International Journal of Parallel Programming (2021) 49:867–885
https://doi.org/10.1007/s10766-021-00718-x

Portable Node-Level Parallelism for the PGAS Model

Pascal Jungblut[1] · **Karl Fürlinger**[1]

Received: 1 October 2020 / Accepted: 21 May 2021 / Published online: 5 June 2021

Abstract
The Partitioned Global Address Space (PGAS) programming model brings intuitive shared memory semantics to distributed memory systems. Even with an abstract and unifying virtual global address space it is, however, challenging to use the full potential of different systems. Without explicit support by the implementation node-local operations have to be optimized manually for each architecture. A goal of this work is to offer a user-friendly programming model that provides portable performance across systems. In this paper we present an approach to integrate node-level programming abstractions with the PGAS programming model. We describe the hierarchical data distribution with *local patterns* and our implementation, MEPHISTO, in C++ using two existing projects. The evaluation of MEPHISTO shows that our approach achieves portable performance while requiring only minimal changes to port it from a CPU-based system to a GPU-based one using a CUDA or HIP back-end.

Keywords PGAS · Parallel computing · Programming models

1 Introduction

Porting performance critical software to new architectures is a challenging task. Programming abstractions like OpenMP provide means to decouple algorithms from implementation details to ease the transition. Yet, the many combinations of system configurations, back-ends and implementations force programmers to modify their code for acceptable performance. In this paper we present the integration of abstractions for node-level parallelism into the Partitioned Global Address Space

✉ Pascal Jungblut
pascal.jungblut@nm.ifi.lmu.de

Karl Fürlinger
karl.fuerlinger@nm.ifi.lmu.de

[1] MNM-Team, Computer Science Department, Ludwig-Maximilians-Universität (LMU) München, Oettingenstr. 67, 80538 Munich, Germany

(PGAS) programming model. This allows us to easily select the best *back-end* (e.g. OpenMP or CUDA) for parallelism and use the global address space at the same time.

1.1 Partitioned Global Address Space

In PGAS each computer dedicates a part of its local memory to a virtual, global address space. In this global space, all processes may access memory locations on remote locations. Each object in the global space is *owned* by one process or more general a rank on which it is stored. Although all global memory is accessible from all nodes, it is desirable to operate primarily on local data for performance reasons. This stems from higher latency and possibly lower bandwidth for remote accesses compared to local ones. Note that remote accesses may also refer to other NUMA-domains on the same node. Often the so-called *owner-computes* rule is used to determine which process is responsible for the computation of a data point. It states that the process writing to an object will perform the computation.

There exist several PGAS implementations, both as dedicated programming languages like Unified Parallel C (UPC), Co-array Fortran and Chapel or as a library for existing languages like Global Arrays or Hierarchically Tiled Arrays (HTA). Although PGAS does not dictate how remote objects are accessed, many libraries use one-sided message passing. *One-sided* implies that only one of the communicating partners is active, i.e. it initiates the transfer, passes all necessary parameters and monitors the progress. These one-sided operations are often implemented using Remote Direct Memory Access (RDMA) so the target of an operation can be truly passive. Hence the semantics are often similar to shared memory programming where threads may read and write arbitrary shared memory at any time, also requiring some form of explicit synchronization.

Figure 1 shows a distribution of an array across nodes. It also includes node-local private memory that cannot be read from other processes. Here, node 2 holds a private integer.

1.2 Abstractions for Node-Level Parallelism

PGAS processes can be mapped to nodes, NUMA-domains, cores, or SMT threads. It can be beneficial to assign processes to NUMA-domains and use shared memory parallelism inside these to avoid costly cross-NUMA accesses. If we take this idea

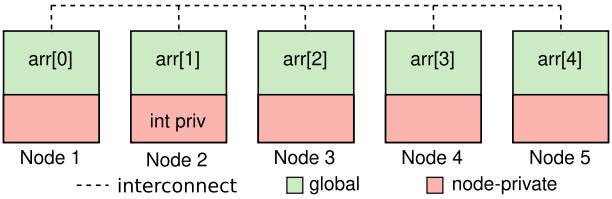


Fig. 1 An array of 5 elements distributed across 5 nodes using the PGAS model

one step further, the algorithms may use the hierarchical structure of the hardware to maximize locality.

