes and computational kernels have been conceived in the past to target cache-based and vector architectures, respectively: compressed row and jagged diagonal storage. Starting from a series of microbenchmarks to single out performance limitations, we apply the gained insight to optimize sparse MVM implementations, reviewing serial and OpenMP-parallel performance on state-of-the-art multi-core systems.

1 Introduction

Large sparse eigensystems or systems of linear equations emerge from the mathematical description of many problems in physics, chemistry, and continuum dynamics. Solving those systems often requires multiplication of a sparse matrix with a vector as the dominant operation. Depending on the matrix dimension, its spar-

G. Schubert · G. Hager
Regionales Rechenzentrum Erlangen, Friedrich-Alexander Universität Erlangen-Nürnberg, Martensstr. 1, D-91058 Erlangen, Germany
e-mail: gerald.schubert@rrze.uni-erlangen.de
e-mail: georg.hager@rrze.uni-erlangen.de

H. Fehske
Ernst-Moritz-Arndt-Universität Greifswald, Institut für Physik, Felix-Hausdorff-Str. 6, D-17489 Greifswald, Germany
e-mail: fehske@physik.uni-greifswald.de

S. Wagner et al. (eds.), *High Performance Computing in Science and Engineering, Garching/Munich 2009*, DOI 10.1007/978-3-642-13872-0_2,

sity and the complexity of the remaining algorithm, the fraction spent in the sparse matrix-vector multiplication (SpMVM) may easily constitute over 99% of total run time. An efficient implementation of the SpMVM on current hardware is thus fundamental, and complements the development of sophisticated high-level algorithms like, e.g., Krylov-subspace techniques [13], preconditioners [7, 10], and multi-grid methods [5, 6].

The exponential growth in available computational power enables larger problems, finer grids and more complex physics to be tackled on present-day supercomputers. Distributed-memory parallelization has always been instrumental in exploiting the full potential of sparse solvers on massively parallel architectures, but with the advent of multi-core processors the per-core performance has begun to decline, and parallel computers have developed a strongly hierarchical structure. Efficient implementations of the SpMVM must hence be adapted to the new shared-memory and cache topology complexities as well. These optimizations will be the focus of this contribution. We target various current systems, including the recent Intel "Nehalem" and AMD "Shanghai" processors.

In recent years, the performance of various SpMVM algorithms has been evaluated by several groups [4, 9, 17]. Covering different matrix storage formats and implementations on various types of hardware, they reviewed a more or less large number of publicly available matrices and reported on the obtained performance. Their results account for the crucial influence of the sparsity pattern on the obtained performance: Similar to a fingerprint, the positions of the non-zero entries in the matrix determine which storage and multiplication scheme is best suited and how fast this operation can be performed on a given platform.

We will take a different approach by isolating the individual contributions to the run time of the SpMVM on a generic level, without resorting (at first) to specific matrices. Based on the assumption that erratic, indirect memory access patterns are decisive for SpMVM performance, an in-depth investigation of latency and bandwidth effects, hardware prefetching, and the influence of non-local memory access on cache-coherent non uniform memory access (ccNUMA) architectures (as implemented in almost all current HPC platforms) will clearly reveal the performance-limiting aspects on the multi-core and multi-socket (node) level. A successful performance model will be predictive for the expected performance of various SpMVM implementations for a given matrix on the basis of its sparsity pattern, and give a hint to the respective optimal storage scheme.

This paper is organized as follows. In Section 2 we describe the dominant sparse matrix storage schemes, CRS and JDS, together with some promising refinements to JDS. Section 4 presents low-level benchmarks for SpMVM-like access patterns and performance results for standard and optimized storage layouts. In Section 5 we elaborate on multi-threaded SpMVM performance, both in multi-core and ccNUMA contexts. Finally, Section 6 gives a summary and outlook to future work.

2 Common Storage Schemes

There is a variety of options for storing sparse matrices [3]. Despite similar memory requirements, the chosen format may substantially impact the SpMVM performance. Prior knowledge about the sparsity pattern (symmetry features, dense subblocks, off-diagonal structures etc.) can be exploited to generate a specialized format, suitable only for one type of physical problem but optimal with respect to data access properties. In absence of such additional information one is limited to general storage schemes. Among these, the most popular one is the compressed row storage (CRS) format, which is also the basis of many sparse solver packages like, e.g., SuperLU [8] and PETSc [1]. In a nutshell, the CRS format stores the matrix in three (one-dimensional) arrays. For each non-zero element, its value (`val`) and column index (`col_idx`) are stored. The third array (`row_ptr`) holds the offset into `val` where the entries belonging to a new matrix row start. If the number of matrix rows is `N_r`, the SpMVM kernel code for CRS looks like this:

```
do i = 1, N_r
  do j = row_ptr(i), row_ptr(i+1) - 1
    resvec(i) = resvec(i) + val(j) * invec(col_idx(j))
  enddo
enddo
```

