GHOST: BUILDING BLOCKS FOR HIGH PERFORMANCE SPARSE LINEAR ALGEBRA ON HETEROGENEOUS SYSTEMS *

MORITZ KREUTZER[†], JONAS THIES[‡], MELVEN RÖHRIG-ZÖLLNER[‡], ANDREAS PIEPER[§], FAISAL SHAHZAD[†], MARTIN GALGON[¶], ACHIM BASERMANN[‡], HOLGER FEHSKE[§], GEORG HAGER[†], AND GERHARD WELLEIN[†]

Abstract. While many of the architectural details of future exascale-class high performance computer systems are still a matter of intense research, there appears to be a general consensus that they will be strongly heterogeneous, featuring "standard" as well as "accelerated" resources. Today, such resources are available as multicore processors, graphics processing units (GPUs), and other accelerators such as the Intel Xeon Phi. Any software infrastructure that claims usefulness for such environments must be able to meet their inherent challenges: massive multi-level parallelism, topology, asynchronicity, and abstraction. The "General, Hybrid, and Optimized Sparse Toolkit" (GHOST) is a collection of building blocks that targets algorithms dealing with sparse matrix representations on current and future large-scale systems. It implements the "MPI+X" paradigm, has a pure C interface, and provides hybrid-parallel numerical kernels, intelligent resource management, and truly heterogeneous parallelism for multicore CPUs, Nvidia GPUs, and the Intel Xeon Phi. We describe the details of its design with respect to the challenges posed by modern heterogeneous supercomputers and recent algorithmic developments. Implementation details which are indispensable for achieving high efficiency are pointed out and their necessity is justified by performance measurements or predictions based on performance models. The library code and several applications are available as open source. We also provide instructions on how to make use of GHOST in existing software packages, together with a case study which demonstrating the applicability and performance of GHOST as a component within a larger software stack.

Key words. sparse linear algebra, heterogeneous computing, software library, task parallelism, large scale computing

1. Introduction and related work.

1.1. Sparse solvers on heterogeneous hardware. Users of modern supercomputers are facing several obstacles on their way to highly efficient software. Out of those, probably the most prominent is the ever increasing level of parallelism in hardware architectures. Increasing the parallelism on the chips – both in terms of the number of cores as well as inside the core itself – is currently the only way to increase the maximum performance while keeping the energy consumption at a reasonable level. The parallelization of hardware architectures peaks in the use of accelerators, coprocessors, or graphics processing units (GPUs) for general purpose computations. Those devices trade off core sophistication against a very high core count, achieving an extremely high level of parallelism with unmatched peak floating point performance per Watt. Today, 15% of all TOP500 [50] systems are heterogeneous and the accelerators in those installations account for more than a third of the entire aggregated TOP500 performance. This evolution has led to the emergence of a large scientific community dealing with various aspects of accelerator programming and a considerable amount of accelerator-enabled software packages. However, "heterogeneous software" often means "accelerator software". A fact which is frequently not being considered is that also the CPU part of a heterogeneous system can contribute significantly to a program's performance. On the

[†]ERLANGEN REGIONAL COMPUTING CENTER, FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG, 91058 ERLANGEN, GERMANY

[‡]GERMAN AEROSPACE CENTER (DLR), SIMULATION AND SOFTWARE TECHNOLOGY, 51147 KÖLN, GERMANY

[§]INSTITUTE OF PHYSICS, ERNST-MORITZ-ARNDT-UNIVERSITÄT GREIFSWALD, 17489 GREIFSWALD, GERMANY

[¶]BERGISCHE UNIVERSITÄT WUPPERTAL, 42097 WUPPERTAL, GERMANY

*This work was supported by the German Research Foundation (DFG) through the Priority Programs 1648 "Software for Exascale Computing" (SPPEXA) under project ESSEX ("Equipping Sparse Solvers for Exascale").

other hand, even in simpler CPU-only machines, properties like memory hierarchy, ccNUMA effects and thread affinity play an important role in high performance code development. The addition of accelerators further increases the level of complexity in the system topology. At the same time, the world's largest compute clusters exhibit a steady growth in terms of cores and nodes as a result of continuously increasing requirements from application users. This poses challenges to algorithms and software in terms of scalability.

There is a wide range of applications which require computation with very large sparse matrices. The development of sparse linear and eigenvalue solvers that achieve extreme parallelism therefore remains an important field of research. One example is the analysis of novel materials like graphene [37] and topological insulators [45] in the field of solid state physics, which is a driving force in the GHOST development within the ESSEX project[1]. These eigenvalue problems require the computational power of full petascale systems for many hours, so it is crucial to achieve optimal performance at all levels.

Recent work in the area of sparse matrix algorithms can roughly be subdivided into three categories. Methods that increase computational intensity (i.e. reduce/avoid communication), methods that hide communication by overlapping it with computation, and fully asynchronous algorithms. To the first category belong, e.g., block Krylov methods and the communication avoiding GMRES (CA-GMRES [30]) method, which require optimized block vector kernels. The second category includes the pipelined CG and GMRES methods [15]. An example for the third category is the asynchronous ILU preconditioner by Chow and Patel [9]. Methods from the latter two categories benefit from an easy-to-use tasking model that delivers high performance. The novel implementation of ILU methods in [9] replaces the poorly scaling forward/backward substitution by a matrix polynomial, increasing the performance requirements of the sparse matrix-vector multiplication in preconditioned Krylov methods.

1.2. Related work. There is a large interest in efficient heterogeneous software driven by the developments in modern supercomputer architectures described above. Many efforts follow a task-parallel approach, which strives to map a heterogeneous workload to heterogeneous hardware in an efficient way. The most prominent software package implementing task-parallel heterogeneous execution is MAGMA [26]. A major drawback of MAGMA is the absence of built-in MPI support, i.e., users have to implement MPI parallelism around MAGMA on their own. Under the hood, MAGMA uses the StarPU runtime system as proposed by Augnoett et al. [2] for automatic task-based work scheduling on heterogeneous systems. Another significant attempt towards heterogeneous software is ViennaCL [42]. Being based on CUDA and OpenCL, this software package can execute the same code on a wide range of compute architectures. However, concurrent use of different architectures for a single operation is not supported. Besides, ViennaCL has limited support for complex numbers, which is problematic for many applications. The same applies to the C++ framework LAMA [24], a library based on MPI+OpenMP/CUDA with special focus on large sparse matrices. PETSc [5] is an MPI-parallel library for the scalable solution of scientific applications modeled by partial differential equations. Its intended programming model is pure MPI, with MPI+X support for GPUs ('X' being CUDA or ViennaCL) and some limited support for threading. It also lacks support of heterogeneous computation of single operations. Another library containing sparse iterative solvers and preconditioners is PARALUTION [35]. However, the multi-node and complex number support is restricted to the non-free version of this software. The Trilinos packages Kokkos and Tpetra [3] implement an MPI+X approach similar to the one used in GHOST. Being implemented in C++, they clearly separate the MPI level

[1] `http://http://blogs.fau.de/essex`

(Tpetra package) from the node level (Kokkos package), whereas GHOST can benefit from tighter integration for, e.g., improved asynchronous MPI communication (cf. subsection 4.2). In subsection 6.1 we will provide a performance comparison of GHOST vs. Trilinos for an eigenvalue solver.

While all of these libraries certainly improve the accessibility of heterogeneous hardware to a wide range of applications, they do not fit our purpose of extreme scale eigenvalue computations in an optimal way. In particular, we believe that a single library for building blocks integrating well-tuned kernels, communication on all levels and good performance on heterogeneous systems 'out of the box' is key to satisfying the needs of scientists who are trying to tackle problems at the edge of 'what can be done'.

1.3. Contribution. In this work, we present the software package GHOST (*General, Hybrid and Optimized Sparse Toolkit*). As summarized in Section 1.2, there is a range of efforts towards efficient sparse linear algebra on heterogeneous hardware driven by modern hardware architectural developments. GHOST can be classified as an approach towards a highly scalable, and truly heterogeneous sparse linear algebra toolkit with a key target in the development process being optimal performance on all parts of heterogeneous systems. In close collaboration with experts from the application side we focus on a few key operations often needed in sparse eigenvalue solvers and provide highly optimized and performance-engineered implementations for those. We show that disruptive changes of data structures may be necessary to achieve efficiency on modern CPUs and accelerators featuring wide single instruction multiple data (SIMD) units and multiple cores.

One may argue about whether performance should be the primary goal in a CS&E software library and whether it is worth the effort to optimize a few core operations for a two-digit percentage gain in performance. Our efforts are targeted at large-scale supercomputers in the petaflop range and beyond, and computing time is a valuable resource there. Even a performance gain below an order of magnitude can become significant in terms of time, energy, and money spent on the large scale. Needless to say, the kernels we provide can be used on smaller clusters or single workstations as well. GHOST does not give up generality or extensibility for this purpose, rather we aim to provide performance-optimized (guided by performance models) kernels for some commonly used algorithms (e. g. the kernel polynomial method [23], the block Jacobi-Davidson method [41] or Chebyshev filter diagonalization [38]). The successful implementations of these methods (which are very popular in fields like material physics and quantum chemistry) will serve as blueprints for other techniques such as advanced preconditioners needed in other CS&E disciplines. In the application areas we consider right now, methods such as incomplete factorization or multigrid can usually not be applied straightforwardly. The matrices that appear may not have an interpretation as physical quantities discretized on a mesh, they may be completely indefinite, and they may have relatively small diagonal entries and/or random elements [12].

A key feature of GHOST is the transparency to the user when it comes to heterogeneous execution. In contrast to other heterogeneous software packages (cf. Section 1.2), GHOST uses a data-parallel approach for work distribution among compute devices. While a task-parallel approach is well-suited for workloads with complex dependency graphs, the data-parallel approach used by GHOST may be favorable for uniform workloads (i.e., algorithms where all parts have similar resource requirements) or algorithms where an efficient task-parallel implementation is unfeasible. On the process level, GHOST's tasking mechanism still allows for flexible work distribution beyond pure data parallelism.