Programming for accelerators often require either low-level and/or vendor-specific languages and libraries like CUDA or OpenCL. These give programmers a high level of control, are often tailored to the requirements of the hardware and thus offer a lot room for optimization. However, the portability is limited: even for languages that are not vendor specific, executing the same code on other hardware necessitates a compatible implementation. Higher-level models like SYCL and Alpaka build on top of these but hide the specific back-end. This potentially allows users to migrate code more easily from one to another platform. Even on the same hardware this can be useful, because it allows a fast evaluation of all supported back-ends. As a side-effect these abstractions use a single-source model. This means that the whole program, including the kernels, are written in one programming language, e.g. C++.

We introduce the combination of abstractions for node-level parallelism and the PGAS programming model. This approach offers flexibility for porting codes to new distributed memory architectures. Our main contributions are:

1. A unified, hierarchical system for data placement across nodes and compute units
2. A flexible distribution of the workload
3. An interface to integrate shared memory parallelism with the PGAS model

To have fine grained control over threads and accelerators, programmers often have to resort to manually extending the PGAS environment, similar to an MPI+X approach that is employed in message passing solutions. We want to provide a usable, flexible and abstract integration of the aforementioned abstractions. For that we allow the implementations to dynamically transfer the ownership of data to an accelerator or other processing elements. The details of this idea are described in Sects. 2.3 and 2.4.

We first detail our approach. In Sect. 3 we describe the C++ implementation using two existing libraries along with an evaluation in Sect. 4. Related approaches are discussed in Sect. 5 and finally we conclude this paper and offer an outlook for future work.

2 Approach

To integrate node-level parallelism kernel acceleration into the PGAS programming model, we extended the latter to allow the distribution of local work to accelerators. Supported algorithms can hand over the control along with a task to *executors* that will gain ownership of (some part of) the data. During this phase it is prohibited for any other entity than the executor to access the data, so data that is needed outside of an executor must be copied before. Accelerators then work asynchronously on the tasks. If an accelerator and the host do not share the same memory space, data must be moved to the accelerator's memory before the execution starts. Conversely, the results from the execution must eventually be copied back to the host. However,

skipping redundant copies from accelerator to host or vice-versa is an optimization available in case a coherent shared memory space is available.

2.1 Definitions

Within one node there can be several *types* of processing elements (PE). We define that two PEs have the same type if they have the same computational capabilities and memory space, e.g. two cores on the same CPU have the same type while a CPU-core and a GPU-thread have different ones. Performance differences from manufacturing or changes in frequency are not considered in this paper. These could nonetheless be included if the focus was more on load balancing. Processing elements or *groups of the same type* of processing elements on a node l are called *entities* E_l. Entities can be freely defined as long as the constraints above are honored. For example it might be useful to define the PEs of CPU socket, a NUMA-domain or simply a group of cores as one entity. Each PE of an entity E is called an *instance* e_i.

2.2 Requirements

Both the PGAS implementation as well as the node-level abstraction need to fulfill some requirements for our approach to work. This is the case for most of the widely used software. We go over some of it in Sect. 5. The PGAS implementation needs to support two features:

- *Persistent and predictable data layout* the runtime is not allowed to move allocated memory from one process to another. The software may support policies for the global data layout or let the user specify it manually.
- *Explicit synchronization* the interoperability of the local and global part requires support for synchronization between processes.

On the node-level the requirements are:

- *Non-blocking memory management and kernel invocation* our approach assumes that kernel invocation and memory movement may both be non-blocking. To our knowledge this is supported natively by all widely used implementations.
- *Separate memory space or coherence with PGAS* the node-level abstraction must either support memory spaces per device or we expect the memory to be coherent with the global memory view, e.g. CUDA-aware MPI.

The requirements for the PGAS implementation are motivated by the support of unaware local implementations: if the data was moved during the execution of the local framework, this would lead to race conditions. On the node we must assume a symmetric situation where the two systems (local and global) are unaware of each other. Thus, it is required to either have explicit support for separate memory spaces or to provide a coherent view of the node-local memory to the PGAS library.

