The LAMA Approach for Writing Portable Applications on Heterogenous Architectures

Thomas Brandes, Eric Schricker, Thomas Soddemann

Abstract Ensuring longevity and maintainability of modern software applications is mandatory for a proper return on investment. Since the hardware landscape is changing rapidly and will continue to do so, it is imperative to take on those topics also in the HPC domain where applications traditionally have a long live-span. For recent years, we have observed a trend towards more and more heterogeneous systems. Realizing the performance promises of the hardware vendors is a huge challenge to the software developer. Portability is the second challenge to be met in this context. In this paper we present our library LAMA. We created this library to address both challenges successfully in the realm of linear algebra and numerical mathematics. We introduce our solutions to heterogeneous memory and kernel management as well as our solutions to task parallelism. In the end we do performance and scalability benchmarks drawing a comparison to PETSc for the example of a CG solver.

1 Introduction

Ensuring longevity and maintainability of modern software applications is mandatory for a proper return on investment. Since the hardware landscape is changing rapidly and will continue to do so, it is imperative to take on those topics also in the HPC domain where applications traditionally have a long live-span. For recent years, we have observed a trend towards more and more heterogeneous systems in computing. Realizing the performance promises of the hardware vendors is a huge challenge to the software developer. Portability is the second challenge to be met in this context.

At present, most heterogeneous systems are composed of CPUs with accelerator boards. Those boards incorporate powerful computing chips like GPUs (predomi-

Thomas Brandes, Eric Schricker, Thomas Soddemann
Fraunhofer Institute for Algorithms and Scientific Computing, Sankt Augustin, Germany,
e-mail: thomas.brandes|thomas.soddemann@scai.fraunhofer.de
e-mail: eric.schricker@scai-extern.fraunhofer.de

nantly), various many-core processors (e.g. Intel Xeon Phi - most used in this field), or even FPGAs and DSPs for more dedicated tasks in bio-informatics or geophysical applications.

In November 2008, the first GPU accelerated super-computer (Tsubame [8]) hit the Top500 list. Today[1] 104 systems make use of GPUs and Intel® Xeon Phi™s acceleration. Four can even be found in the Top 10. By having a look at the road-maps of the prevailing hardware vendors, it is by far more than just conceivable that heterogeneity in the hardware landscape is going to grow in the future.

Programming those systems and achieving reasonable code performance is a challenge. Programming as well as memory models exhibit tremendous differences between architectures. Optimizing code on non-traditional CPU architectures is poorly developed. Addressing the performance portability aspect across hardware architectures is a challenge in itself. In addition, work balancing approaches and task (co-)scheduling on those hardware architectures is in its infancy. However, these novel architectures are offering high processing element densities and memory bandwidths. Application domains with a huge demand for computing power are going to make use of those architectures.

Today, we are expected to face the task of porting legacy codes to benefit heterogeneous computing infrastructures. That process reveals design weaknesses which should be avoided in new developments at all costs. That was the motivator and the incentive for creating the Library for Accelerated Math Applications.

LAMA started out as a linear algebra library for sparse matrices. It supports an abstraction scheme which allows a separation of concerns regarding implementing numerical algorithms vs. implementing clever and highly tuned code segments for utilizing modern and future hardware architectures. The main design goals can be formulated as

- being easily extensible as far as supporting novel hardware architectures are concerned
- being flexible regarding its application domains.

This paper describes the general design of the LAMA library. Next, we want to outline the main concepts of our implementation (chapter 2). Later on, we perform a comparison especially with PETSc [1] (chapter 3). Finally we sum up the presented approaches and draw some conclusions for future work.

2 LAMA

Extensibility and flexibility are main design goals for LAMA. We see extensibility as a main goal in order to answer user demands for creating new algorithms and augmenting LAMA with new features. This also holds for the underlying subsystem with respect to supporting next generations of hardware architectures. Flexibility pays

[1] Status is the Top500 list from November 2015 [9].

respect to user requirements on being able to support a broad variety of hardware architectures in a seamless manner.

Therefore, a modular software stack was created. In its layering scheme levels of abstractions are stacked on top of each other from strong hardware dependence to complete hardware agnostic implementation layer for numerical algorithms. Each of those layers is organized as a separate library. That allows any part of the library to be employed or reused independently from the ones on the higher levels of abstraction.

The dependencies to internal and external libraries are reduced to a minimum, and interfaces are kept as simple as possible. Figure 1 illustrates the design of the software stack with its dependencies.