GHOST unifies optimized low-level kernels whose development process is being guided by performance modelling into a high-level toolkit which allows resource-efficient execution on modern heterogeneous hardware. Note that for uniform workloads, performance models

Alias	Model	Clock (MHz)	SIMD (Bytes)	Cores/ SMX	b (GB/s)	P^{peak} (Gflop/s)
CPU	Intel Xeon E5-2660 v2	2200	32	10	50	176
GPU	Nvidia Tesla K20m	706	128...512*	13	150	1174
PHI	Intel Xeon Phi 5110P	1050	64	60	150	1008

Table 1.1: Relevant properties of all architectures used in this paper. The attainable memory bandwidth as measured with the STREAM [28] benchmark is denoted by b and P^{peak} is the theoretical peak floating point performance. Turbo mode was activated on the CPU and the GPU was configured with ECC enabled.
*: SIMD processing is done by 32 threads. Hence, the SIMD width in bytes depends on the data type in use: 128 bytes is valid for 4-byte (single precision floating point) data while 512 bytes corresponds to complex double precision data.

like the roofline model [53] are a suitable tool to check an implementation's efficiency. In recent work, GHOST has proven to scale up to the petaflop level, extending the scaling studies presented in [23]. A list of challenges we are addressing specifically and the corresponding sections in this paper can be given as follows:

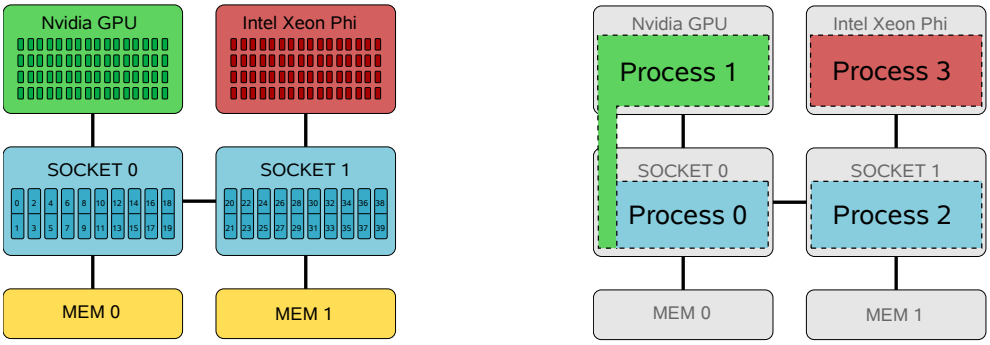
(i) Emerging asynchronous sparse solver algorithms require a light-weight, affinity-aware and threading-friendly task-based execution model. In this context, the high relevance of OpenMP should be noted, which requires the tasking model to be compatible to OpenMP-parallel codes. See Section 4.2.
(ii) Existing software rarely uses all components of heterogeneous systems in an efficient manner. See Sections 4.1 and 5.1.
(iii) The potential performance of compute libraries is often limited by the requirement of high generality, leading to a lack of application-specific kernels. See Section 5.3.
(iv) The possibilities for application developers to feed their knowledge into compute libraries for higher performance are often limited. See Section 5.4.
(v) The applicability of optimization techniques like vector blocking is often limited due to restrictions in existing data structures. Fundamental changes to data structures are often hard to integrate in existing software packages. See Sections 5.1 and 5.2.

GHOST is available as a BSD-licensed open source download [14]. Along with it, a list of sample application based on GHOST (e.g., a Conjugate Gradient solver and a Lanczos eigensolver) can be downloaded. On top of that, the iterative solver package and sister project of GHOST named PHIST [36] can use GHOST to execute more sophisticated algorithms like, e.g., the block Jacobi-Davidson eigensolver as described in [41], and blocked versions of the MinRes and GMRES linear solvers.

1.4. Testbed. All experiments in this paper have been conducted on the Emmy[2] cluster located at the Erlangen Regional Computing Center. Table 1.1 summarizes the hardware components used in this cluster. The Intel C/C++ compiler in version 14 and CUDA in version 6.5 have been used for compilation. Intel MKL 11.1 was used as the BLAS library on the CPU.

2. Design principles. In this section, fundamental design decisions of the GHOST development are discussed and justified. This includes the support of certain hardware archi-

[2] http://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/systeme/emmy-cluster.shtml



(a) Heterogeneous node

(b) Process placement

Fig. 2.1: Heterogeneous compute node and sensible process placement as suggested by GHOST.

tectures as well as fundamental parallelization paradigms.

2.1. Supported architectures and programming models. Many modern compute clusters comprise heterogeneous nodes. Usually, such nodes consist of one or more multi-core CPUs and one or more accelerator cards. In the TOP500 [50] systems of November 2014, 96% of all accelerator performance stems from NVIDIA Fermi/Kepler GPUs or Intel Xeon Phi (to be called "PHI") coprocessors. Hence, we decided to limit the accelerator support in GHOST to those two architectures. Instead of implementing support for any kind of hardware architecture, our primary goal is to stick to the dominant platforms and to develop properly performance-engineered code for those.

Although GPUs and PHIs share the name *accelerators*, there are significant differences in how those devices are operated. GPUs can only be driven in *accelerator mode*, i.e., data transfers and compute kernels must be launched explicitly from a main program running on a host CPU. The PHI can be operated in accelerator mode, too. However, in addition to this, the PHI can also be driven in *native mode*, i.e., in the same way as a multicore CPU would be used. In GHOST only native execution on the PHI is supported, i.e., the PHI hosts its own process. Hence, the PHI can be considered as a CPU node on its own. With regard to the PHI as a multi-(many-)core CPU, it has to be taken into account that serial code may run at very low performance due to its very simple core architecture.

2.2. Parallelism in GHOST. For illustration of the principles, we consider a heterogeneous node as shown in Fig. 2.1a. This node contains two multicore CPU sockets with ten cores and two-way hyper-threading each. In total, there are 20 hardware threads or processing units (PUs) available per socket. In addition to that, one GPU and one PHI are attached to the node as accelerators. Note that a node with two different accelerator architectures is unlikely to be installed in a production system.

In terms of parallelization, GHOST implements the "MPI+X" paradigm, i.e., coarse-grained parallelism is done by means of MPI accompanied with fine-grained and device-specific parallelization mechanisms ("X"). One may certainly omit the "X" part and go with plain MPI altogether if the hardware can be efficiently utilized in this way; the plain fact that modern hardware exhibits complex topologies does not mean that a hybrid programming model is required in all cases, and it may sometimes even be counterproductive. However,

interesting opportunities in terms of load balancing, additional levels of parallelism, communication hiding, etc., arise from combining MPI with a threading model [40]. This is why GHOST supports OpenMP for the "X" component on CPUs. Further down the hardware hierarchy, implicit vectorization by compiler-friendly code and pragmas as well as explicit vectorization using Single Instruction Multiple Data (SIMD) intrinsics provide efficient single-threaded code. On Nvidia GPUs, CUDA is used as the "X" parallelization layer.

In general, parallelism in compute applications can be categorized into *data* and *task* parallelism. The term *data parallelism* describes a number of workers operating on the same data set, each having assigned a certain amount of work. The term *task parallelism* describes workers working on independent tasks at the same time. GHOST implements both data (between processes) as described in Section 4.1 and task parallelism (inside a process) as analyzed in Section 4.2.

In many cases, algorithms from sparse linear algebra are centered around a single and potentially large sparse system matrix. Hence, the distribution of work in GHOST is done in a matrix-centered way. More precisely, the system matrix is distributed row-wise across the MPI processes. The amount of work per process can either be expressed by the number of rows or the number of nonzero elements. Details on the implementation are given in Section 4.1.

3. Available data structures. There are two major data structures in GHOST: Sparse matrices (`ghost_sparsemat_t`) and dense matrices (`ghost_densemat_t`). Dense vectors are represented as dense matrices with a single column or row. Both data structures implement a row-wise distribution among MPI processes. We do not support 2D partitionings of these data structures or direct conversion routines between them, but this may be added in the future.

3.1. Sparse matrices. GHOST supports the SELL-C-σ sparse matrix storage format as introduced in [23]. Note that this is not necessarily a restriction, as the well-known CRS storage format can be expressed as SELL-1-1. Further special cases of SELL-C-σ will be listed in subsection 5.1. More details on sparse matrix storage are given in Section 5.1. As mentioned in Section 2.2, the sparse system matrix is the central data structure in GHOST.

A significant performance bottleneck for highly scalable sparse solvers may be the generation of the system matrix. In GHOST this matrix can be stored in a file, either in the Matrix Market [27] format or a binary format which resembles the CRS data format. However, the scalability of this approach is intrinsically limited. The preferred method of matrix construction in GHOST is via a callback function provided by the user, which allows to construct a matrix row by row. The function must have the following signature:

```
int getrow(ghost_gidx_t row, ghost_lidx_t *len, ghost_gidx_t *col, void *val, void *arg);
```

GHOST passes the global matrix row to the function. The user should then store the number of non-zeros of this row in the argument `rowlen` and the column indices and values of the non-zeros in `col` and `val`. Any further arguments can be passed in `arg`. The maximum number of non-zeros must be set in advance such that GHOST can reserve enough space for the `col` and `val` arrays.

There are several reasons which make it necessary to permute rows of the sparse system matrix. A global (inter-process) permutation of matrix rows can be applied in order to minimize the communication volume of, e.g., the sparse matrix vector multiplication (SpMV) kernel and to enforce more cache-friendly memory access patterns. Currently, GHOST can be linked against PT-SCOTCH [8] for this purpose. A matrix' row lengths and column indices are passed to PT-SCOTCH which results in a permutation vector on each process containing global indices. Afterwards, the matrix is assembled on each process according to the global

permutation. Our experiments revealed that this approach is limited in terms of scalability. For that reason, we are going to include support for more global permutation schemes that improve communication reduction in future work, such as the parallel hypergraph partitioner as implemented in Zoltan [11].

In addition to the global permutation, a local (intra-process) permutation can be applied, e.g., to minimize the storage overhead of the SELL-C-σ sparse matrix format (cf. Section 5.1. Another potential reason for a local matrix permutation is row coloring. GHOST has the possibility to permute a sparse matrix according to a coloring scheme obtained from ColPack [13]. This kind of re-ordering may be necessary for the parallelization of, e.g., the Kaczmarz [20] algorithm or a Gauß-Seidel smoother as present in the HPCG benchmark.

Note that an application-based permutation, e.g., by optimizing the numbering of nodes in a mesh-based problem, usually leads to better overall performance and should be preferred over an a posteriori permutation, e.g., with PT-SCOTCH. In GHOST the former can be achieved by the user by a sensible implementation of the matrix construction via the callback interface.

3.2. Dense matrices. GHOST is a framework for sparse linear algebra. Dense matrices are mainly occurring as dense vectors (dense matrices with a single column) or blocks of dense vectors (to be referred to as *block vectors*). Section 5.2 will cover the aspect of block vectors in more detail. Block vectors can be considered as tall and skinny dense matrices, i.e., dense matrices with a large row count (in the dimension of the sparse system matrix) but relatively few (at most a few hundred) columns. Furthermore, a `ghost_densemat_t` can be used to represent small local or replicated matrices, e.g. the result of an inner product of two (distributed) block vectors.

Instead of allocating its own memory, a dense matrix can also be created as a *view* of another dense matrix or a view of arbitrary data in memory. This makes it easily possible to let a function work on a sub-matrix or a subset of vectors in a larger block vector without having to copy any data. Additionally, by viewing "raw" data in memory it is possible to integrate GHOST into existing code (cf. Section 6). A potential disadvantage of using non-GHOST data structures is the violation of data alignment which may result in a performance loss. GHOST implements different kinds of views, as shown in Fig. 3.1. In general, compact views allow vectorized computation with the matrix data. This is not the case for scattered views due to the "gaps" in memory layout in the leading dimension caused by columns not included in the view. In this case, it may be favorable to create a compact clone of the scattered view before executing the computation.

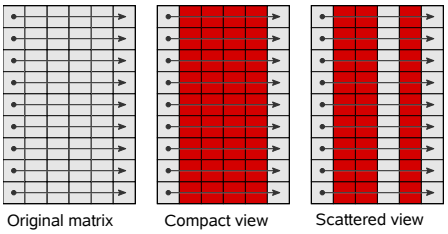


Fig. 3.1: Views of a dense matrix.