The success of the CRS format lies in its simplicity and the high computational performance on most cache-based architectures, which results from the inner loop having the characteristics of a sparse scalar product with an algorithmic balance of 10 bytes/floating point operation (Flop) [12, 14]. On vector systems, however, this format has the serious drawback of a short inner loop if the number of non-zeros per row is not at least a couple of hundreds.

To make SpMVM run well on vector processors, alternative formats have been conceived, the most prominent being "jagged diagonals storage" (JDS). In JDS, large vector lengths are given priority over minimizing data transfer: In a first step the rows and columns of the matrix are permuted such that the number of elements per row decreases with increasing row index. All further calculations are then performed in this permuted basis. In each row the permuted non-zero elements are shifted to the left. The resulting columns of decreasing length are called *jagged diagonals* and are stored consecutively in memory. There is one array for non-zero values (`val`) and one for column indices (`col_idx`), just as with CRS. A third array (`jd_ptr`) holds the starting offsets of the jagged diagonals in `val` and `col_idx`. If `N_j` is the number of diagonals, the SpMVM kernel code for JDS looks like this:

```
do diag = 1, N_j
  diagLen = jd_ptr(diag+1) - jd_ptr(diag)
  offset = jd_ptr(diag) - 1
  do i = 1, diagLen
    resvec(i) = resvec(i) + val(offset+i) * invec(col_idx(offset+i))
  enddo
enddo
```

Since the inner loop has the characteristics of a sparse vector triad with an algorithmic balance of 18 bytes/Flop [12], its computational intensity is smaller than with CRS, but due to the large loop length it is much better suited for vector processors and similar machines.

In view of the changing supercomputing landscape, where traditional cache-based and vector architectures are being replaced by hybrid, hierarchical systems comprising multi-core chips and accelerator hardware, it is worthwhile pursuing both CRS and JDS as potentially promising approaches to storing general sparse matrices. In order to adapt the JDS format to the bandwidth-starved situation on current multi-core architectures, we have to balance the vectorization aspect against the potential of cache and register reuse. Blocking ("NBJDS") and outer loop unrolling ("NUJDS") techniques seem most promising in this respect, and can reduce the algorithmic balance of JDS considerably, so that it eventually becomes equal to CRS balance [12]. Standard blocking cuts all jagged diagonals into blocks of a given size. Instead of updating the complete result vector for each jagged diagonal as a whole, only the elements of the current block are processed for all jagged diagonals that have entries in this block, to the effect that the corresponding part of the result vector remains in cache. During a block update, accessing a new jagged diagonal requires skipping entries in the `val` and `col_idx` arrays. In the "RBJDS" scheme we avoid this non-contiguous access by storing all elements of a block consecutively. In the outer-loop-unrolled "NUJDS" scheme each element of the result vector is updated by two (or more) jagged diagonals simultaneously. If the unrolling factor equals the number of jagged diagonals, this variant is identical to the CRS scheme, aside from working in the permuted basis.

While all optimizations on the plain JDS format reduce memory traffic for writing the result, the discontinuous access to the input vector remains a performance bottleneck. Depending on the sparsity of the matrix and the anticipated number of required SpMVMs it may be beneficial to use a more sophisticated initial ordering of the elements. In the "SOJDS" scheme the elements in a row are sorted such that within each column of a block the input vector is accessed with stride one (or as close as possible). Figure 1 summarizes the ordering of the `val` and `col_idx` arrays for the different storage schemes.

3 Test Bed

All benchmark tests were performed on three test systems:

Woodcrest: A two-socket UMA-type node based on 3 GHz dual-core Intel Xeon 5160 processors (Core 2 architecture) with a 1333 MHz frontside bus. The two cores in a socket share a 4 MB L2 cache. Measured STREAM Triad bandwidth is about 6.5 GB/s.