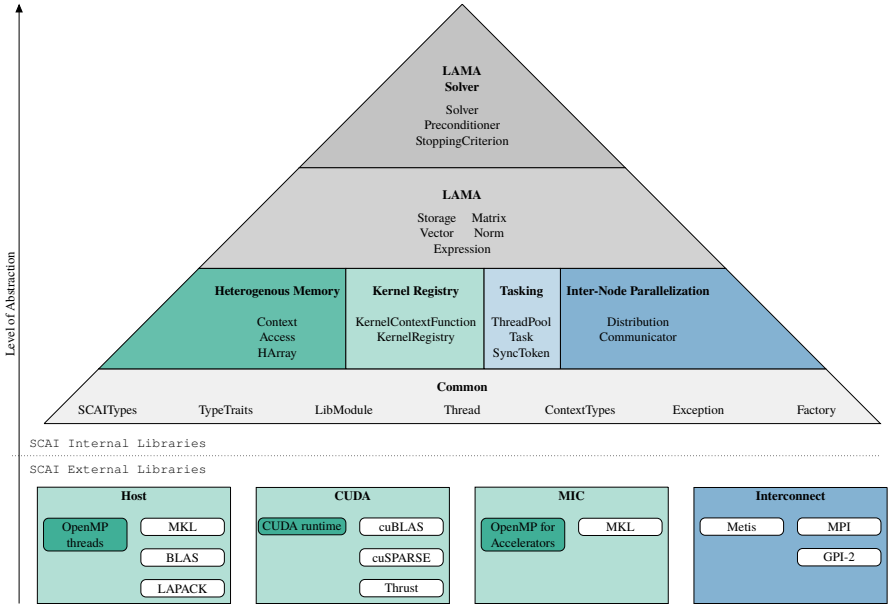


Fig. 1 Hierarchical design of LAMA showing the software stack of LAMA with major concepts in the separate libraries as well as internal and external dependencies. External dependencies are mapped by color.

On the bottom all external libraries are listed that are (optionally) used by LAMA for the support of the different architectures and inter-node communication. Primarily, those are the runtime libraries for supported devices (OpenMP® thread, CUDA® driver and runtime, OpenMP for Accelerator)[2] as well as vendor specific BLAS, sparse BLAS and LAPACK implementations. Currently we provide interfaces to Intel MKL/BLAS, cuBLAS and cuSPARSE. For inter-node parallelization LAMA makes use of the communication standards MPI, for the message passing programming

[2] In LAMA the host CPU is considered as a device that consists of multiple cores and therefore OpenMP is used for parallelization. OpenMP is also exploited for the implementation of kernels on the Intel Xeon Phi architecture using the offload support.

model, and GPI-2, for the partitioned global address space programming model. Efficient implementations are provided on the respective sides.

The next level indicates our main concept for the design of portable parallel applications. A strict separation between memory management, kernel routines, tasking and communication creates a common layer for all supported devices, asynchronous execution and distributing data. As a consequence, all routines implemented on the next higher levels are independent of the underlying device and communication model.

More details about the memory and the kernel management will follow in section 2.1 and 2.2. Section 2.3 explains the principles of tasking and section 2.4 details how data is distributed across node boundaries. The top two levels with the most abstraction from the hardware model exhibit the functionality for dealing with matrices and vectors and finally linear equation solvers. The main functionalities are described in section 2.5 and 2.6.

2.1 Heterogeneous Memory

The central concept of the LAMA design is the possibility to implement operations independently from the underlying hardware architecture. Hence, we introduce an abstraction for memory and kernels. That abstraction has independent back ends (represented by a `Context`), supporting one processor kind or a particular group of hardware. In the following we are going to introduce LAMAs memory abstraction.

The goal for the memory abstraction layer is providing means for accessing data on all supported hardware devices transparently in a unique way. Specialized device kernels on the layer below will be called for executing the data accesses on a particular hardware device. Listing 1 shows an example for the implementation of an add method operating on two arrays. The optional argument `prefCtx` specifies which `Context` this operation should ideally be executed in (s.b.).

The heterogeneous memory management provides a container class `HArray` which allows memory management on any supported back end. During its lifetime it can have incarnations on different back ends. From its first use on at least one incarnation is always valid. Accessing data is only possible via an explicit `Read-` or `WriteAccess` for a given `Context`. The constructor of the corresponding access object takes care that valid data is available, i.e. potentially necessary memory transfers are performed implicitly. Figure 2 demonstrates and explains the impacts of accesses for a snapshot of possible states of the involved arrays from listing 1.

The decision on which `Context` the operation is executed is independent from the operation itself. LAMA supports preferred `Contexts` that can be specified for the dedicated data structure. It can be considered as a hint where the data is needed in future. A good choice might take into account on which `Context` already valid data is available in order to reduce memory traffic.

```
1  template<typename T>
   void add( HArray<T>& res, const HArray<T>& a, const HArray<T>& b,
       ContextPtr prefCtx = ContextPtr() )
3  {
       IndexType n = a.size(); /* assuming b.size() is the same */
5      static LAMAKernel<UtilKernelTrait::add<T> > add;
       ContextPtr ctx = add.getValidContext( prefCtx );
7      ReadAccess<T> readA( a, ctx );
       ReadAccess<T> readB( b, ctx );
9      WriteOnlyAccess<T> write( res, ctx, n );
       add[ctx]( write.get(), readA.get(), readB.get(), n );
11 }
```

Listing 1 Generic implementation for an `add` method operating with two arrays: In line 5 the required kernel method is looked up from the registry and a valid `Context` for the kernel is retrieved (line 6). Afterwards the mandatory `Access` instances on the arrays are tied to the active `Context` (lines 7 to 9), then the device specific kernel method is called (line 10).