Dense matrices can be chosen to be stored in a (locally) row- or column-major manner. In many cases, row-major storage (which corresponds to interleaved storage of block vectors) yields better performance and should be preferred over column-major storage (cf. Section 5.2). On the other hand, column-major storage may be required for easy integration of GHOST into existing software. GHOST offers mechanisms to change the storage layout either in-place or out-of-place, while copying a block vector.

4. Runtime features. In this section we describe runtime features which are deeply woven into the software architecture and constitute GHOST's unique feature set. In contrast to

the so-called *performance* features which will be introduced in Section 5, they are fundamentally built into the library and hard to apply to other approaches.

4.1. Transparent and data-parallel heterogeneous execution. The distribution of work among the heterogeneous components is done on a per-process basis where each process (MPI rank) is bound to a fixed set of PUs. This allows flexible scaling and adaption to various kinds of heterogeneous systems. The sets of PUs on a single node are disjoint, i.e., the compute resources are exclusively available to a process. For the example node shown in Fig. 2.1a, the minimum amount of processes for heterogeneous execution on the full node is three.

Application developers are frequently confused by the ccNUMA memory structure of modern compute nodes and how to handle it to avoid performance penalties. Although the required programming strategies are textbook knowledge today, establishing perfect local memory access may be tricky if a multithreaded process spans multiple ccNUMA domains even if proper thread-core affinity is in place and parallel first-touch initialization is performed [17]. A simple way to avoid ccNUMA problems is to create one process per multicore CPU socket, which would result in a process count of four as illustrated for the example node in Fig. 2.1b. Processes 0 and 2 cover one CPU socket each. Process 1 drives the GPU. As this has to be done in accelerator mode, this process also occupies one core of the host system. Note that this core is located on the socket whose PCI Express bus the GPU is attached to and thus has to be subtracted from Process 0's CPU set. Process 3 is used for the PHI. The process can directly be located on the accelerator which is used in native mode, i.e., no host resources are used for driving the PHI.

For each numerical function in the GHOST library, implementations for the different architectures are present. However, the choice of the specific implementation of a kernel does not have to be made by the user. Consequently, in almost all usage cases, no changes to the code are necessary when switching between different hardware architectures. An exception of this rule is, e.g., the creation of a dense matrix view from plain data: If the dense matrix is located on a GPU, the plain data must be valid GPU memory.

An intrinsic property of heterogeneous systems is that the components differ in terms of performance. For efficient heterogeneous execution it is important that the performance differences get reflected in the work distribution. In GHOST the underlying sparse system matrix gets divided on a row-wise basis among all processes. For example, if component A is expected to have twice the performance of component B, process A will get assigned a twice as large chunk (either in terms of rows or in terms of nonzero elements) of the system matrix as process B. Figure 4.1 illustrates the row-wise distribution of a sparse matrix among the example processes shown in Fig. 2.1b. As the performance of sparse solvers is often bound by main memory bandwidth, the device-specific maximum attainable bandwidth, as given in Table 1.1, has been chosen as the work distribution criterion in this example. Note that an arbitrary work share for each process/architecture can easily be specified at runtime.

Internally, each process gets assigned a *type* which allows to define the compute platform used by an executable. Valid types are `CPU` and `GPU`. The type can be set explicitly at runtime either via API calls or by specifying an environment variable. If multiple processes are launched on a node containing CPUs and GPUs, the type gets selected automatically if not explicitly specified. In this case, Process 0 is always of type `CPU`, initially covering all CPUs in the node. Processes 1 to *N* are of type `GPU` where *N* is equal to the number of GPUs attached to the node. For each GPU process getting added to a node, a small CPU set (usually a single core) gets subtracted from Process 0's resources. If any more than (1 + "Number of GPUs") processes get placed on a node, the addition of any further processes causes a division of Process 0's CPU set into equally sized smaller CPU sets. A good number of processes

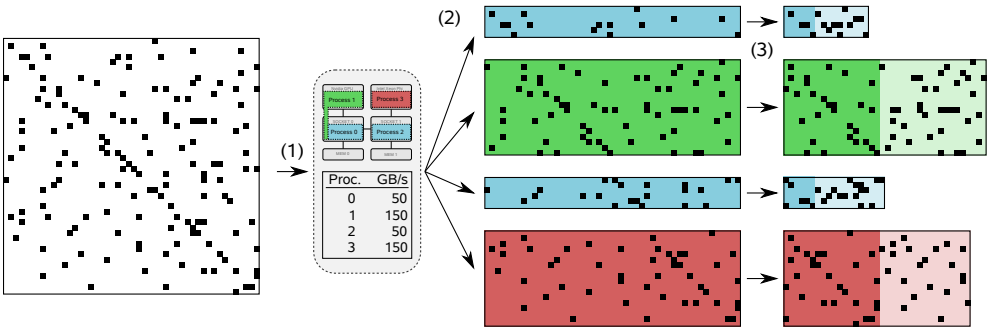


Fig. 4.1: Heterogeneous row-wise distribution of a sparse matrix. Step (1) is the determination of process weights according to the device's peak memory bandwidths. In step (2), a partial sparse matrix is created on each process. In order to avoid integer overflows and storage overhead in the communication buffers, the column indices of elements in the remote matrix part (pale colors) are compressed in step (3).

to put on a node is ("Number of CPUs" + "Number of GPUs"), which is an easy way to avoid NUMA locality problems by having one process per CPU socket.

In the following we demonstrate the heterogeneous execution capabilities on our example node using a simple program which measures the SpMV performance for a given matrix and storage format (downloadable from the GHOST website [14]). In this case, we used the `Janna/ML_Geer` matrix [3] (dimension $n = 1,504,002$, number of non-zeros $n_{nz} = 110,686,677$) stored in SELL-32-1. Performance will be reported in Gflop/s, with 1 Gflop/s corresponding to a minimum memory bandwidth of 6 GByte/s. This relation is founded on the minimum code balance of the SpMV kernel. If we want to perform computations on the CPU only and use one process per CPU socket, the type has to be set explicitly and a suitable number of processes has to be launched on the host:

```
> GHOST_TYPE=CPU mpiexec -nopin -np 2 -ppn 2 ./spmvbench -v -m ML_Geer.mtx -f SELL-32-1
...
[GHOST] PERFWARNING: The number of MPI processes (1) on this node is not optimal!
                     Suggested number:  3 (2 NUMA domains + 1 CUDA device)
...
[GHOST] PERFWARNING: There is 1 Xeon Phi in the set of active nodes but only 0 are used!
Region      | Calls |    P_max | P_skip10
-----------------------------------------
spmv (GF/s) |   100 | 1.64e+01 | 1.64e+01
```

The overall number of processes is set via the `-np` flag and the number of processes per host is set using `-ppn`. Note that automatic thread/process pinning by the MPI startup script has been suppressed by `-nopin`. This should always be done to avoid conflicts with GHOST's resource management. The maximum performance over all 100 runs is given in `P_max`. `P_skip10` shows the average performance over all but the first ten iterations. The performance warnings (omitted in the following listings) issued by GHOST indicate that the node is not used to its full extent. The suggested process count of three is in accordance with the knowledge about the node architecture; each node contains two CPU sockets and one GPU. The Intel PHI attached to this node has to be considered as a node on its own. The achieved performance of 16.4 Gflop/s matches the prediction of a simple roofline model for this algorithm and two CPU sockets. If the example program should use the GPU for computation,

[3] `http://www.cise.ufl.edu/research/sparse/matrices/Janna/ML_Geer.html`

the following command has to be invoked:

```
> GHOST_TYPE=GPU ./spmvbench -v -m ML_Geer.mtx -f SELL-32-1
...
Region      | Calls |    P_max | P_skip10
----------------------------------------
spmv (GF/s) |   100 | 2.28e+01 | 2.27e+01
```

From the single-device runs we can easily deduce that the GPU execution was 2.75 times as fast as the execution on a single CPU socket. In the current implementation the weight, i.e., the amount of work, assigned to each process in heterogeneous runs, has to be specified manually. Future work will include automatic weight selection based on micro-benchmarks and dynamic adaption of weights at runtime (cf. 7.2). Starting the example program using three processes and a work ratio between CPU and GPU of 1:2.75 yields the following:

```
> mpiexec -nopin -np 3 -ppn 3 ./spmvbench -v -m ML_Geer.mtx -f SELL-32-1 -w 1:2.75
...
[GHOST] PE0 INFO: Setting GHOST type to CPU.
[GHOST] PE1 INFO: Setting GHOST type to GPU.
[GHOST] PE2 INFO: Setting GHOST type to CPU.
...
Region      | Calls |    P_max | P_skip10
----------------------------------------
spmv (GF/s) |   100 | 3.11e+01 | 3.09e+01
```

The information log messages indicate that the process types have automatically been set as described above. The achieved performance is less than the accumulated single-device performances. This is due to the MPI communication of input vector data which is done in each SpMV iteration. For testing purposes, it is possible to suppress the communication by selecting an appropriate SpMV routine. Note that this does not give the correct result for the SpMV operation if the input vector data changes between successive iterations.

```
> mpiexec -nopin -np 3 -ppn 3 ./spmvbench -v -m ML_Geer.mtx -f SELL-32-1 -w 1:2.75 \
  -s nocomm
...
Region      | Calls |    P_max | P_skip10
----------------------------------------
spmv (GF/s) |   100 | 3.85e+01 | 3.73e+01
```

Now, the heterogeneous performance matches the sum of single-device performances. In order to include the node's PHI in the computation, the library and executable have to be compiled for the MIC architecture, resulting in an additional executable file `./spmvbench.mic`. For setting up heterogeneous execution using the Xeon Phi, the following has to be done:

```
> # Assemble machine file for three MPI ranks on the host and one on the PHI
> echo -e "$(hostname -s):3\n$(hostname -s)-mic0:1" > machinefile
> export I_MPI_MIC=1              # Enable MPI on the PHI
> export I_MPI_MIC_POSTFIX=.mic  # Specify the postfix of the MIC executable
```

Using all parts of the heterogeneous node for computing the communication-suppressed SpMV, the following is obtained:

```
> mpiexec -nopin -np 4 -machinefile machinefile \
  ./spmvbench -v -m ML_Geer.mtx -f SELL-32-1 -w 1:2.75:2.75 -s nocomm
...
Region      | Calls |    P_max | P_skip10
----------------------------------------
spmv (GF/s) |   100 | 5.64e+01 | 5.47e+01
```

The PHI got assigned the same share of work as the GPU. Note that there may still be optimization potential regarding the load balance. The total node performance adds up to approximately 55 Gflop/s, which indicates a good use of the aggregated memory bandwidth of all resources. If the SpMV function including communication is used, the optimal weights are slightly different, due to a higher communication effort on the GPU and PHI.

```
> mpiexec -nopin -np 4 -machinefile machinefile \
  ./spmvbench -v -m ML_Geer.mtx -f SELL-32-1 -w 1:2.3:2.1
...
Region      | Calls |    P_max | P_skip10
-----------------------------------------
spmv (GF/s) |   100 | 3.97e+01 | 3.88e+01
```

Note that the inclusion of the PHI barely leads to a performance benefit over the CPU+GPU run. This is due to the small amount of work on each device and an increasing dominance of communication over the slow PCI express bus as a result of this. The impact of communication may be reduced by matrix re-ordering (cf. Section 3.1) or more sophisticated communication mechanisms (using GPUdirect, pipelined communication, etc., cf. Section 7.2).