Shanghai: A two-socket ccNUMA-type node based on quad-core AMD Opteron 2378 processors at 2.4 GHz with two-channel DDR2-800 memory. All four cores

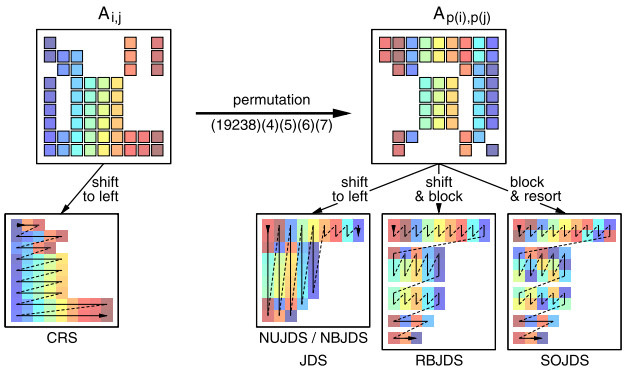

Fig. 1 Construction of the `val` and `col_idx` arrays for different storage schemes for a given matrix $A_{i,j}$. All JDS-flavored storage formats are based on the permuted matrix $A_{p(i),p(j)}$, with an additional sorting operation for SOJDS. The arrays within each sub-panel indicate the storage ordering of the elements. While the storage pattern for JDS, NUJDS and NBJDS is identical, these methods differ in the corresponding access pattern in the multiplication

on a chip share a 6 MB L3 cache. Measured STREAM Triad bandwidth is about 20 GB/s.

Nehalem: A two-socket ccNUMA-type node based on quad-core Intel "Core i7" (Nehalem) X5550 processors at 2.66 GHz with 3-channel DDR3-1333 memory. The four cores on a chip share an 8 MB L3 cache. Measured STREAM Triad bandwidth is about 35 GB/s.

For comparison we also obtained performance data on one node of HLRB-II.

4 Limitations of Serial Performance

As compared to the peak performance of a processor, the performance for a SpMVM is governed by memory access, and will usually be far less than 10% of peak. To single out different aspects of this performance reduction, in a first step we focus on basic operations that are the building blocks for the SpMVM in the following sections.

4.1 Basic Sparse Vector Operations

The inner loops of CRS and JDS differ in essence by the amount of data which is written in each iteration. While for CRS the result may be kept in a register and written to memory once after completing the inner loop, in the JDS case the whole result vector is written to memory `N_j` times. Concerning data to be read, the two building blocks are identical. Besides one (large) consecutive array, `val`, the elements of the

Table 1 Basic sparse vector operations, addition (ADD) and scalar product (SCP). Implementations are packed dense (PD), direct constant stride k (CS) and indirect addressing (IS/IR). For the latter we distinguish two cases: either constant stride in the index array for IS, that is `ind(i)=k*i` or random stride for IR. Then k is the mean stride

	ADD	SCP
PD	`s = s + B(i)`	`s = s + A(i) * B(i)`
CS	`s = s + B(k*i)`	`s = s + A(i) * B(k*i)`
IS / IR	`s = s + B(ind(i))`	`s = s + A(i) * B(ind(i))`

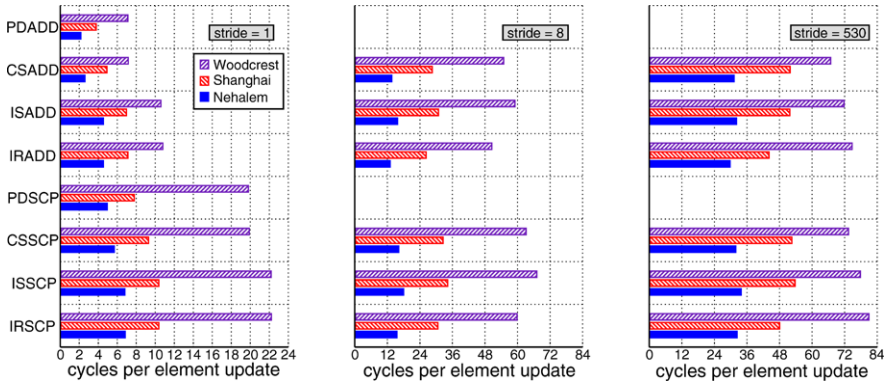

Fig. 2 Performance of the basic sparse operations given in Tab. 1 on different hardware architectures. Abstracting from the used hardware, performance is given in required cycles per non-zero element update. The considered strides correspond to dense packing, one entry per cache line and one entry per memory page. In the latter case we used stride $k = 530$ in order to circumvent the performance penalties by cache trashing effects for $k = 512$ [see Fig. 3(a)]

second array, `invec`, are determined indirectly by the indexing array `col_idx`. The access to `invec` comprises three performance penalties, which we will discuss by means of microbenchmarks. First, the indirect addressing of `invec` costs an additional 4 bytes per iteration for reading the (dense) indexing vector. Second, even if we replace the indirect addressing, `invec(col_idx(i))`, by a direct access with non-unit stride, `invec(k*i)`, unnecessary data is transfered since for each entry a whole cache line is read. Third, the indirect addressing may make efficient hardware prefetching almost impossible. Therefore, the relevant restriction for the performance might be latency, not memory bandwidth.