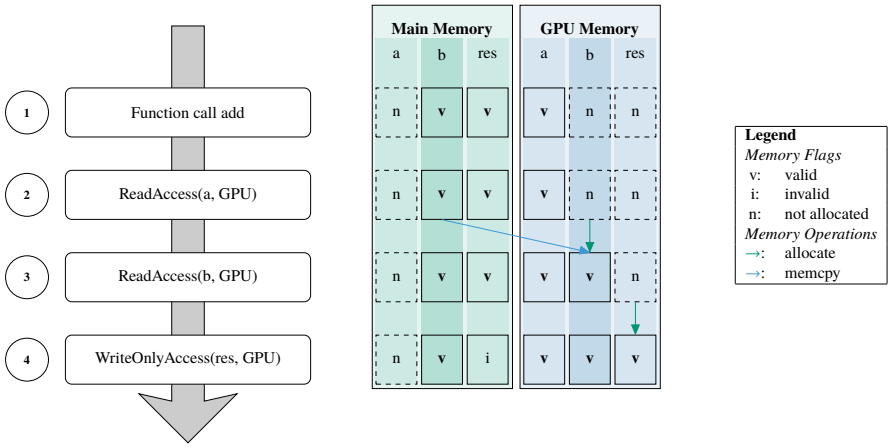


Fig. 2 Incarnations of heterogeneous arrays before and after an operation on GPU: Array a has only a valid incarnation on the GPU, b and res only on the CPU. The operations should be executed on the GPU. While the `ReadAccess` for a is directly possible, the `ReadAccess` of b implies the allocation of data on the GPU and the corresponding memory transfer. The `WriteAccess` for res invalidates the incarnation on the CPU without freeing the memory and allocates memory on the GPU. As this `WriteAccess` is declared as write-only, the previously valid data on the CPU is no more needed and no memory transfer is involved.

2.2 Heterogeneous Kernel

In Listing 1 the interface to our kernel management has already silently been introduced: you can access static references to a kernel functions (see line 5) and execute them on valid `Contexts` (line 10). So a transparent usage of a kernel routine

on various `Contexts` is given. This design has the advantage that by adding a new back end the developer only needs to implement really required functions.

A default `Context` is returned by the registry (line 6) if it is not given on the preferred `Context`. This is helpful in the development of new algorithms as a routine does not have to be implemented for all `Contexts` at once. And it can be a good solution for non time-critical routines e.g. in the setup or for routines that can not be implemented efficiently on a certain `Context`. For obtaining optimal performance all required methods have to be implemented for a the preferred `Context`, nevertheless, of course. This kind of kernel implementation introduces an additional layer (the shown generic function in listing 1) for executing kernels but exhibits the desired flexibility to run on all given devices.

In case of template functions, a registration to a `Context` also implies the instantiation for the needed value type. Both, registration and use of kernel routines employ so-called `KernelTraits` (refer to (I) in listing 2), that are structures describing the signature of the routine and providing unique names used for registration (refer to (III)). They facilitate the use of kernel routines but they also increase the safety by avoiding any mismatches between registration and use.

In the first version of LAMA, the implementation of the `KernelRegistry` exploited abstract classes and pure virtual methods. Unfortunately virtual methods cannot be template methods as overloading and overriding would be mixed up. A possible workaround with a virtual routine for each supported type does not only conflict with extensibility as a design goal but also implies an unacceptable amount of source coding. Hence, in the current implementation, we introduced a new data structure which supports overloading and overriding by a map of function pointers. The penalty of less safety as in the previous approach is compensated by employing `KernelTraits`.

2.3 Task Parallelism

Heterogeneous architectures exhibit a large amount of parallelism. It is essential for optimal usage of those architectures to make use of that inter-architecture parallelism. Besides the previously described parallelism within a given kernel back end, there is additional parallelism inside a node, e.g. between host and accelerator. In the following, we draw a sketch of our task parallel execution model. Inter-node parallelism is treated later in section 2.4.

For intra-node task parallelism we identified two types of tasks that can be executed independantly from the each other.

1. memory transfers between host and device or between devices
2. computation on host or accelerator device

Tasks may be executed asynchronously to optimize the usage of available system resources.

```
// (I) kernel traits for vector add
template<typename ValueType>
struct add
{
  typedef void ( *FuncType )(
    ValueType* z,
    const ValueType* x,
    const ValueType* y,
    const IndexType n );

  static const char* getId()
  {
    return "Util.add";
  }
};

// (II) OpenMP implementation for vector add kernel
template<typename T>
void add( T res[], const T a[], const T b[], IndexType n )
{
    #pragma omp parallel for
    for( IndexType i = 0; i < n; ++i )
    {
        res[i] = b[i] + a[i];
    }
}

// (III) registration for the vector add kernel for the host
   context
KernelRegistry::register( UtilKernelTrait::add<double>, &add,
   context::Host );
KernelRegistry::register( UtilKernelTrait::add<float>, &add,
   context::Host );
```

Listing 2 Code snippets showing the implementation of a vector add kernel for the host: (I) The corresponding `KernelTrait` for the vector add kernel. It is used to identify the kernel inside the `KernelRegistry`. The kernel signature has to match the typedef of `FuncType`. (II) Shows the OpenMP kernel which implements the vector addition. (III) Defines the registration for the `Host` for different value types using the `KernelTrait` and the function pointer to the kernel is figured.

In our tasking library we use threads for the host back end. Threads are taken from a thread pool to avoid the overhead of thread creation. For the CUDA back end streams are used. The Xeon Phi back end just handles asynchronous memory transfer by the corresponding offload directives, so far.

`SyncToken`s are introduced to avoid access conflicts on `HArray`s. They resume the ownership of the corresponding `Read`- and `WriteAccess`es that are used within the task. Tasks which cannot be executed asynchronously return dummy tokens.

In LAMA we apply task parallelism mostly in two major cases: for hiding memory transfer to and between accelerators and inter-node communication while executing calculation tasks. E.g., a pre-fetch on an `HArray` can be invoked while computation is carried out to get data transferred to where it is needed in time. Especially our matrix-vector-multiplication benefits from this execution model. With accelerator support this additional includes asynchronous memory transfer. For more details please refer to [5].