4.2. Affinity-aware resource management. Although GHOST follows a data-parallel approach across processes for heterogeneous execution, work is organized in tasks on the process level. The increasing asynchronicity of algorithms together with the necessity for sensible hardware affinity and the avoidance of resource conflicts constitute the need for a unit of work which is aware of the underlying hardware has to be used: a GHOST task. Affinity and resource management is implemented by means of the `hwloc` library [7] which is, besides a BLAS library, the only build dependency of GHOST.

There are existing solutions for task parallelism. Apart from the ones named in Section 1.2, OpenMP tasks or Intel's Threading Building Blocks can be mentioned here. However, a crucial requirement in the design phase of GHOST was to support affinity-aware OpenMP parallelism in user-defined tasks. As we could not find a light-weight existing solution which meets our requirements, we decided to implement an appropriate tasking model from scratch. For example, both Intel TBB and Cilk Plus warn about using those frameworks together with OpenMP in their user manuals. This is due to potential performance issues caused by core over-subscription. As OpenMP is in widespread use in scientific codes, this limitation disqualifies the integration of TBB and Cilk Plus tasks for many existing applications. Note that GHOST tasks lack a list of features compared to existing solutions, such as intelligent resolution of dependencies in complex scenarios. Yet, for most of our usage scenarios they work well enough. In our opinion, a holistic performance engineering approach is a key to optimal performance for complex scenarios. Thus, we decided to make the resource management a part of the GHOST library.

Generally speaking, a task's work can be an arbitrary user-defined function. OpenMP can be used inside a task function without having to worry about thread affinity or resource conflicts. The threads of a task are pinned to exclusive compute resources, if not specified otherwise. GHOST tasks are used in the library itself, e.g., for explicitly overlapping communication and computation in a parallel SpMV (see below). However, the mechanism is designed in a way that allows easy integration of the tasking capabilities also into user code. The user-relevant properties of a task are as follows:

```c
typedef struct {
    void * (*func) (void *); /* callback function where work is done */
    void * arg; /* arguments to the work function */
    void * ret; /* return value of the work function */
    struct ghost_task_t **depends; /* list of tasks on which this task depends */
    int ndepends; /* length of dependency list */
    int nthreads; /* number of threads for this task */
    int numanode; /* preferred NUMA node for this task */
    ghost_task_flags_t flags; /* flags to configure this task */
} ghost_task_t;
```

The user-defined callback function containing the task's work and its arguments have to be provided in the `func` and `arg` fields. The function's return value will be stored by GHOST

in `ret`. If the execution of this task depends on the completion of `ndepends` other tasks, those can be specified as a list of tasks called `depends`. The number of threads for the task has to be specified in `nthreads`. Usually, a suitable number of PUs will be reserved for this task. The field `numanode` specifies the preferred NUMA node of PUs reserved for this task, which is important for situations where different tasks work with the same data in main memory in a process which spans several NUMA nodes. In this situation, and assuming a NUMA first touch policy, one can enforce a task which works on specific data to run on the same NUMA node as the task which initialized this data. The `flags` can be a combination of the following:

```
typedef enum {
    GHOST_TASK_DEFAULT = 0; /* no special properties */
    GHOST_TASK_PRIO_HIGH = 1; /* enqueue task to head of the task queue */
    GHOST_TASK_NUMANODE_STRICT = 2; /* execute task _only_ on the given NUMA node */
    GHOST_TASK_NOT_ALLOW_CHILD = 4; /* disallow child tasks to use this task's PUs */
    GHOST_TASK_NOT_PIN = 8; /* neither reserve PUs for this task nor pin its threads */
} ghost_task_flags_t;
```

Figure 4.2 shows a simple flow chart of the execution of a GHOST application which uses task parallelism with a single task. In the initialization phase GHOST creates a number of *shepherd threads* which will immediately wait on a condition. As a task gets enqueued, this condition gets signalled which causes an arbitrary shepherd thread to be woken up. Note that the `enqueue()` function returns immediately. A decision whether the task can be executed is now made by the shepherd thread based on the task's resource requirements. If they are met, an initial OpenMP parallel region gets opened in which all threads of the task get pinned to their exclusive PU and each PU is set to *busy* in the `pumap`. The task's work function now is called by the shepherd thread and is executed in parallel to the user code which followed the call to `enqueue()`. Due to physical persistence of OpenMP threads, an OpenMP parallel region in the task function will be executed by the same threads as those which have been pinned by GHOST. Note that this persistence is not required by any standard. However, our experiments have shown that the most relevant OpenMP implementations GOMP and Intel OpenMP work like this, which makes the assumption of persistent OpenMP threads realistic in practice. After completion of the task's work, the PUs are freed and threads are unpinned in another OpenMP parallel region. At finalization time, the shepherd threads are terminated.

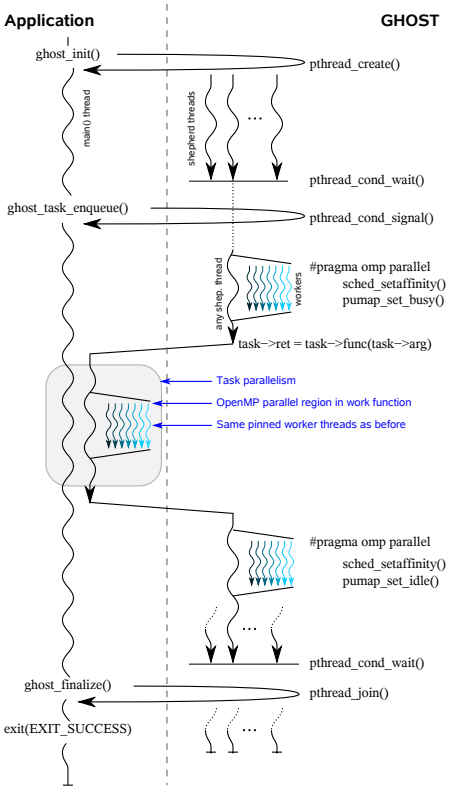


Fig. 4.2: Program flow of an example application using a single GHOST task for asynchronous task parallelism.

It is possible to create nested tasks, i.e., tasks running inside other tasks. A parent task can be configured such that none of its children steals resources of it by specifying the `GHOST_TASK_NOT_ALLOW_CHILD` flag. If this flag is not set, it is expected that par-

ents wait for their child tasks and thus, the children can occupy the parent's resources (as demonstrated in the task-mode SpMV example below). In the simplest case, there is only a single task which includes all the work done in the entire application. This "main task" should be created for all GHOST applications for controlled thread placement and the avoidance of resource conflicts. Moreover, while conducting performance analyses using hardware performance counters, controlled placement of threads is inevitable for making sense of the measurements. On top of this, tasks can be used to implement task-level parallelism by having several tasks running concurrently. Due to the fact that starting a task is a non-blocking operation, asynchronous execution of work is inherently supported by GHOST tasks. Normally, each task uses its own set of resources (= PUs) which is not shared with other tasks. However, as mentioned above, a task can also be requested to not reserve any compute resources. The PUs and their busy/idle state are managed process-wide in a bitmap called `pumap`. The PUs available to GHOST can be set at the initialization phase. This feature can be used, e.g., for integration with third-party resource managers that deliver a CPU set to be used.

A realistic usage scenario for task level parallelism is communication hiding via explicit overlap of communication and computation. This can be done, e.g., in a parallel SpMV routine, which will be called task-mode SpMV. The following code snippet shows the implementation of a task-mode SpMV using GHOST tasks.

```
ghost_task_t *curTask, *localcompTask, *commTask;

/* get the task which I am currently in and which will be split into child tasks */
ghost_task_cur(&curTask);

/* create a heavy-weight task for computation of the local matrix part */
ghost_task_create(&localcompTask, &localcompFunc, &localcompArg,
curTask->nthreads-1, GHOST_NUMANODE_ANY);

/* create a light-weight task for communication */
ghost_task_create(&commTask, &commFunc, &commArg, 1, GHOST_NUMANODE_ANY);

/* task-parallel execution of communication and local computation */
ghost_task_enqueue(commTask); ghost_task_enqueue(localcompTask);
ghost_task_wait(commTask); ghost_task_wait(localcompTask);

/* use the current (parent) task for remote computation */
remotecompFunc(remotecompArgs);

/* destroy the child tasks and proceed with current (parent) task */
ghost_task_destroy(localcompTask); ghost_task_destroy(commTask);
```

In this example, a main task is being split up into two child tasks. Communication and local computation are being overlapped explicitly. In principle, this could also be done via non-blocking MPI calls. However, experience has shown that even nowadays some MPI implementations do not fulfill non-blocking communication requests in an asynchronous way. This has been discussed in various publications where also several attempts to solving this problem have been proposed by, among others, Wittmann et al. [54] and Denis [10]. Thus, in order to create an assured overlap, independent of the MPI library, GHOST's tasking mechanism could be used. Note that in many application scenarios based on GHOST tasks, an MPI implementation supporting the `MPI_THREAD_MULTIPLE` compatibility level is required.

Figure 4.3 depicts the potential performance gain by using GHOST tasks. In this example, 100 parallel SpMV operations on 4 CPU-only compute nodes (as shown in Section 1.4) using the `vanHeukelum/cage15`[4] ($n = 5,154,859$, $n_{nz} = 99,199,551$) stored in SELL-32-1024 have been performed. Note that both overlapped variants require a splitting of the process-local matrix into a local and a remote part, where the remote part contains entries with column indices who require communication of input vector data. Important observations are:

[4] `http://www.cise.ufl.edu/research/sparse/matrices/vanHeukelum/cage15`

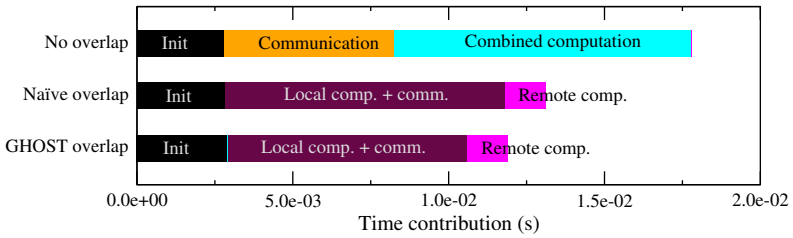


Fig. 4.3: Runtime contributions in different SpMV variants. The "No Overlap" mode communicates input vector data synchronously before it computes the full SpMV. The "Overlap" modes communicate input vector data and at the same time compute the process-local part of the SpMV. Here, the "Naïve" version relies on the asynchronicity of non-blocking MPI calls, whereas the "GHOST" version uses explicit overlap by means of GHOST tasks.

(i) Overlapping communication and computation pays off in this case. The runtime for the two overlapped variants is significantly lower than for the non-overlapped variant. Note that this may not always be the case: The overlapped versions require the result vector to be stored twice and the cost of this may be higher than the benefit from communication hiding.

(ii) The MPI library apparently features asynchronous point-to-point communication routines for this problem. The execution time for overlapped local computation and communication indicates that those operations are really overlapping. Note that this may not be the case in general, even for this MPI library. It is as well possible that the communication volume is below the "eager limit" and larger messages would not be transferred asynchronously.