2.3 Data Placement

A crucial part of the interaction between the node-level back-end and the PGAS environment is the strategy for data placement. How inter-node and intra-node distribution of data is configured has a large influence on the achievable performance and compatibility. The owner-computes model already motivates the usage of favorable layouts for many hardware topologies. The strategy to avoid costly remote accesses implies that data locality is a priority.

PGAS implementations let the user specify data placement configurations to varying degrees. Some like UPC derive the data distribution from the number of elements and the number of nodes. Others allow more fine grained control up to Hierarchically Tiled Arrays where each hierarchy level represents a hardware level. DASH implements the *pattern concept* which lets the user map each element of a container to an arbitrary location in global memory. We extend this pattern concept to work with entities instead of processes (i.e. MPI ranks). As a basis we use the pattern concept as described in previous work [8]. The mapping of data onto global memory location is a three-step process: first, each element in a container is assigned to a location in the global index space. Second the index space is divided into blocks which are finally distributed across units. This approach is suited best for a one-to-one mapping from processes to accelerators, but is not flexible enough for more complicated setups. Current HPC-systems may have two or three accelerators per CPU-socket.

We extended the (global) patterns that describe the mapping from the global domain space to global indices by *local patterns*. A local pattern P_l maps node-local blocks B_n as defined by the global pattern P onto entities: $P_l : B_n \mapsto E$. Note that the local patterns only consider node-local blocks so this mapping is independent from the global memory layout. Additionally, the mapping of a block does not imply any data movement. Only when the data is requested to be *owned* by an algorithm, data may be moved to an entity's memory space.

The motivation for this hierarchical approach is two-fold. First each unit n may define a different pattern for its blocks B_n. This could be due to algorithmic optimizations or resource imbalance. Second, for the same blocks multiple local patterns may be defined, because no data movement takes place. This is especially useful to balance the load individually per algorithm. For example, if there are $b_l = |B_n|$ local blocks, one local pattern may assign $w_{CPU} \cdot b_l$ blocks to the CPU and $w_{GPU} \cdot b_l$ blocks to the system's GPUs and change w_{CPU} and w_{GPU} between invocations. A pattern may define an arbitrary amount of blocks b_l for a container.

Local blocks that are assigned to an entity with a separate memory space must be copied before usage; either explicitly by issuing an appropriate call or implicitly by allocating memory in a compatible memory space. For example all recent CUDA-enabled devices support *unified memory* to transparently migrate pages from the host memory space to the CUDA-device's space. If this is not available or wanted, for example due to limitations of the PGAS implementation, it is possible to track local blocks that were already copied to an entity's memory space. The runtime may then eliminate redundant migrations. Unified memory may have a negative performance impact, since some runtime has to keep track of page faults and

migrate memory pages on demand. To counter this, we can use prefetching when an entity requests ownership of a block, e.g. using `cudaMemPrefetchAsync`. However, the evaluation shows that excessive prefetching may have a negative impact on the scalability due to congestion on the memory bus.

Figure 2 shows a local pattern of a 2-dimensional array on node n. It contains 4×4 contiguous blocks of memory. The top-left ones are mapped to the OMPEntity while the bottom and right blocks are mapped to two separate instances of the CUDAEntity (i.e. two CUDA-enabled GPUs). Exemplary the seven blocks assigned to CUDAEntity will be prefetched before the kernel execution is started.

For the evaluation we implemented a flexible local pattern. Here b_i denotes the i-th local block of a node-local process with index p and t the total number of instances of the mapped entity E. The local pattern maps b_i to $e_k \in E$.

- An *identity*-pattern with $k = i$. This is only valid if $|b_l| = t$.
- A *round-robin*-pattern with $k = i - t\lfloor \frac{i}{t} \rfloor$. Trivially, if $|b_l| = t$ this is equivalent to the identity pattern.
- An *x-per-process*-pattern with $k = i - x\lfloor \frac{i}{x} \rfloor + px$.