2.4 Distributed Memory Support

When it comes to larger problem sizes, the hardware resources of a single node are often not sufficient any more for handling the mathematical problem efficiently. Hence, multiple nodes will be used in parallel which requires the support for coping with halo data. Optimizing the distribution of data is vital to the performance of the calculations. Therefore in LAMA a `Distribution` can be calculated by graph partitioning algorithms using Metis [4]. This also allows weighting distribution sizes even for non-homogeneous systems.

With assigning a `Distribution` to a data structure, every compute process has a knowledge of the part of the data it is responsible for and a `CommunicationPlan` is calculated. This plan describes the pattern and amount of data used for exchanging data elements at the border to neighboring processes. Than plan can be reused, so by setting up and saving it we save overhead with every exchange.

The pure communication layer introduces an abstraction from the underlying communication library by using a `Communicator` instance. It has special routines, that are optimized for its needs. Currently, LAMA supports two specialized implementations: MPI and GPI-2. MPI is an established standard for distributed memory support. GPI-2 is a recent development in the field of paritioned global address space (PGAS). It is an API for one-sided and asynchronous communication that allows a perfect overlap between computation and communication. Together with the support of asynchronous computations some applications take advantage of it.

2.5 Matrices and Vectors

Generic matrices and vectors are implemented on top of `HArray`. Hence, they are independent from `Context`s. However, a `Matrix` or `Vector` have `Distribution`s. That allows a transparent use in a multi-node setup.

Several specializations of the generic `Matrix` are available in LAMA (see figure 3). Hence, algorithms can be formulated using the `Matrix` interface. A valid line of code for a matrix A and the vectors x and y can look like the following: `y = A * x;` This expression syntax is realized by employing expression templates

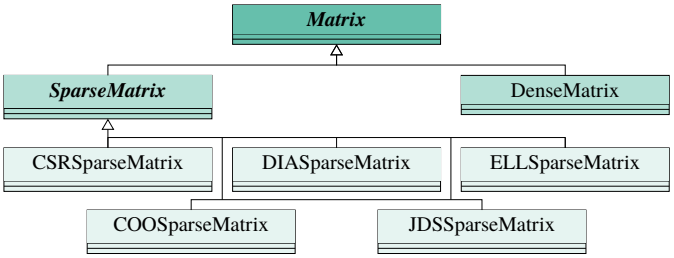


Fig. 3 Matrix inheritance hierachy: currently LAMA contains a dense matrix format as well as COO, CSR, DIA, ELL and JDS sparse matrix formats (for more information about sparse matrix storage formats refer to [7]). Every format has its own advantages and disadvantages. E.g., the CSR-format works well on CPU, while the ELL- or specialized JDS-format are usually optimal for architectures like GPUs.

(for reference on expression templates see [10]). On the basis of the underlying layers it can be executed single- or multi-node on each supported `Context`.

Inside an expression the `Distributions` for matrices and vectors have to be compatible. Therefore, LAMA supports redistributing data by assigning a new distribution. Matrix data is always distributed line by line and it is split up in two parts, a local part of the matrix and its halo part. In the sparse matrix-vector multiplication for the calculation on the halo part data of the right-hand-side vector needs to be transferred before the calculation can be executed.

As mentioned before, LAMA supports overlapping communication and calculation. For this firstly the communication of the halo data is initiated, so the calculation on the local matrix can be executed asynchronously. Afterwards, computation on the transferred data takes place. The benefits of that approach are described for the algebraic multi-grid method in [5].

2.6 Solver Framework

As the highest level of abstraction LAMA provides a solver framework with a large set of linear `IterativeSolvers`. Each solver makes use of the textbook syntax with distributed matrices and vectors. Employing LAMA a user can implement solvers almost precisely as formulated in mathematical syntax as shown in figure 4.

The implementation of a solver does not make any assumption about multi-nodal distribution or execution device. As a result a `Solver` only has to be written once and can be executed in all heterogeneous environments including multiple nodes and different accelerators. Furthermore, the user can explore with LAMA different target hardware architectures as well as different fields of target problems with the right kind of solver and sparse matrix format.

$r_0 = b - Ax_0$;
$d_0 = r_0$;

for $k = 0, 1, \ldots$ **to** $\|r_{k+1}\| < tol$ **do**
  $z = Ad_k$;
  $\alpha_k = \frac{r_k^T r_k}{d_k^T z}$;
  $x_{k+1} = x_k + \alpha_k d_k$;
  $r_{k+1} = r_k - \alpha_k z$;
  $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$;
  $d_{k+1} = r_{k+1} + \beta_k d_k$;
end

```
void cg( Vector& x, const Matrix& A,
    const Vector& b, const Scalar tol )
{
  // inherit type, context of b
  RuntimeVectors tmpVectors(b, 3);
  Vector& r = tmpVectors[0];
  Vector& d = tmpVectors[1];
  Vector& z = tmpVectors[2];
  r = b - A * x;
  d = r;
  Scalar rOld = r.dotProduct(r);
  L2Norm norm;
  for (int k = 0; norm(r) < tol; k++)
  {
    z = A * d;
    Scalar alpha = rOld /
                   d.dotProduct(z);
    x = x + alpha * d;
    r = r - alpha * z;
    Scalar rNew = r.dotProduct(r);
    Scalar beta = rNew / rOld;
    d = r + beta * d;
    rOld = rNew;
  }
}
```

Fig. 4 Comparison of the mathematical description (left) of the CG method with its implementation (right) in LAMA textbook syntax. The matrices and vectors involved can have any `Distribution`, any `Storage` format or any precision and might have valid data on any `Context`. The temporary runtime `Vectors` r, d, and z are allocated with the constructor and freed with the destructor of *RuntimeVectors*, they will have the same type and `Context` as the input `Vector` b. The operations will be executed (preferably) on the `Context` set for `Matrix` A and `Vector` b.