(iii) Affinity matters. Although one would not expect the task mode variant to perform any better than the MPI-overlapped variant, the execution time for local computation and communication is lower for the version using GHOST tasks. This can be explained by explicit thread placement.

5. Performance features. In this section we present several features of GHOST that constitute a unique feature set leading to high performance for a wide range of applications. The goal of GHOST is neither to provide a "Swiss army knife" for sparse matrix computations nor to re-invent the wheel. Instead, existing implementations are used and integrated into the GHOST interface whenever possible and feasible. In contrast to the *runtime* features presented in Section 4, the described performance features may be available in other libraries as well. In order to justify their implementation in GHOST, short benchmarks or performance models will be shown to demonstrate the potential or measurable benefit over standard solutions.

5.1. Sparse matrix storage. For the SpMV operation, the choice of a proper sparse matrix storage format is a crucial ingredient for high performance. In order to account for the heterogeneous design of GHOST and simplify heterogeneous programming, SELL-C-σ is chosen to be the only storage format implemented in GHOST. This is no severe restriction, since SELL-C-σ can "interpolate" between several popular formats (see below). We briefly review the format here. A detailed and model-guided performance analysis of the SpMV kernel using the SELL-C-σ format can be found in [22].

SELL-C-σ features the hardware-specific tuning parameter C and the matrix-specific tuning parameter σ. The sparse matrix is cut into chunks, each containing C rows where

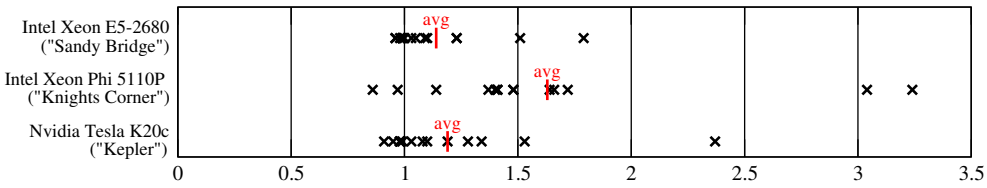


Fig. 5.1: Performance of SpMV using the unified SELL-C-σ storage format relative to vendor-supplied libraries with the device-specific data formats CRS for CPU/PHI and HYB for GPU (figure taken from [22]).

C should be a multiple of the hardware's SIMD width. In a heterogeneous environment, the relevant SIMD width should be the maximum SIMD width over all architectures. For instance, considering our example node's properties as given in Table 1.1 and using 4-byte values (single precision) and indices, the minimum value of C should be

$$[\max(32,64,128)\text{ Bytes}]/[4\text{ Bytes}] = 32.$$

The rows in a chunk are padded with trailing zeros to the length of the chunk's longest row. The chunk entries are stored column-/diagonal-wise. Additionally, in order to avoid excessive storage overhead for matrices with strongly varying row lengths, σ rows are sorted according to their nonzero count before chunk assembly. As this is a local operation, it can be trivially parallelized (which is also done in GHOST). Note that, due to its general formulation, a range of further storage formats can be represented by SELL-C-σ:

- SELL-1-1 $\widehat{=}$ CRS/CSR
- SELL-N-1 $\widehat{=}$ ITPACK/ELLPACK [34]
- SELL-C-1 $\widehat{=}$ Sliced ELLPACK [31]

Thus, a wide selection of known sparse matrix storage formats is supported by GHOST. A single storage format for all architectures greatly facilitates truly heterogeneous programming and enables quick (matrix) data migration without conversion overhead.

Figure 5.1 shows the relative performance of the SELL-C-σ SpMV against the vendor-supplied libraries Intel MKL and NVIDIA cuSPARSE using the device-specific sparse matrix storage format (CRS in MKL and HYB in cuSPARSE). It turns out that the performance of SELL-C-σ is on par with or better than the standard formats for most test matrices.

For easy integration in existing software stacks, GHOST allows to construct a SELL-C-σ matrix from raw CRS data, i.e., row pointers, column indices, and values. A common case in CS&E applications is the subsequent appearance of multiple sparse matrices with the same sparsity pattern but different values. Let us assume that we want to use GHOST and SELL-C-σ for computations with a CRS matrix obtained from another source. Obviously, gathering row lengths and column indices as well as the assembly of communication data structures and permutation vectors only has to be done for the first read-in in this case. Given the ML_Geer matrix (cf. subsection 4.1) present in CRS, we want to perform SpMV using GHOST with SELL-32-128 on two CPU sockets with one MPI rank each. We find that an initial complete construction of this matrix in GHOST (including communication buffer setup and SELL permutation) costs as much as 48 SpMV operations. Note that the communication buffer setup, which has to be done independently of the library or the sparse matrix format, accounts for 78% of the construction time. Each subsequent matrix construction only needs to update the matrix values. Hence, all values need to be read from the CRS data and written to the SELL-C-σ matrix. Taking into account the additional read operation due to the write-allocate of the SELL-C-σ matrix, we have at least $3 \times n_{\text{nz}}$ matrix elements to transfer. The

relative cost depends on the matrix data type. Considering double precision data (and 32-bit indices), subsequent CRS-to-SELL-C-σ conversions should cost as much as two SpMV operations. This performance can also be observed in GHOST. A possible future feature may be sparse matrix views. Using a view, a SELL-1-1 matrix could just point to existing CRS data and GHOST could be used for computation with existing matrices at no conversion cost.

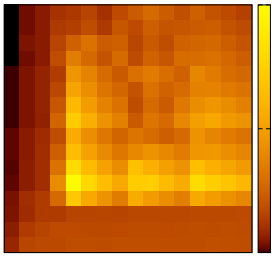
Note that GHOST differentiates between local and global indices. Even for sparse system matrices of moderately large size, it may be necessary to have 64-bit integer numbers for global indices. However, for the process-local part of the system matrix, 32-bit integers may still be sufficient. As data movement should be minimized, especially for (often bandwidth-bound) sparse solvers, it is possible to configure GHOST with 64-bit indices for global quantities (`ghost_gidx_t`) and with 32-bit indices for local quantities (`ghost_lidx_t`). Thus, the column and row indices of the entire process-local sparse matrix can be stored using 32-bit integers. Note that compression of the remote column indices as shown in Fig. 4.1 is inevitable in this case. Considering the minimum amount of data transfers for the SpMV operation, using 32-bit instead of 64-bit column indices for the sparse matrix results in a reduction of data transfers between 16 % (complex double precision data) and 33 % (single precision data).

It is also possible to incorporate matrix-free methods into GHOST. The SpMV routine is stored as a function pointer in the `ghost_sparsemat_t`. A user can replace this function pointer by a custom function that performs the SpMV in any (possibly matrix-free) way, while GHOST handles other kernels and communication of vector data.

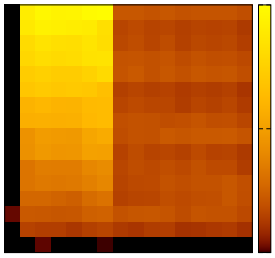
5.2. Block vectors. The architectural performance bottleneck for sparse linear algebra computations is the main memory bandwidth for a wide range of algorithms. Hence, reducing the movement of data through the memory interface often improves performance. One well-known way to achieve this is to process multiple vectors at once in a SpMV operation if allowed by the algorithm. Classic block algorithms are, e.g., the block Conjugate Gradient (CG) method proposed by O'Leary et al. [33] and the block GMRES method introduced by Vital [51]. The continued relevance of this optimization technique is seen in recent publications, e.g., by Röhrig-Zöllner et al. [41] in which the authors present a block Jacobi-Davidson method. Vector blocking is also very relevant in the field of eigenvalue solvers for many inner eigenpairs. For example, the FEAST algorithm [39] and Chebyshev filter diagonalization [44] profit from using block vector operations. Basic work on potential performance benefits from using block vectors has been conducted by Gropp et al. [16], where a performance model for the Sparse Matrix Multiple Vector Multiplication algorithm (SpMMV) has been established. Support for block vectors (which are also represented by objects of `ghost_densemat_t`) has been implemented for many mathematical operations (as presented in Section 5.5) in GHOST.

Generally speaking, block vectors resemble tall and skinny dense matrices, i.e., matrices with many rows and few columns. Although they are represented by general dense matrices, it has turned out that existing BLAS implementations tend to deliver poor performance in numerical kernels using tall and skinny dense matrices. This is the reason why selected tall and skinny matrix kernels have been implemented directly in GHOST. Vectorized and fully unrolled versions of those kernels are automatically generated at compile time for some predefined small dimensions. See Section 5.4 for details on code generation and its impact on performance.

Let V $(n \times m)$ and W $(n \times k)$ be tall and skinny dense matrices where $m,k \ll n$. They are distributed in a row-wise manner among the processes, similar to the system matrix in Fig. 4.1. X should be an $m \times k$ matrix which is redundantly stored on each process. Three



(a) $W \leftarrow VX$



(b) $X \leftarrow V^T W$

Fig. 5.3: Speedup of custom tall and skinny matrix kernels over Intel MKL on a single CPU socket. V is $n \times m$, W is $n \times k$ and X is $m \times k$, where $m, k \ll n$.

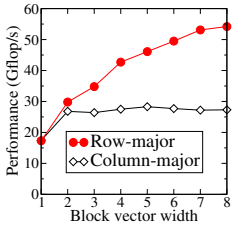


Fig. 5.4: SpMMV performance of row- and column-major block vectors for increasing width.

functions using tall and skinny dense matrices are currently implemented in GHOST:

- $X \leftarrow \alpha V^T W + \beta X$
  Tall skinny matrix transposed times tall skinny matrix (corresponds to inner product of block vectors): `ghost_tsmttsm()`
- $W \leftarrow \alpha VX + \beta W$
  Tall skinny matrix times small matrix: `ghost_tsmm()`
- $V \leftarrow \alpha VX + \beta V$
  In-place version of `ghost_tsmm()`: `ghost_tsmm_inplace()`

One may assume that a mature library like the Intel MKL yields optimal performance for a widely-used kernel like general dense matrix matrix multiplication (GEMM) for matrices of any shape. However, this is not the case as we demonstrate in Fig. 5.3. Similar observations concerning performance drawback of MKL in the context of tall and skinny dense matrices have been made by Anderson et al. [1]. The GEMM kernel with (not too small) square matrices can reach a modern processor's peak floating point performance if properly optimized. The architectural performance bottleneck is the CPU's compute capability in this case. However, this does not hold for tall and skinny matrices. The possibilities of blocking are limited due to the small matrix dimensions in this case. This results in a GEMM kernel which should ideally be memory-bound (as long as the dimension n is sufficiently large). In Fig. 5.3 it can be observed that the GHOST versions of both kernels are at least on par with the MKL performance for relatively small dimensions. The potential speedup can be up to $30\times$ for some matrix sizes. Note that if the general function `ghost_gemm()` is called, it first checks whether a suitable specialized function is applicable before calling the BLAS library.

It is a known concern in extreme-scale computing that reduction operations are susceptible to truncation errors. In GHOST, the computation of the inner product of two tall skinny matrices (`ghost_tsmttsm()`) is one of the kernels where problems of this kind may occur for very large n. This motivated the addition of a variant of this kernel which uses Kahan summation [21]. Depending on the width of the block vectors m and k, and hence the computational intensity of the inner product, the overhead from the additional floating point operations is small or negligible compared to standard summation [18]. However, the improvement in accuracy may be significant which could result in a lower iteration count for some iterative algorithms and such a smaller time to solution. This has been demonstrated, e.g., by Mizukami [29] for CG-like methods.