The x-per-process-pattern is useful for cases where the number of node-local processes is less than the available entities. It distributes x consecutive local blocks to each entity. The evaluation in Sect. 4 contains an example with the DGX-1 system where the total runtime is optimal for four processes per node with eight GPUs.

2.4 Computation

The definition of a local pattern does not imply any data movement. Transfers only happen when an algorithm schedules work on local blocks. The execution of an algorithm is split into three phases:

1. *Initialization* Allocate memory, copy missing blocks to the desired memory space and initialize local variables.
2. *Computation* Pass ownership to assigned entities and execute kernels using the executors.

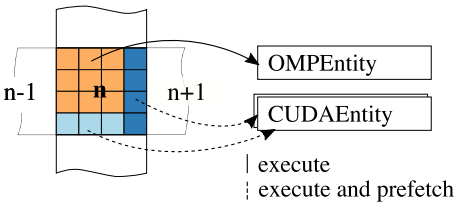


Fig. 2 2-dimensional local blocks of memory on node n are executed on different entities. The local pattern describes the exact assignment (indicated by color) to entities. Depending on the memory spaces the blocks will be prefetched or copied to the entities memory. Here, two instances of CUDAEntity exist which represent two distinct GPUs

3. *Finalization* Copy the result buffers to the host memory space and release ownership of the local blocks.

During the computation phase, accesses to non-owned blocks are only allowed if the owning entity does not use the same memory space, i.e. there exists a separate copy on the device memory of the mapped entity. For all other accesses the memory must be copied beforehand. Algorithm 1 shows how the execution is scheduled in more detail. This strategy is also employed in pure PGAS applications, e.g. by copying the ghost cells in a stencil application in chunks to avoid the latencies for element-wise remote accesses.

Algorithm 1 Execution of local blocks on their assigned entities

```
Initialize
entities ← E_l
for all entity ← entities do
    b_e ← BlocksForEntity(entity)
    for all b_{e_n} ← b_e do
        Block until entity gets ownership over b_{e_n}
        if entity shares memory space with b_{e_n} then
            Prefetch b_{e_n} to entity
        else if Block b_{e_n} currently not in entity's memory then
            Copy b_{e_n} to entity
        else
            Do nothing
        end if
        Execute kernel on entity with b_{e_n}
        Release ownership of b_{e_n}
    end for
end for
```

3 Implementation

We implemented this approach using two already existing libraries. The PGAS library used here is DASH. For the compute-intensive kernel operations, we chose Alpaka. Both are pure C++ libraries that require a C++14 compatible compiler. We will briefly describe both frameworks in the following section and how we integrated them as MEPHISTO.

Figure 3 shows the overall architecture. MEPHISTO is a thin layer on top of DASH and Alpaka which in turn both support multiple backends. DASH handles

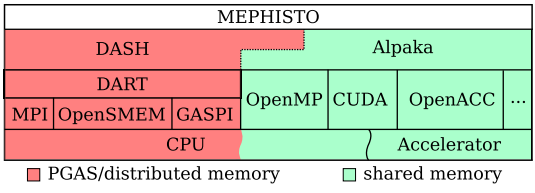


Fig. 3 MEPHISTO architecture with DASH and Alpaka

global inter-process communication using PGAS libraries such as one-sided MPI and Alpaka supports parallelization through its backends while providing the user (DASH in this case) a unified interface.

3.1 DASH

DASH is a general purpose C++14 PGAS library that implements distributed data structures and optimized algorithms similar to the Standard Template Library (STL). It is built on top the DASH run time (DART) which supports a range of distributed memory abstractions like one-sided MPI, OpenSHMEM or GASPI. The containers like `dash::array` and `dash::map` are compatible with their STL counterparts, so they can be used with STL algorithms. However, the STL is not optimized for the PGAS environment: DASH algorithms minimize remote access and may employ low-level implementations for better performance. For example `dash::transform_reduce` will use the reduce implementation of the underlying technology, e.g. `MPI_Reduce`. Listing 1 shows a simple program using a DASH array and two algorithms to generate a sorted array of random numbers.

```
1 dash::Pattern<2> blocked(8, 12, dash::BLOCKED, dash::NONE)
2 auto array = dash::array<double>(blocked);
3 dash::generate(array.begin(), array.end(), random_gen);
4 dash::sort(array.begin(), array.end(), array.begin());
```

Listing 1: A DASH array with 8×12 elements is created where the first dimension is specified as blocked. The distributed array is filled with random numbers and sorted using PGAS-aware algorithms.