Solvers are specializations of the the abstract class `Solver`. `IterativeSolver` in turn is a specialization for solvers that are composed of single iterations and where a `StoppingCriterion` decides about the number of iterations in the solve method. The iterative solvers are categorized into splitting, Krylov subspace and multigrid methods. A full list of all available iterative solvers is shown in figure 5. Additionally an inverse solver[3] can be used as direct solver on the coarsest grid of the AMG. In principle, each solver serve as a preconditioner for every other solver. If that make sense numerically is left to the user. Algebraic preconditioners like ILU or block methods are under construction.

Which kind of solver should be used for a specific kind of problem is in the hands of the user. Also the chosen sparse matrix format should suite the matrix structure. Using a solver is really easy as shown in listing 3 for a CG solver.

[3] The inverse solver uses an interface to the ScaLAPACK library and can be used only for block-cyclic distributions and the MPI communicator

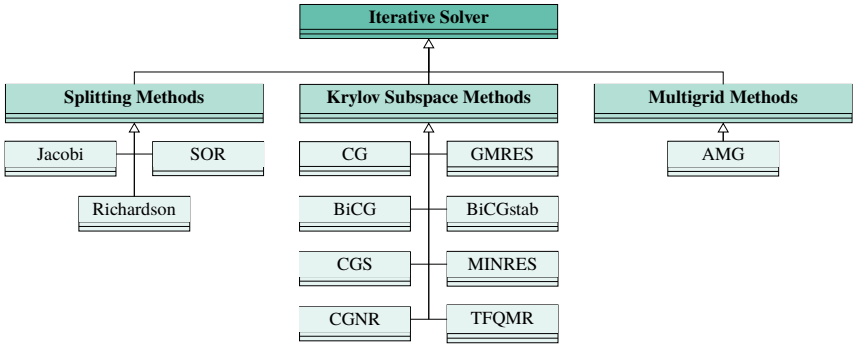


Fig. 5 Iterative solvers inheritance hierarchy: currently LAMA supports the three categories of splitting, Krylov subspace and multigrid method with various solvers.

```
CG cgSolver( "CGSolver"/*, logger*/ );
cgSolver.intialize( matrix );
cgSolver.setStoppingCriterion( new ResidualThreshold( new L2Norm
    (), Scalar( 0.0001 ), ResidualThreshold::Absolute ) );
// optional preconditioning:
// cgSolver.setPreconditioner( otherSolverPtr );
cgSolver.solve( solutionVector, rhsVector );
```

Listing 3 Usage of a Iterative Solver: Set up the solver (line 1), initialize it with the system matrix (line 2), set a stopping criterion (line 3), perform optional preconditioning (line 5) and call solve with a starting solution and a right hand side vector (line 6). When the routine returns, the `solutionVector` holds the solution of the last performed iteration. In order to define when the iteration has to stop, different kind of stopping criteria are provided that might also be combined.

2.7 Extensibility and Maintainability

LAMA makes use of widely applied design patterns, among them templates and factories. Here, we briefly want to motivate our choices.

Templates are employed throughout LAMA at various places, e.g. to introduce some kind of abstraction from arithmetic data types. Data structures can be instantiated for float, double, complex or double complex. The use of quad precision data types like long double and long double complex is restricted to the CPU as these data types are not directly supported by accelerators anyway, yet.

Various factories are also offered in LAMA. All base class instances of general concepts like `Matrix`, `Vector`, `Communicator`, `Distribution` and `Solver` are created using the appropriate `Factory` for obtaining a particular implementation. This allows LAMA to be seamlessly extended with new class specializations without the need of touching the code which uses the general types.

New functionality can be added by loading a separate library module at runtime, very similar to the import feature of the Python. Technically this is realized by the

factories where new derived classes are registered in the application at runtime. This is also very convenient for a dual license model as commercial extensions of LAMA (e.g. highly optimized kernels or very efficient solvers) can be added dynamically to an already installed free installation.

Furthermore, our design enhances the quality of tests within LAMA. Each extension with a new derived class can be tested directly for correct functionality and only class specific tests have to be added.

3 Performance Comparison

In this section we briefly compare LAMA's performance to PETSc, which is a linear algebra library with the similar focus and similar functionalities in comparison with LAMA. It is written in C and exists for quite some time. PETSc enjoys a large and vivid user community. Furthermore, in one case a basic MKL BLAS (native) implementation is used for comparison, as well.

The following benchmark results show the total run times for a CG solver after 1000 iterations[4]. The testing environment (physical system, used compilers and software) is explained in appendix 5.1. The matrices used in the following benchmarks were taken from the University of Florida Collection [6] (see appendix 5.2).

Figure 6 and 7 show the single node performance for CPU and GPU. The weak scaling for both libraries is evaluated in figure 8 by using three Poisson matrices on one to four nodes.