In order to achieve a transparent support of block vectors in GHOST we implemented

several BLAS level 1 routines and equipped them with block vector support. Currently, the list of block vector operations includes `axpy`, `axpby`, `scal`, and `dot`. Each of those operations works vector-wise. In addition, versions of `axpy`, `axpby`, and `scal` with varying scalar factors for each vector in the block have been added: `vaxpy`, `vaxpby`, `vscal`. Obviously, as block vectors are also represented as dense matrices, all of those operations could be realized using existing BLAS level 3 routines. For example, the `vscal` kernel could be replaced by a multiplication with a diagonal matrix containing the scaling factors on its diagonal. However, this would come at additional cost by transferring zeros, which we want to avoid. Additionally, the concerns about the efficiency of BLAS level 3 operations for tall and skinny dense matrices apply here as well.

Figure 5.4 shows benchmark results for the SpMMV kernel, comparing row-major and column-major storage of block vectors with increasing width. Storing the block vector in row-major corresponds to interleaved storage. As expected, the performance for row-major storage surpasses the performance of column-major storage due to a better data access pattern. This is well known, and both vendor-supplied sparse linear algebra libraries (Intel MKL and NVIDIA cuSparse) support row-major block vectors in their SpMMV kernel.

5.3. Kernel fusion. Many sparse iterative solvers consist of a central SpMV routine accompanied by several BLAS level 1/2 functions. It is thus useful to augment the SpMV with more operations according to our needs. The general sparse matrix vector multiplication function $y = \alpha(A-\gamma I)x+\beta y$ encompasses many of the practical use cases. In GHOST this operation can be chained with the computation of the dot products of $\langle y,y\rangle$, $\langle x,y\rangle$, and $\langle x,x\rangle$ as well as the BLAS level 1 operation $z = \delta z+\eta y$. This approach is similar to the well-known optimization technique of *kernel fusion*. Similar thoughts led to the addition of the so-called BLAS 1.5/2.5 operators `AXPY_DOT`, `GE_SUM_MV`, `GEMVT`, `TRMVT`, and `GEMVER` to the updated BLAS standard [6]. Siek et al. [46] observed the application specificity of these BLAS x.5 operators and made an attempt towards a domain-specific compiler to generate arbitrarily fused kernels consisting of different BLAS calls. This work has been continued by Nelson et al. [32], who plan to adapt their framework towards sparse matrices in future work. Recently, the idea of kernel fusion has gained new attention in the GPU programming community ([43, 49, 52]).

The in- and output vectors of the augmented SpMMV kernel are of `ghost_densemat_t`. Hence, they may also be (views of) block vectors. The values of α, β, γ, and η and the storage location of computed dot products are passed to the function via variadic arguments, which results in a single interface function for any kind of (augmented) sparse matrix (multiple) vector multiplication. Note that each augmentation on top of the standard SpM(M)V can be enabled separately. In the following we show a small example of how to use this function.

```
/* compute y=Ax where y and x may be block vectors */
ghost_spmv(y,A,x,GHOST_SPMV_DEFAULT);

/* declare a scalar shift for each vector in the block */
double shift[nvecs];
/* initialize shifts... */

/* compute y=(A-γI)x with different γ for each vector in the block */
ghost_spmv(y,A,x,GHOST_SPMV_VSHIFT,shift);

/* initialize space for storing the dot products and a scaling factor */
double dot[3*nvecs]; double neg = -2.0;

/* compute y=(A-γI)x-2y with different γ for each vector, chained with ⟨x,y⟩ */
ghost_spmv(y,A,x,GHOST_SPMV_VSHIFT|GHOST_SPMV_AXPBY|GHOST_SPMV_DOT_XY,&neg,shift,dot);
```

Both, the use of block vectors and kernel fusion, have large potential regarding performance, depending on the algorithm. For example, for the Kernel Polynomial Method, a method for

computing the eigenvalue density of quantum systems as analyzed in [23], a 2.5-fold performance gain for the overall solver could be achieved by using block vectors and augmenting the SpMV. Fused kernels are likely to be more cumbersome from an implementation point of view than fine-grained kernels. For instance, fusing the SpMMV operation with block vector dot products on a GPU leads to complex data access patterns which make an efficient implementation hard to achieve (see [23] for details). Due to the potentially high complexity of fused kernels and fundamental architectural differences, hand-optimized implementations for each target architecture can hardly be avoided if the focus is on high efficiency in heterogeneous settings.

A significant design decision for scientific computing libraries is whether and how to use task and data parallelism. A task-parallel approach for work distribution between heterogeneous devices, as implemented in MAGMA [26], may conflict with the presented optimization technique of kernel fusion so that some optimization potential is left unused. In cases where the potential benefits of task parallelism are limited, such as the sparse matrix algorithms targeted by GHOST, data parallelism with kernel fusion may thus be favored over task parallelism.

5.4. Low-level implementation and code generation. GHOST is implemented with the goal of efficient execution from a single core to the petaflop level. Modern CPUs feature SIMD units which cause code vectorization to be a crucial ingredient for efficient core-level code. For kernels with sufficient simplicity, automatic vectorization is likely to be done by the compiler. If this is not the case, GHOST addresses this issue by using compiler pragmas, or SSE, AVX or MIC compiler intrinsics for explicit vectorization. Benchmarks on one CPU showing the impact of vectorization on SpMV performance can be seen in Fig. 5.5. Here, we used the `Sinclair/3Dspectralwave`[5] matrix ($n = 680,943$, $n_{\mathrm{nz}} = 30,290,827$) in complex double precision. A first observation is that all three variants reach the same maximum performance when using the full socket. Due to the the bandwidth-bound nature of the SpMV kernel this limit corresponds to the CPU's maximum memory bandwidth. However, the faster saturation of the explicitly vectorized SELL kernel allows to use less cores to reach the same performance. The spare cores can be used for working on independent tasks (cf. Section 4.2) or they can be switched off in order to save energy. Hence, good vectorization should always be a goal, even for bandwidth-bound kernels. Note that this is especially true on accelerator hardware, where the width of vector units is typically larger than on standard hardware (cf. Table 1.1).

An obstacle towards efficient code often observed by application developers is lacking performance of existing program libraries due to their inherent and indispensable requirement of generality. Often, better performance could be achieved if performance-critical components were tailored to the application. Obviously, this goal is opposing the original goal of program libraries, namely general applicability. An important feature of GHOST for achieving high performance is code generation. At compile time, the user can specify common dimensions of data structures for which versions of highly-optimized kernels will be compiled. A prominent example is the width of block vectors, i.e., the number of vectors in the block. This number typically is rather small, potentially leading to overhead due to short loops in numerical kernels.

The positive impact of hard-coded loop lengths on the performance of SpMMV with increasing block vector width is demonstrated in Fig. 5.6. The hardware and problem setting is the same as described above for Fig. 5.5, i.e., the performance for one vector is the same

[5] `http://www.cise.ufl.edu/research/sparse/matrices/Sinclair/3Dspectralwave.html`

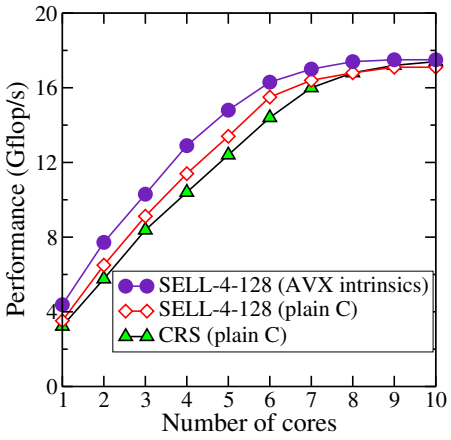


Fig. 5.5: Intra-socket performance on a single CPU showing the impact of vectorization on SpMV performance for different storage formats.

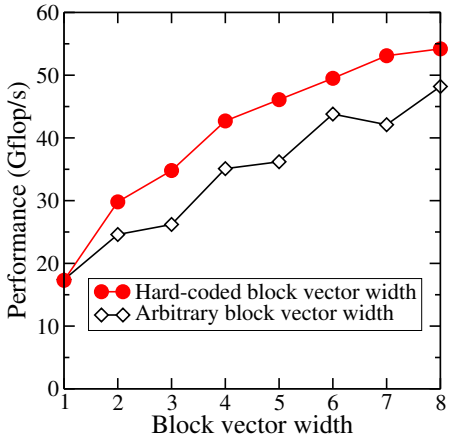


Fig. 5.6: The impact of hard-coded loop length on the SpMMV performance with increasing block vector width on a single CPU.

as the saturated performance of Fig. 5.5. If we configure the block vector widths $1,\ldots,8$ at compile time a significant performance benefit can be achieved compared to a variant where none of them is configured. This is due to more optimization possibilities for the compiler due to simpler code and a lower impact of loop overheads. Note that for both variants the SELL chunk height C was configured at compile time.

Code generation in GHOST serves two purposes. First, marked code lines can be duplicated with defined variations. Second, it is possible to generate variations of functions, similar to C++ templates. A very simple example code using GHOST code generation markers could read as follows:

```
#GHOST_FUNC_BEGIN#FOO=1,2#BAR=5,6
int func_FOO_BAR(void) {
    #GHOST_UNROLL#printf("@\n");#FOO
    return FOO+BAR;
}
#GHOST_FUNC_END
```

After preprocessing, this would be expanded to the following four functions:

```
int func_1_5(void) {
    printf("1\n");
    return 1+5;
}
int func_2_5(void) {
    printf("1\n");
    printf("2\n");
    return 2+5;
}
```

```
int func_1_6(void) {
    printf("1\n");
    return 1+6;
}
int func_2_6(void) {
    printf("1\n");
    printf("2\n");
    return 2+6;
}
```

Note that in this example the factor for code duplication `FOO` depends on the function variant. This disqualifies the straightforward use of C++ templates for function variation, as values of the template parameters would have to be known before compilable (i.e., with evaluated `GHOST_UNROLL` statements) code is present. Note that it is not always possible to replace the generation of code line variants by loops, e.g., for the declaration of variables.

Fallback implementations exist for all compute kernels. This guarantees general appli-

Function	CPU	GPU	PHI
axpy	H	H	H
axpby	H	H	H
scal	H	H	H
dot	H	H	H
vaxpy	H	H	H
vaxpby	H	H	H
vscal	H	H	H
aug_spmmv	$G_{I,H}$(H)	G_H(H)	$G_{I,H}$(H)
gemm	L	L	L
tsmttsm	$G_{I,H}$(L)	L	G^I
tsmttsm_kahan	$G_{I,H}(\times)$	$\times$	$G_H(\times)$
tsmm	$G_{I,H}$(L)	L	G_H
tsmm_inplace	$G_H(L^\dagger)$	$L^\dagger$	$G_H(L^\dagger)$

Table 5.1: Currently available numerical kernels in GHOST. Note that all BLAS level 1/2 functions are block vector-enabled.
"H": High-level implementation.
"L": A library is called.
"G": The kernel is auto-generated at compile time (cf. Section 5.4), fallback version in parentheses.
†: In-/output vector gets cloned.
$\times$: Non-Kahan version gets called along with a warning.
$\cdot_H$: Auto-generated high-level code.
$\cdot_I$: Auto-generated intrinsics code.

cability of GHOST functions. The degree of specialization gets diminished until a suitable implementation is found, which probably implies a performance loss. For example, if a kernel is not implemented using vectorization intrinsics and hard-coded small loop dimensions, a version with one arbitrary loop dimension is searched. If none of the small loop dimensions is available in an explicitly vectorized kernel, GHOST checks for the existence of a high-level language implementation with hard-coded small loop dimensions, and so on. In case no specialized version has been built into GHOST the library will select the highly-general fallback version.