In an attempt to factor out the influence of loading the dense vector `val`, we abstract even more from the inner loop body by considering (indirect) sparse vector additions and scalar products. In Fig. 2 we give the required cycles per element update for the sparse scalar products and additions summarized in Tab. 1. Hereby the dense operations for stride $k = 1$ serve as a baseline. Note that the array lengths have been chosen such that the problem does not fit in any cache-level. Therefore the performance is limited by memory bandwidth, and the benefit of using packed SSE loads (`addpd`) instead of scalar ones (`addsd`) in the assembler code is marginal.[1]

[1] We do observe the expected factor of two for problem sizes which fit into cache.

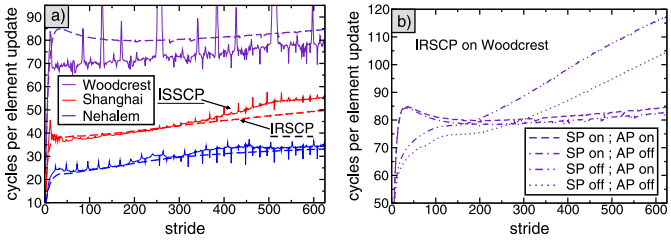

Fig. 3 Left panel: Performance of indirect scalar products for constant (ISSCP – thin solid lines) and random (IRSCP – thick dashed lines) strided access. Right panel: Performance for deactivated strided (SP) and adjacent cache line (AP) prefetcher for IRSCP on Woodcrest

The obtained performance using scalar loads is identical to the CSADD case which means that the additional multiplication by the stride k does not cause a performance penalty as long as the innermost loop is sufficiently unrolled. Indirect addressing causes an overhead of around 50% for ISADD, which is in accordance with the excess transferred data for the indexing array. Increasing the stride to $k = 8$ we observe the anticipated performance drop since for each element a whole cache line is read and seven eighths of it are useless in this case. This also holds for $k = 530$, but the number of translation lookaside buffer (TLB) misses is drastically larger for this stride, leading to a further penalty. A detailed comparison of the performance as a function of constant and random stride is given in Fig. 3(a) for ISSCP and IRSCP. For the IRSCP benchmark we generated a non-zero element for each entry of `invec` for which a drawn random number is smaller than a threshold given by the inverse mean stride $p = 1/k$.

Especially on the Woodcrest system, we encounter drastic performance drops for ISSCP at strides which are multiples of powers of two. On the other architectures these spikes also exist but are less severe. They can be attributed to an effective reduction of cache-size due to cache trashing. Randomizing the strides in IRSCP the distinct peaks disappear. A persisting feature is the bulge of reduced performance for $k < 25$, again most pronounced for the Woodcrest system. To pin down its origins, we deactivated the hardware prefetching mechanisms of the processor: (i) The adjacent cache line prefetch (AP), which loads two instead of one cache line on each miss, and (ii) the strided prefetcher (SP), which detects regular access patterns and tries to maintain continuous data streams, hiding latency. As shown in Fig. 3(b), turning off the strided prefetcher, the bulge vanishes and up to strides around $k \approx 200$ the performance stays better than for active prefetchers. However, this is not a general option since for larger strides SP is crucial for the performance and disabling it results in massive penalties. If we disable the adjacent cache line prefetcher, we observe an additional performance gain, either for activated or disabled SP, the latter being more pronounced. This can be attributed to the reduced memory traffic since now for each element only one cache line is fetched from memory instead of two. In general it is surprising how efficiently hardware prefetching works even with irregular strides.

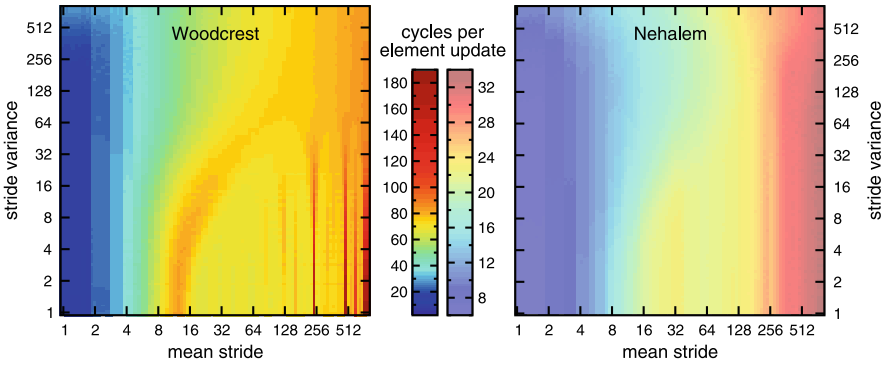

Fig. 4 Performance of IRSCP for an index vector with strides drawn from a Gaussian probability distribution with given mean value and variance of the stride