For the case of the CPU-only use we have just measured for the CSR format. Here, both libraries make use of Intel's highly performant MKL BLAS implementation "under the hood". This gives a good base line for our design overhead as well as for PETSc's. Looking at the results, the native BLAS implementation is (of course) fasted. Comparing with PETSc, we see a minimal benefit for our competitor in this case. However, the results demonstrates for both libraries that their designs introduce a negligible performance loss. Furthermore, that also proves that in this case our (memory) model works successfully.

On a single GPU results for the CSR format also are nearly the same with a tiny benefit in favor of LAMA. This is not a big surprise as both packages rely on the CSR sparse matrix-vector multiplication provided by the cuSPARSE library, which dominates the runtime (about 80%). For the axpy and dot operations, LAMA calls cuBLAS routines while PETSc exploits implementations using the Thrust library. Analysis by the NVidia Visual Profiler shows a small advantage for LAMA, here.

When using the ELL format on a **single GPU**, there is currently a small advantage for PETSc in terms of performance. PETSc uses the corresponding kernel of the cuSPARSE library while LAMA uses an own CUDA kernel. However, the LAMA kernel provides the possibility to bind the right-hand-side vector in the matrix-vector multiplication to the texture cache of the GPU that can improve the data locality. On

[4] The time for one CG iteration is mainly given by one sparse matrix-vector multiplication, four dot products and three axpy operations.

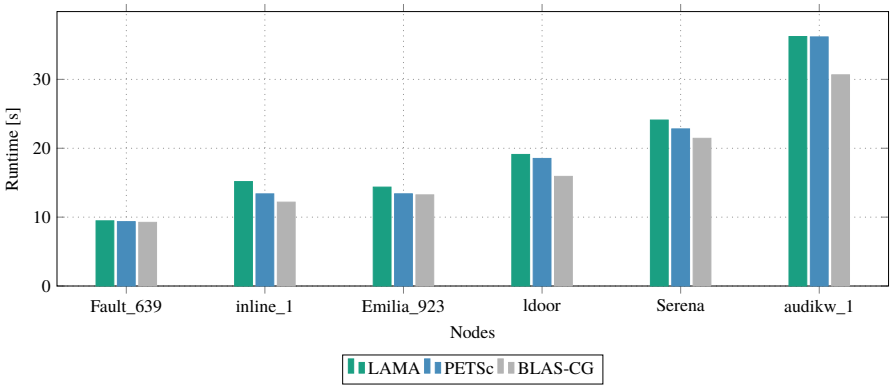


Fig. 6 CPU comparision LAMA - PETSc (CSR): single node CPU performance utilized by six MPI-processes. Runtime is proportional to the number of non-zeros - only the irregulare structure of *inline_1* and *audikw_1* show remarkably higher runtime.

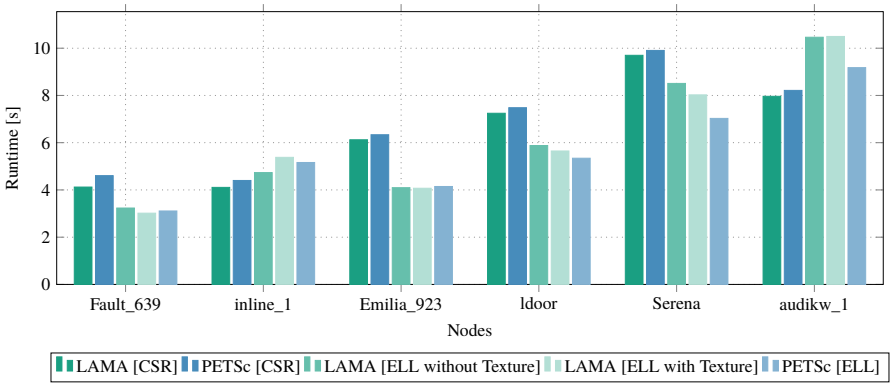


Fig. 7 GPU comparision LAMA - PETSc (CSR, ELL): single GPU performance for CSR- and ELL sparse matrix format. PETSc relies for both formats on the cuSPARSE kernels, LAMA for CSR, too, for ELL on its own kernel (with the possibility to use the texture cache for the right-hand-side vector). Regarding the storage format, the ELL format has shorter runtimes, only for *inline_1* and *audikw_1* the CSR format is the faster one. These latter two matrices have nearly twice the number of entries per row and here the CSR format outperforms the ELL format.

Nvidia's Kepler GPU architecture the use of texture cache increases the performance slightly except for one test case. Nevertheless, the cuSPARSE kernel improved to previous versions and now outperforms the LAMA kernel.

Since the LAMA GPU-Kernels are also tuned for **multi GPU** (via MPI) usage, it is worthwhile to have a look at performance results in that context. We have extended the previous use case to a weak scaling use case for both libraries (figure 8). Note, that LAMA supports asynchronous execution and therefore overlapping of communication and computation. We have not seen something similar in PETSc. Furthermore, halo data is treated differently in PETSc.