5.5. Overview of available building blocks. Table 5.1 gives an overview of currently existing numerical kernels in GHOST and details about their implementation. This list is likely to be extended in future development, especially once GHOST gains broader attention from other communities.

6. Using GHOST in existing iterative solver packages. In this section we want to briefly discuss how GHOST can be used with existing sparse solver libraries. A characteristic feature of typical iterative solvers for sparse linear systems or eigenvalue problems is that they require only the application of the matrix to a given vector. It is therefore good practice to separate the implementation of such methods from the data structure and details of the SpMV.

One approach that originated in the days of Fortran 77 is the 'Reverse Communication Interface' (RCI). The control flow passes back and forth between the solver routine and the calling program unit, which receives instructions about which operations are to be performed on which data in memory. While this programming model is still widely used in, e.g., ARPACK [25] and even in modern libraries such as Intel MKL [19], it is awkward and error-prone by today's standards. Another idea is to use callback functions for selected operations. For example, the eigensolver package PRIMME [47] requires the user to provide the SpMMVM and a reduction operation on given data.

Neither RCI nor simple callbacks can make optimal use of GHOST. Obviously such software could only make use of accelerators by means of offloading inside a function scope. If no special attention is paid to data placement, this is typically inefficient due to the slow PCI express bus between CPU and device. Even on the CPU, GHOST preferably would control memory allocation itself to achieve alignment and NUMA-aware placement of data. Another drawback is the restriction to data structures as prescribed by such solvers. For instance, the required storage order of block vectors is typically column-major, which may be also inefficient (cf. subsection 5.2).

The Trilinos package Anasazi [4] takes a different approach. It requires the user to implement what we call a 'kernel interface', an extended set of callback functions that are the only way the solver can work with matrices and vectors. New (block) vectors are created by cloning an existing one via the kernel interface. Thus, memory allocation stays on the user side and can be done, e.g., on a GPU, with a custom data layout, or applying further optimizations.

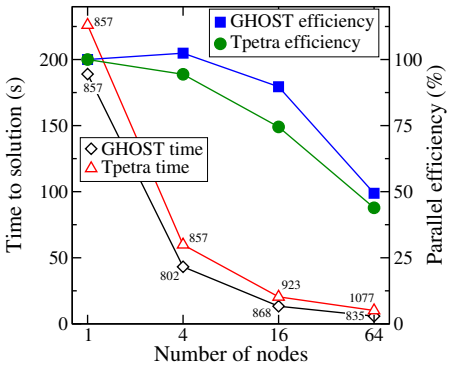
In the iterative solver package PHIST [36] we use a similar kernel interface which is written in plain C. It does not require a very general vector view concept and has some functionality for executing (parts of) kernels asynchronously as GHOST tasks. PHIST also provides GHOST adapters for Anasazi and the linear solver library Belos from Trilinos, with the restriction that views of permuted columns of a block vector do not work with row-major storage. This is not a grave restriction as the feature is – to our knowledge – hardly used in the packages. For a performance study of the block Jacobi-Davidson method implemented in PHIST (using GHOST), see [41].

6.1. Case study: An eigensolver with Trilinos and GHOST. We have demonstrated the applicability and performance of GHOST in a number of publications. In [41], we have presented and implemented a block Jacobi-Davidson method using PHIST & GHOST on up to 512 dual-socket CPU nodes. A fully heterogeneous GHOST implementation of the kernel polynomial method which we scaled up to 1024 CPU+GPU nodes has been demonstrated in [23]. In the meantime, we have continued our scaling studies of this application to 4096 heterogeneous nodes. Recent work includes the implementation of Chebyshev filter diagonalization, for which we show performance data on up to 512 dual-socket CPU nodes in [38].

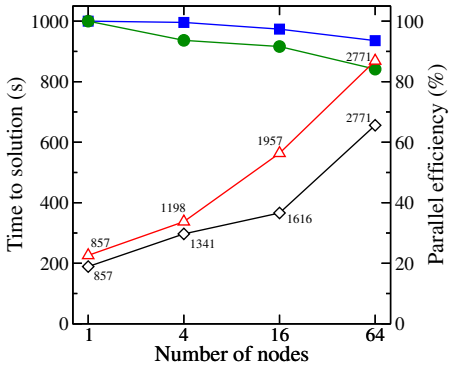
While all of the presented work has been conducted within activities closely related to the GHOST project, we see that it is of special interest for a broader potential user base how GHOST could integrate in existing CS&E software stacks. In the following we want to demonstrate the applicability and performance of GHOST using the Krylov-Schur method [48] for finding a few eigenvalues of large sparse matrices. An implementation of this method is available in the Anasazi package [4] of Trilinos. As mentioned in the previous section, PHIST can serve as an interface layer between algorithmic packages like Anasazi and kernel libraries like GHOST or Tpetra (+Kokkos). Developers can thus work at a high level of abstraction and have the option to switch between kernel implementations. For this study, we are using version 11.12.1 of Trilinos and an MPI+X approach with OpenMP parallelization on the socket level. The test case is the non-symmetric MATPDE[6] problem. It represents a five-point central finite difference discretization of a two-dimensional variable-coefficient linear elliptic equation on an $n \times n$ grid with Dirichlet boundary conditions. The ten eigenvalues with largest real part are sought using a search space of twenty vectors. The convergence criterion is a residual tolerance of 10^{-6}. We set the random number seed in GHOST in a way which guarantees consistent iteration counts between successive runs.

GHOST integrates well with Anasazi and is straightforward to use on this level. Moreover we show in Figure 6.1 that GHOST surpasses Tpetra both in terms of performance and scalability. On a single node one can save about 16% of runtime for the entire solver. Figure 6.1a reveals a higher parallel efficiency of GHOST. Consequently, the better node-level performance gets amplified on larger node counts, resulting in a 42% runtime saving on 64 nodes. For weak scaling, similar conclusions can be drawn from Figure 6.1b. At the largest node count, the parallel efficiency of GHOST is ten percentage points above Tpetra. Relevant GHOST features used in the presented runs are resource management (thread pinning), SpMV with SELL-C-σ and auto-generated kernels for tall & skinny dense matrix multiplica-

[6] http://math.nist.gov/MatrixMarket/data/NEP/matpde/matpde.html



(a) Strong scaling runtime (left axis) and parallel efficiency (right axis) for $n = 2^{12}$.



(b) Weak scaling runtime (left axis) and parallel efficiency (right axis) for $n = 2^{12,\ldots,15}$.

Fig. 6.1: Scaling behavior of GHOST and Tpetra on up to 64 dual-socket CPU nodes for Anasazi's implementation of the Krylov-Schur method. The annotations show the number of iterations until convergence. The computed parallel efficiencies consider changed iteration counts.

tions. Note that even higher performance could possibly be obtained by exploiting advanced algorithmic optimizations available with GHOST, such as kernel fusion, block operations and communication hiding. However, those would potentially require a re-formulation of the algorithm which is not what we wanted to demonstrate here.

7. Conclusion and outlook.

7.1. Conclusions. GHOST is a novel and promising attempt towards highly scalable heterogeneous sparse linear algebra software. It should not be considered a comprehensive library but rather a toolbox featuring approaches to the solution of several problems which we have identified as relevant on modern hardware in the context of sparse solvers. A crucial component of highly efficient software, especially in the complex environment of heterogeneous systems, is sensible resource management. Our flexible, transparent, process-based and data-parallel approach for heterogeneous execution is accompanied by a lightweight and affinity-aware tasking mechanism, which reflects the requirements posed by modern algorithms and hardware architectures. During the ongoing development, we have observed that high performance is the result of a mixture of ingredients. First, algorithmic choices and optimizations have to be made considering the relevant hardware bottlenecks. In the context of sparse solvers, where minimizing data movement is often the key to higher efficiency, this includes, e.g., vector blocking and kernel fusion. Second, while implementing those algorithms, it is crucial to have an idea of upper performance bounds. This can be accomplished by means of performance models, which form a substantial element of our development process. This is demonstrated in [22] and [23]. An optimal implementation may come at the cost of fundamental changes to data structures, e.g., storing dense matrices as row- instead of column-major or changing the sparse matrix storage format from CRS to SELL-C-σ. During the ongoing development it has turned out that often the generality of the interface has to be traded in for high performance. There are several ways to relax this well-known dilemma. Very promising is, e.g., a close collaboration between library and application developers with the possibility for the latter to feed their application-specific knowledge into the library. In

GHOST this idea is implemented by automatic code generation.

7.2. Outlook. In its current state, GHOST has no provision for exploiting matrix symmetry. Obviously, there is large potential for increased of performance if symmetric (or Hermitian) matrices were treated as such. The implementation is challenging, but cannot be avoided in the long run. Bringing sparse solvers to a very large scale is often limited by malicious sparse matrix patterns which lead to communication dominating the runtime. This can be ameliorated by bandwidth reduction of the sparse matrix. A goal for further development is the evaluation and implementation of additional bandwidth reduction algorithms like, e.g., hypergraph partitioning [11]. Furthermore, the optimization of heterogeneous MPI communication, e.g., using GPUdirect which bypasses the host memory in GPU-GPU communication, should be investigated in order to improve the communication performance. Future architectural developments, like deeper memory hierarchies and a tighter integration of "standard" and "accelerated" resources require rethinking existing performance models and possibly new implementations. Currently, the heterogeneous work distribution weights have to be specified by the user, mostly based on knowledge about the involved hardware architectures and their capabilities. In future work, micro-benchmarks will be integrated into GHOST that allow automatic determination of device-specific work weights. On top of that, another important goal for future development is dynamic and automatic load balancing during an iterative solver's runtime. Currently, the sparse matrix portion for each process is fixed during the entire runtime. By using the SELL-C-σ storage format, it will be straightforward to communicate matrix data at runtime between heterogeneous devices to overcome load imbalances.

Acknowledgments. This work was supported by the German Research Foundation (DFG) through the Priority Programs 1648 "Software for Exascale Computing" (SPPEXA) under project ESSEX ("Equipping Sparse Solvers for Exascale"). We would like to thank Intel Germany and Nvidia for providing test systems for benchmarking. Special thanks go to Andreas Alvermann for providing sparse matrix generation functions for testing and everyone else who contributed to GHOST, directly or indirectly.

REFERENCES

[1] M. ANDERSON, G. BALLARD, J. DEMMEL, AND K. KEUTZER, *Communication-avoiding QR decomposition for GPUs*, in Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International, May 2011, pp. 48–58.