Up to now we have focused on inner loop kernels, for which `B` is accessed in a strictly monotonic order with random positive strides only. Clearly, the complete access pattern of the SpMVM kernels is not monotonic since negative strides arise when the first element of a row refers to an entry of `invec` that has a lower index than the last entry of the previous row. In order to investigate the influence of such backward jumps, we extend our IRSCP studies to Gaussian-distributed strides (Fig. 4). Fixing mean stride and variance of the distribution independently, we obtain a refined control over distribution characteristics allowing for negative strides provided the variance is large enough. The obtained results for the Woodcrest system underline the findings shown in Fig. 3(a). The peak structure in the ISSCP data is reproduced for small variances with only a minor effect on performance due to the stride jitter. For large stride variance the smooth performance variation agrees with the IRSCP data in Fig. 3(a) but the bulge observed there is missing. We therefore attribute the bulge to the peculiarities of the above stride distribution for which the grows with the mean stride as $k(k-1)$. Hence, the curves in Fig. 3(a) correspond to a cut along a tilted axis in Fig. 4, combining properties for small (large) variances at small (large) mean strides. On the Nehalem system the fine performance structures are missing and we only observe an overall decrease of the performance with increasing mean stride.

For Nehalem we also investigated the possibility of using non-temporal loads, which bypass the cache hierarchy when moving data from memory to registers. An instruction with the corresponding hint to the architecture (`movntdqa`) was introduced in SSE4.1 that is supported by the Nehalem processor. The non-temporal load may reduce the penalty connected with loading full cache lines of which only a fraction is actually used. However we could not see any effect, positive or negative, perhaps due to the hint-status of `movntdqa`. On current processors it behaves like a standard load.

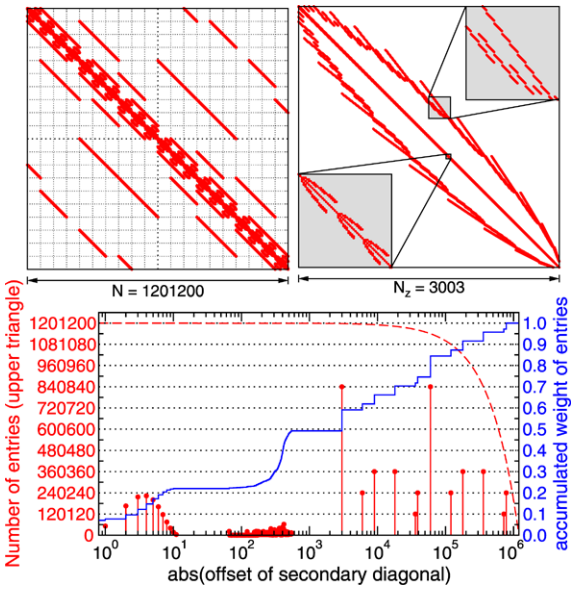

Fig. 5 Top left panel: Sparsity pattern of the Holstein-Hubbard Hamiltonian with dimension $N = 1201200$. Top right panel: detailed look at the matrix structure near the diagonal. This panel magnifies the matrix by a factor of 400, i.e. its linear size corresponds to $1/20$ of a dotted square in the left panel. In the further magnifying insets ($N_{zz} = 54,250$) individual matrix entries can be distinguished. Bottom panel: Compressed information on the matrix structure in terms of offset and occupation of secondary diagonals. The dashed red line gives the total (zero and non-zero) number of elements for each secondary diagonal