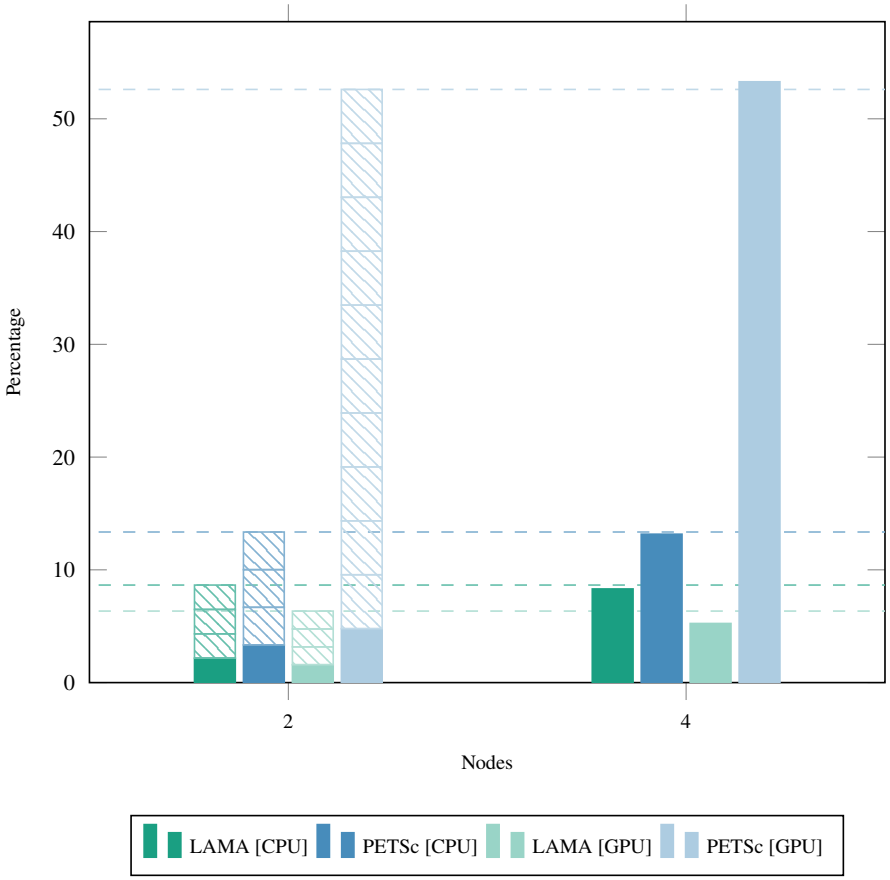


Fig. 8 Overhead for two/four node weak scaling LAMA - PETSc (CPU - CSR, GPU - ELL): executed on 3D 27-point poisson matrices with 10 million unknown per node. Small overhead (under 5%) for both, PETSc and LAMA, on the CPU and GPU for scaling on two nodes. On four nodes the overhead increase on CPU by a factor of 4 for both libraries. On GPU LAMAs overhead increase by less than factor 4 to 5%, while PETSc loses significant performance by more than factor 11 to 50%.

In this case we observe a much better scalability for LAMA in direct comparison with PETSc. The communication overhead has more influence as on the CPU. This has two reasons. In our setup the GPU is already much faster than the CPU. Furthermore, the overhead increases due to the down- and upload of halo exchange data from and to the GPU. For a more detailed view on handling the halo data in LAMA and further results see [5].

4 Summary

In this paper we presented the design, architecture, and performance comparisons of LAMA. We showed that with the abstraction of the memory model, compute kernels, and the introduction of asynchronous execution and communication it is possible to use LAMA for developing algorithms independently of the underlying hardware architecture. The central role for our memory abstraction model is given with the heterogeneous array. This allows us to introduce our concept of automatic up- and downloading of data to avoid invalid memory accesses.

That approach is novel and currently unique. It completely differs from our competitors implementations.

Our approach speeds up the development process since communication between heterogeneous hardware parts and across node boundaries have already been encapsulated in our library. At the same time this approach also reduces the risk of programming errors.

We showed that such a design introduces an overhead in the computation. Compared to our strongest competitor PETSc that overhead is fortunately negligible (CPU) or at least significantly small (single GPU). Moreover, the extra code for managing memory works in our favor when have a look at the multi-GPU results.

Our design introduces a kernel management composed of a registry with static registration at runtime. Employing `KernelTraits` in this case provides a high flexibility and ensures already at compile time that the right methods implementations are being used.

In LAMA we implement techniques for concealing necessary (and possibly time consuming) data transfers by making use of asynchronous execution. That is always advisable when working with accelerators with limited bandwidth interconnects.

Furthermore,

- the "separation of concerns" design of our tasking library enables that for any computation kernel.
- The library for inter-node parallelization handles the domain decomposition and communication.
- The underlying communication libraries are exchangeable and, hence, a developer can easily extend LAMA to support new hardware architectures.

The performance comparison with PETSc has shown that we achieve similar single node performance. The results of the performance comparison with single architecture implementations as well as PETSc underline, that achieving good performance results in general is not conflicting with a design focusing on portability and extensibility.

Currently, we are actively working on incorporating performance models for improving and adjusting the graph partitioning at runtime[3]. Application aware Co-scheduling of LAMA with other applications (or application parts) is another topic we currently investigate in the FAST project [2]. In addition, we continuously increase the number of available iterative solvers and pre-conditioners.

LAMA can be tested at `http://libama.scai.fraunhofer.de:8080/lamaui/`

Acknowledgments

This work was funded by the BMBF (German ministry of research and education) through the projects MACH, FAST, and WAVE.