[2] C. AUGONNET, S. THIBAULT, R. NAMYST, AND P.-A. WACRENIER, *StarPU: a unified platform for task scheduling on heterogeneous multicore architectures*, Concurrency and Computation: Practice and Experience, 23 (2011), pp. 187–198.

[3] C. G. BAKER AND M. A. HEROUX, *Tpetra, and the use of generic programming in scientific computing*, Sci. Program., 20 (2012), pp. 115–128.

[4] C. G. BAKER, U. L. HETMANIUK, R. B. LEHOUCQ, AND H. K. THORNQUIST, *Anasazi software for the numerical solution of large-scale eigenvalue problems*, ACM Trans. Math. Softw., 36 (2009), pp. 13:1–13:23.

[5] S. BALAY, S. ABHYANKAR, M. F. ADAMS, J. BROWN, P. BRUNE, K. BUSCHELMAN, L. DALCIN, V. EIJKHOUT, W. D. GROPP, D. KAUSHIK, M. G. KNEPLEY, L. C. MCINNES, K. RUPP, B. F. SMITH, S. ZAMPINI, AND H. ZHANG, *PETSc Web page*. `http://www.mcs.anl.gov/petsc`, 2015.

[6] L. S. BLACKFORD, J. DEMMEL, J. DONGARRA, I. DUFF, S. HAMMARLING, G. HENRY, M. HEROUX, L. KAUFMAN, A. LUMSDAINE, A. PETITET, R. POZO, K. REMINGTON, AND R. C. WHALEY, *An updated set of basic linear algebra subprograms (BLAS)*, ACM Transactions on Mathematical Software, 28 (2001), pp. 135–151.

[7] F. BROQUEDIS, J. CLET-ORTEGA, S. MOREAUD, N. FURMENTO, B. GOGLIN, G. MERCIER, S. THIBAULT, AND R. NAMYST, *Hwloc: A generic framework for managing hardware affinities in HPC applications*, in Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and

Network-based Processing, PDP '10, Washington, DC, USA, 2010, IEEE Computer Society, pp. 180–186.

[8] C. CHEVALIER AND F. PELLEGRINI, *PT-Scotch: A tool for efficient parallel graph ordering*, Parallel Comput., 34 (2008), pp. 318–331.

[9] E. CHOW AND A. PATEL, *Fine-grained parallel incomplete factorization*, SIAM J. Sci. Comp., 37 (2015), pp. 169–193.

[10] A. DENIS, *POSTER: a generic framework for asynchronous progression and multithreaded communications*, in Cluster Computing (CLUSTER), 2014 IEEE International Conference on, Sept 2014, pp. 276–277.

[11] K. DEVINE, E. BOMAN, R. HEAPHY, R. BISSELING, AND U. CATALYUREK, *Parallel hypergraph partitioning for scientific computing*, in Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, April 2006, pp. 10 pp.–.

[12] M. GALGON, L. KRÄMER, J. THIES, A. BASERMANN, AND B. LANG, *On the parallel iterative solution of linear systems arising in the FEAST algorithm for computing inner eigenvalues*, Parallel Computing (accepted), (2015).

[13] A. H. GEBREMEDHIN, D. C. NGUYEN, M. M. A. PATWARY, AND A. POTHEN, *ColPack: Software for graph coloring and related problems in scientific computing.*, ACM Trans. Math. Softw., 40 (2013), p. 1.

[14] *GHOST: General, Hybrid, and Optimized Sparse Toolkit*. `http://tiny.cc/GHOST`. Accessed: June 2015.

[15] P. GHYSELS, T. J. ASHBY, K. MEERBERGEN, AND W. VANROOSE, *Hiding global communication latency in the GMRES algorithm on massively parallel machines*, SIAM J. Sci. Comp., 35 (2013), pp. C48–C71.

[16] W. D. GROPP, D. K. KAUSHIK, D. E. KEYES, AND B. F. SMITH, *Towards realistic performance bounds for implicit CFD codes*, in Proceedings of Parallel CFD99, Elsevier, 1999, pp. 233–240.

[17] G. HAGER AND G. WELLEIN, *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press, Inc., Boca Raton, FL, USA, 1st ed., 2010.

[18] J. HOFMANN, D. FEY, J. EITZINGER, G. HAGER, AND G. WELLEIN, *Performance analysis of the Kahan-enhanced scalar product on current multicore processors*, CoRR, abs/1505.02586 (2015).

[19] *Intel Math Kernel Library*. `https://software.intel.com/en-us/intel-mkl`. Accessed: June 2015.

[20] S. KACZMARZ, *Angenäherte Auflösung von Systemen linearer Gleichungen*, Bulletin International de l'Académie Polonaise des Sciences et des Lettres, 35 (1937), pp. 355–357.

[21] W. KAHAN, *Pracniques: Further remarks on reducing truncation errors*, Commun. ACM, 8 (1965), p. 40.

[22] M. KREUTZER, G. HAGER, G. WELLEIN, H. FEHSKE, AND A. R. BISHOP, *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units*, SIAM J. Sci. Comput., 36 (2014), pp. C401–C423.

[23] M. KREUTZER, A. PIEPER, G. HAGER, A. ALVERMANN, G. WELLEIN, AND H. FEHSKE, *Performance engineering of the kernel polynomial method on large-scale CPU-GPU systems*, in 29th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2015), Hyderabad, India, May 2015.

[24] *LAMA: Library for accelerated mathematical applications*. `http://www.libama.org`. Accessed: June 2015.

[25] R. B. LEHOUCQ, C.-C. YANG, AND D. C. SORENSEN, *ARPACK users' guide : solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*, SIAM, Philadelphia, 1998.

[26] *MAGMA: Matrix algebra on GPU and multicore architectures*. `http://icl.cs.utk.edu/magma/`. Accessed: June 2015.

[27] *Matrix Market Exchange Format*. `http://math.nist.gov/MatrixMarket/formats.html#MMformat`. Accessed: June 2015.

[28] J. D. MCCALPIN, *Memory bandwidth and machine balance in current high performance computers*, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, (1995), pp. 19–25.

[29] E. MIZUKAMI, *The accuracy of floating point summations for CG-like methods*, Technical Report 486, (1997).

[30] M. MOHIYUDDIN, M. HOEMMEN, J. DEMMEL, AND K. YELICK, *Avoiding communication in sparse matrix computations*, in IEEE Intern. Parallel and Distributed Processing Symposium, IPDPS '08, IEEE, 2008. Long version appeared as UC Berkeley EECS Technical Report UCB/EECS-2007-123.

[31] A. MONAKOV, A. LOKHMOTOV, AND A. AVETISYAN, *Automatically tuning sparse matrix-vector multiplication for GPU architectures*, in High Performance Embedded Architectures and Compilers, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, eds., vol. 5952 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 111–125.

[32] T. NELSON, G. BELTER, J. G. SIEK, E. JESSUP, AND B. NORRIS, *Reliable generation of high-performance matrix algebra*, ACM Transactions on Mathematical Software, 41 (2015).

[33] D. P. O'LEARY, *The block conjugate gradient algorithm and related methods*, Linear Algebra and its Applications, 29 (1980), pp. 293 – 322. Special Volume Dedicated to Alson S. Householder.

[34] T. C. OPPE AND D. R. KINCAID, *The performance of ITPACK on vector computers for solving large sparse*

linear systems arising in sample oil reseervoir simulation problems, Communications in Applied Numerical Methods, 3 (1987), pp. 23–29.

[35] *PARALUTION*. `http://www.paralution.com`. Accessed: June 2015.

[36] *PHIST: Pipelined Hybrid-parallel Iterative Solver Toolkit*. `https://bitbucket.org/essex/phist`. Accessed: June 2015.

[37] A. PIEPER, R. L. HEINISCH, G. WELLEIN, AND H. FEHSKE, *Dot-bound and dispersive states in graphene quantum dot superlattices*, Phys. Rev. B, 89 (2014), p. 165121.

[38] A. PIEPER, M. KREUTZER, M. GALGON, A. ALVERMANN, H. FEHSKE, G. HAGER, B. LANG, AND G. WELLEIN, *High-performance implementation of Chebyshev filter diagonalization for interior eigenvalue computations*. Submitted, 2015.

[39] E. POLIZZI, *Density-matrix-based algorithm for solving eigenvalue problems*, Phys. Rev. B, 79 (2009), p. 115112.

[40] R. RABENSEIFNER, G. HAGER, AND G. JOST, *Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes*, 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 0 (2009), pp. 427–436.

[41] M. RÖHRIG-ZÖLLNER, J. THIES, M. KREUTZER, A. ALVERMANN, A. PIEPER, A. BASERMANN, G. HAGER, G. WELLEIN, AND H. FEHSKE, *Increasing the performance of the Jacobi-Davidson method by blocking*. Accepted for publication in the SIAM J. Sci. Comput., 2014.

[42] K. RUPP, F. RUDOLF, AND J. WEINBUB, *ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs*, in Intl. Workshop on GPUs and Scientific Applications, 2010, pp. 51–56.

[43] K. RUPP, J. WEINBUB, A. JÜNGEL, AND T. GRASSER, *Pipelined iterative solvers with kernel fusion for graphics processing units*, CoRR, abs/1410.4054 (2014).

[44] G. SCHOFIELD, J. R. CHELIKOWSKY, AND Y. SAAD, *A spectrum slicing method for the Kohn-Sham problem.*, Computer Physics Communications, 183 (2012), pp. 497–505.

[45] G. SCHUBERT, H. FEHSKE, L. FRITZ, AND M. VOJTA, *Fate of topological-insulator surface states under strong disorder*, Phys. Rev. B, 85 (2012), p. 201105.

[46] J. G. SIEK, I. KARLIN, AND E. R. JESSUP, *Build to order linear algebra kernels*, in Workshop on Performance Optimization for High-Level Languag es and Libraries (POHLL 2008), April 2008.

[47] A. STATHOPOULOS AND J. R. MCCOMBS, *PRIMME: preconditioned iterative multimethod eigensolver–methods and software description*, ACM Trans. Math. Softw., 37 (2010), pp. 1–30.

[48] G. W. STEWART, *A krylov–schur algorithm for large eigenproblems*, SIAM Journal on Matrix Analysis and Applications, 23 (2002), pp. 601–614.

[49] S. TABIK, G. ORTEGA, AND E. GARZN, *Performance evaluation of kernel fusion BLAS routines on the GPU: iterative solvers as case study*, The Journal of Supercomputing, 70 (2014), pp. 577–587.

[50] *TOP500 Supercomputer Sites*. `http://www.top500.org`. Accessed: June 2015.

[51] B. VITAL, *Etude de quelques mthodes de rsolution de problmes linaires de grande taille sur multiprocessor*, PhD thesis, Universit de Rennes, Rennes, 1990.

[52] M. WAHIB AND N. MARUYAMA, *Scalable kernel fusion for memory-bound GPU applications*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, Piscataway, NJ, USA, 2014, IEEE Press, pp. 191–202.

[53] S. WILLIAMS, A. WATERMAN, AND D. PATTERSON, *Roofline: An insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76.

[54] M. WITTMANN, G. HAGER, T. ZEISER, AND G. WELLEIN, *Asynchronous MPI for the masses*, CoRR, abs/1302.4280 (2013).