4.2 Resulting Performance for SpMVM

The low-level results from the previous section can be applied to a wide variety of sparse numerical problems. Since the sparsity pattern of a matrix influences the data access characteristics in an essential way, this investigation will focus on a special physical problem, which is characterized by a Holstein-Hubbard Hamiltonian [16]. Two basic distributions of non-zero elements are present in the corresponding sparse matrix: From the sparsity pattern in Fig. 5 we see that a considerable fraction of the matrix entries is concentrated in (rather dense) secondary diagonals. The remaining elements are scattered evenly over a band containing several hundred secondary diagonals, impeding the use of multi-diagonal storage schemes. Such a split structure is typical for electron-phonon systems. In addition, since the eigenvalues of a Hamiltonian are real-valued physical observables the corresponding matrix is Hermitian (symmetric in the real case). From the point of view of data transfer in the SpMVM this potentially allows for a further optimization which we do not investigate here. Tracing back the matrix properties to the characteristics of our previous benchmarks, the information contained in the sparsity pattern is too detailed. In an attempt to compress this information we show in the lower panel of Fig. 5 the num-

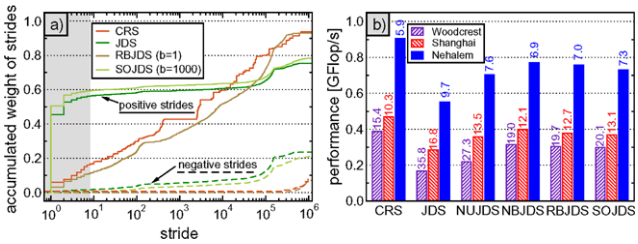

Fig. 6 Left panel: Distribution function of strides for the different storage schemes. The used block size for RBJDS (SOJDS) is 1 (1000). Solid (dashed) lines correspond to forward (backward) jumps in `invec`. Right panel: Serial performance for the complete SpMVM of the test matrix using the different kernels. For the blocked versions the block sizes have been chosen to optimize the performance (see Fig. 7). In addition to the performance in Flop/s the required cycles per element update are indicated for comparison with previous results

ber of non-zero elements as a function of their distance to the main diagonal together with the corresponding distribution function. From the latter we see that about 50% of the non-zero elements are contained in the twelve outermost secondary diagonals. Each of those is a potential candidate for special treatment by a dense storage scheme. Such hybrid implementations have been proposed in the literature [4], but we will not go into more detail on this aspect here.

The stride distribution in the SpMVM kernel is influenced by more aspects than just the secondary diagonal structure. A further important factor is the storage scheme [see Fig. 6(a)]. The stride distribution for CRS directly reflects the secondary diagonal structure, with negative offsets only once at the beginning of a new row. Since the matrix has on average 14 non-zero elements per row, the accumulated weight of backward jumps is around 7%. The positive strides are almost evenly distributed. The required permutation of the matrix for setting up the JDS format does not change this distribution significantly if the elements are still accessed row by row. This is the case for the RBJDS with block size $b = 1$. Increasing the block size the distribution gradually approaches the limiting case of JDS, corresponding to a block size of the matrix dimension. For the JDS almost 60% of the strides are smaller than 8 entries (64 bytes), enhancing cache line reuse. The drawback is a tripled amount of backward jumps. The structure of our test matrix leaves only limited potential for SOJDS and the optimized stride distribution does not differ significantly from JDS. Comparing the resulting performance of all schemes [Fig. 6(b)], the CRS outperforms all JDS-flavored variants. Interestingly, despite the improved memory access pattern, RBJDS and SOJDS cannot outperform NBJDS if an optimal block size is chosen in all cases. As expected, the benefit of the advanced blocked JDS formats is a wider range of suitable block sizes (see Fig. 7). This may be important with parallel execution if load balancing becomes an issue, which is however not the case for the considered test matrix.

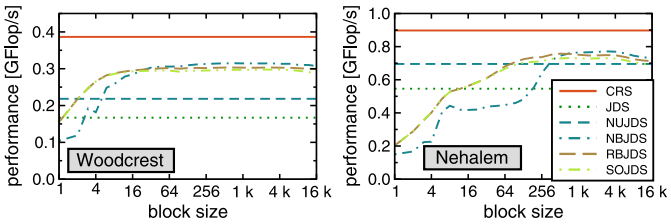

Fig. 7 Block size dependence of the serial performance for the SpMVM with the test matrix. For comparison also the performance for the schemes without blocking (CRS, JDS, NUJDS) is given. Data for Shanghai is not shown because the characteristics are very similar to Nehalem, although on a lower level

5 Shared-Memory Parallel SpMVM

In order to fully exploit the potential of multi-processor architectures, parallel execution is mandatory. In this work we consider OpenMP parallelization of our SpMVM code on the systems described in Sec. 3. Since all current commodity HPC systems are of the ccNUMA type, we distinguish clearly between intra-socket and inter-socket scaling behavior. Of course, pinning all threads to the physical cores is crucial for obtaining reliable performance data, because NUMA placement and bus contention effects are important factors for the performance of all bandwidth bound algorithms. While for a serial process this can be accomplished easily using the `taskset` command, pinning OpenMP threads is more involved. We implemented thread affinity by overloading the pthread library to ensure correct pinning of each thread created during program execution [2].