5 Appendix

5.1 Test Environment

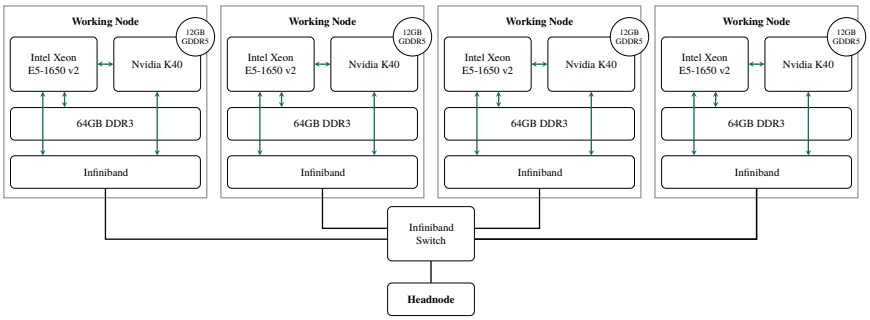


Fig. 9 Setup of the test system: four node hybrid HPC system. Each node consists of an Intel Xeon® E5-1650v2 with 64GB RAM and a Nvidia© K40 GPU (12GB GDDR 5 memory) as accelerator which is connected via PCI Express bus (15754 MB/s). The nodes are connected over infiniband.

Category	Name	Version
Compiler	gcc	4.9.1
	g++	4.9.1
	nvcc	7.0
Support	boost	1.58.0
Communication	OpenMPI	1.10.1
BLAS	MKL	11.1.2
	cuBLAS	7.0.28
Sparse-Matrix	cuSPARSE	7.0.28

Table 1 Used compilers and libraries for building LAMA and PETSc (both: development version of November 2015)

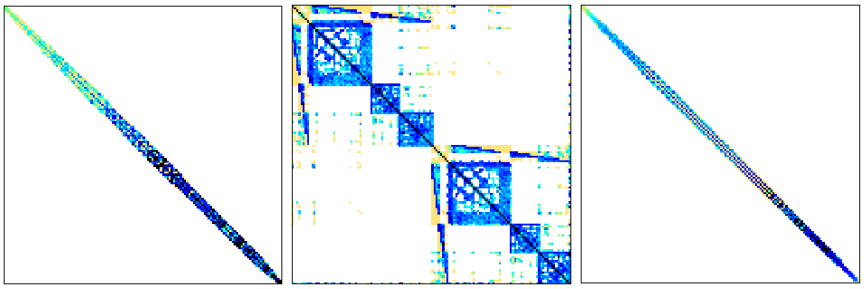
	Dimension		Nonzeros			Memory Usage	
Name	Rows	Columns	total	$\frac{\varnothing}{\text{row}}$	$\frac{\max}{\text{row}}$	CSR	ELL
Fault_639	638,802	638,802	28,614,564	44	318	332.34	2327.18
inline_1	503,712	503,712	36,816,342	73	843	425.17	4861.42
Emilia_923	923,136	923,136	41,005,206	44	57	476.31	605.70
ldoor	952,203	952,203	46,522,475	48	77	539.67	842.71
Serena	1,391,349	1,391,349	64,531,701	46	249	749.12	3970.07
Audikw_1	943,695	943,695	77,651,847	82	345	895.86	3729.51

Table 2 Representive set of large test matrices from the University of Florida Sparse Matrix Collection [6] (sorted by number of total nonzeros): memory usage in MB for double precision.

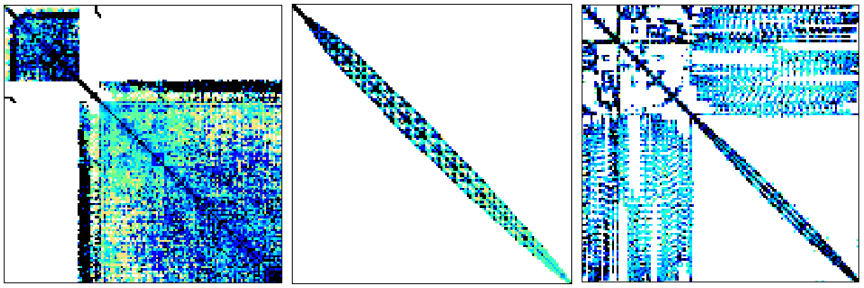
5.2 Test Matrices

References

1. Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang, *PETSc Web page*, http://www.mcs.anl.gov/petsc, 2015.
2. FaST consortium, *Find a suitable topology for exascale applications (fast)*, https://www.fast-project.de/, 2015.
3. WAVE consortium, *Eine portable hpc-toolbox zur simulation und inversion von wellenfeldern (wave)*, https://www.wave-project.de/, 2016.
4. George Karypis and Vipin Kumar, *A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs*, SIAM Journal on Scientific Computing **20** (1999), no. 1, 359—392.
5. Jiri Kraus, Malte Förster, Thomas Brandes, and Thomas Soddemann, *Using LAMA for efficient AMG on hybrid clusters*, Computer Science - Research and Development **28** (2013), no. 2-3, 211–220 (English).
6. University of Florida, *The university of florida sparse matrix collection*, https://www.cise.ufl.edu/research/sparse/matrices/, 2015.
7. Y. Saad, *Iterative Methods for Sparse Linear Systems: Second Edition*, Society for Industrial and Applied Mathematics, 2003.
8. Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer, *Top500, the list, november 2015*, http://www.top500.org/lists/2008/11/, 2015.
9. _____, *Top500, the list, november 2015*, http://www.top500.org/lists/2015/11/, 2015.
10. Todd Veldhuizen, *Expression templates*, C++ Report **7** (1995), no. 5, 26–31.



(a) fault_639 (b) inline_1 (c) Emilia_923



(d) ldoor (e) Serena (f) audikw_1

Fig. 10 Used test matrices.