5.1 Intra-Socket Performance

In current multi-core designs like Nehalem and Shanghai, a single thread is not able to saturate the memory bandwidth of a socket. The reasons for this are complex [15] and beyond the scope of this contribution. For memory bound applications like the SpMVM it is therefore obligatory to use several threads per socket. In order to minimize parallelization overhead, it is however desirable to use the smallest number of threads that saturate the memory bandwidth. In Fig. 8 we compare the performance of the SpMVM for our test matrix versus the number of OpenMP threads on different architectures. While for Nehalem and Shanghai the performance scales up to three threads per socket, using a second thread per socket on Woodcrest results in no performance gain. As expected from the STREAM bandwidth numbers (see Sect. 3) Nehalem outperforms Shanghai by a factor of roughly two. The more recent AMD Istanbul processor has the same memory subsystem as Shanghai and is thus not expected to achieve better performance per socket.

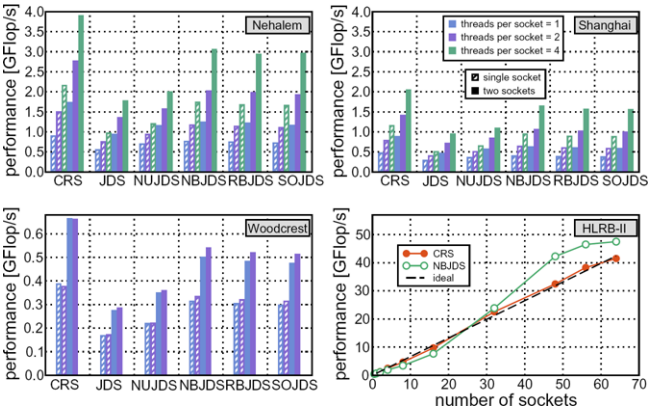

Fig. 8 Performance of OpenMP parallel SpMVM kernels for one and two sockets versus number of threads per socket. Data is based on static scheduling and a block size of 1000. In the lower right panel the measured (ideal) speedup on a HLRB II node is given by solid (dashed) lines. Two threads per node were used on this machine, and the CRS and NBJDS baselines were identical

5.2 Inter-Socket Performance

Using two sockets, performance on Woodcrest increases by about 60%, which is expected because this UMA-type FSB-based node architecture is known to show scalability problems [11]. The ccNUMA systems scale distinctly better, provided that proper page placement is implemented by employing first touch initialization. Apart from minor deficiencies which can be attributed to the access to `invec` our data reflects this feature (see Fig. 8). Placement of the input vector is imperfect by design as non-local accesses from other NUMA domains cannot be avoided. This is a property that strongly depends on the matrix structure, of course.

Choosing the correct loop scheduling is vital on ccNUMA nodes. Because of first touch placement, dynamic and guided scheduling are usually ruled out, except for strongly load imbalanced problems. In Fig. 9 we investigate the impact of different OpenMP schedulings and chunk sizes using eight threads on the Nehalem system. As expected, best overall performance is achieved with static scheduling and the CRS format. Small chunk sizes that lead to data blocks smaller than a memory page are hazardous for performance since page placement becomes random. For all JDS flavors, using large block sizes together with large OpenMP chunks leads to underutilized threads because the number of chunks becomes too small (top right sector in all blocked JDS panels).

In summary, the superior performance of the CRS is maintained even for dual socket ccNUMA systems. For the considered Hamiltonian the possible benefits of load balancing by guided or dynamic loop scheduling does not outweigh the penalties from disrupted NUMA locality.

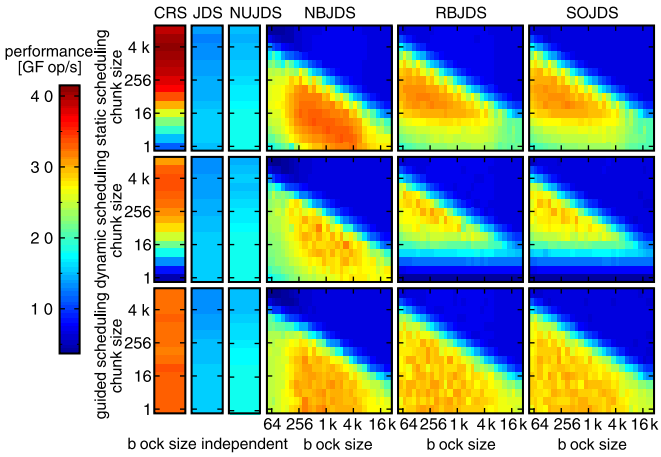

Fig. 9 Performance of the SpMVM kernels as a function of block size, scheduling policy and scheduling chunk size for 2×4 threads on Nehalem

5.3 HLRB-II Scalability

For comparison we performed scaling runs on a single node in the HLRB-II bandwidth partition (two cores per locality domain). In order to put the results shown in Fig. 8 into perspective, we must note that the whole matrix fits into the aggregated L3 caches of only 12 sockets. Thus we expect two effects: (i) superlinear speedup might be observed and (ii) JDS should have a significant performance advantage on many threads, because of the negative performance impact of short loops (as in the CRS kernel) on the Itanium2. Although CRS is slightly faster on 32 sockets or less, NBJDS dominates for large thread counts, which confirms our predictions.

6 Conclusion and Outlook

We analyzed the achievable performance of different SpMVM implementations on current multi-core multi-socket architectures. By a series of microbenchmarks, the basic operations of the SpMVM have been investigated in detail. The hardware prefetching mechanism of current x86 processors have been shown to work unexpectedly well, even for moderately random data access patterns.

Several SpMVM storage schemes have been benchmarked with respect to their serial and OpenMP-parallel performance, using a Hamiltonian matrix from solid state physics. In summary, the CRS format outperforms the best cache-blocked JDS schemes by at least 20%. Nevertheless we believe that JDS deserves further attention due to its long inner loop lengths, which might be advantageous on future processor and accelerator designs.

Future work will encompass a hardware counter analysis of SpMVM in order to get even more detailed information on its data access requirements. In view of massively parallel systems distributed memory and hybrid implementations will be thoroughly investigated.

Acknowledgements We thank J. Treibig and G. Wellein for valuable discussion and acknowledge financial support from KONWIHR II (project HQS@HPC II). We are also indebted to LRZ München for providing access to HLRB-II.

References

1. URL http://www.mcs.anl.gov/petsc/petsc-as/
2. URL http://code.google.com/p/likwid
3. Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H.: Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide. SIAM, Philadelphia (2000)
4. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of Supercomputing Conference 2009 (2009). To be published
5. Brandt, A.: Guide to multigrid development. Lect. Notes Math. **960**, 220 (1981)
6. Briggs, W.L., Henson, V.E., McCormick, S.F.: A Multigrid Tutorial. SIAM, Philadelphia (2000)
7. Demmel, J.: Applied Numerical Linear Algebra. SIAM, Philadelphia (1997)
8. Demmel, J.W., Gilbert, J.R., Li, X.S.: An asynchronous parallel supernodal algorithm for sparse gaussian elimination. SIAM J. Matrix Analysis and Applications **20**(4), 915–952 (1999)
9. Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., Koziris, N.: Performance evaluation of the sparse matrix-vector multiplication on modern architectures. J. Supercomputing (2008). DOI 10.1007/s11227-008-0251-8
10. Greenbaum, A.: Iterative Methods for Solving Linear Systems. SIAM, Philadelphia (1997)
11. Hager, G., Stengel, H., Zeiser, T., Wellein, G.: Rzbench: Performance evaluation of current hpc architectures using low-level and application benchmarks. In: S. Wagner, M. Steinmetz, A. Bode, M. Brehm (eds.) High Performance Computing in Science and Engineering, Garching/Munich 2007. Transactions of the Third Joint HLRB and KONWIHR Status and Result Workshop, Dec 3-4, 2007, LRZ Garching, pp. 485–501 (2009). arXiv:0712.3389
12. Hager, G., Wellein, G.: Optimization techniques for modern high performance computers. Lect. Notes Phys. **739**, 731 (2008)
13. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2 edn. SIAM, Philadelphia (2003). URL http://www-users.cs.umn.edu/~saad/books.html
14. Schönauer, W.: Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers. Self-edition (2000). URL http://www.rz.uni-karlsruhe.de/~rx03/book
15. Treibig, J., Hager, G., Wellein, G.: Complexities of performance prediction for bandwidth-limited loop kernels on multi-core architectures. In: High Performance Computing in Science and Engineering, Garching/Munich 2009, p. 3–12 (2010)
16. Wellein, G., Röder, H., Fehske, H.: Polarons and bipolarons in strongly interacting electron-phonon systems. Phys. Rev. B **53**, 9666 (1996)
17. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplications on emerging multicore platforms. Parallel Comput. **35**, 178 (2009)