The data distribution across the processes is specified with the DASH patterns. Listing 1 shows how a distributed array is allocated with the data layout defined by the 2-dimensional `dash::Pattern`. The data is partitioned *blocked* in the first dimension, i.e. each of the P processes holds at most $\frac{1}{P}$ consecutive elements in this dimension.

It is then filled with random numbers using `dash::generate` algorithm and finally sorted by `dash::sort`.

3.2 Alpaka

The Abstraction Library for Parallel Kernel Acceleration (Alpaka) [20] is a C++14 header only meta-programming library for node-level parallelism. It supports several back ends like C++ `std::thread`, OpenMP 2, OpenMP 5 with offloading, Boost Fiber, CUDA or HIP.

In Alpaka, the user describes kernels as plain C++ functors and uses Alpaka's interface for interactions such as synchronization, data movement or kernel invocations. During compile-time one selects a back-end using C++'s type system. In Listing 2 the selected back-end is the serial CPU accelerator (line 2) which will execute the kernel single threaded on the CPU. To use another back-end, for

example CUDA, one would only need to change `AccCpuSerial` to `AccGpuCudaRt`.

The programming model is similar to the one CUDA and OpenCL offer, i.e. the work is split up into multiple threads per block and blocks per grid. There are some conceptual extensions to support different platforms: a kernel executed on a CPU thread typically yields the best performance when it operates on many contiguous memory locations, negating the overhead of thread management. In contrast the programming model for GPUs encourages a mapping of one element per thread. Alpaka offers an additional *execution layer* that allows looping over elements. For a GPU-based back-end the loop-size would typically be one and, since it can be set at compile time, the loop is optimized out or expresses vectorization. Note that Alpaka's blocks do not correspond to local blocks as described in Sect. 2.1 but to the concept known from CUDA.

```
1 using Dim = alpaka::dim::DimInt<3>;
2 using Acc = alpaka::acc::AccCpuSerial<Dim, size_t>;
3 // MyKernel is a C++ functor
4 MyKernel kernel;
5 alpaka::kernel::exec<Acc>(queue, workDiv, kernel);
```

Listing 2: Invocation of an Alpaka kernel. Type of accelerator is set at compile time using only types. Here `workDiv` configures the decomposition into grids, blocks and thread-elements.

Alpaka provides queues that are conceptually similar to CUDA streams. A queue belongs to one accelerator, i.e. one particular device, and schedules kernels to execute. Both blocking and non-blocking queues are provided. In this integration we exclusively use the non-blocking queues so the synchronization can be managed by MEPHISTO.

3.3 Integration

This work aims to integrate PGAS with node-level abstractions for portable performance between system configurations. As a prototype we integrated Alpaka and DASH into MEPHISTO. In this section we describe the technical details of the approach outlined in Sect. 2.

To implement the local patterns we extended the existing patterns in DASH. A local pattern inherits from a global pattern and extends it with one essential method:

```
1 template<typename Entity>
2 std::vector<block_spec> LocalPattern::blocks_for_entity(
3   const Entity &e);
```

Listing 3: Method to retrieve the local blocks for an entity.

This method can be implemented for each entity, for example `CudaRTEntity` or `NumaEntity`. Assigning a new local pattern is a non-collective operation so no synchronization is required. With `blocks_for_entity` the DASH algorithms can retrieve the mapping of local blocks to entities and gain ownership.

We used the concept of C++'s *execution policies* and *executors* to allow users to specify on which entities an algorithm should execute. Execution policies can be used to relax the guarantees given by the STL, e.g. to have `std::for_each(std::execution::par_unseq, begin, end, f)` apply `f` in parallel and possibly vectorized over `[begin, end)`. The policies can be extended by executors to specify *where* an algorithm is executed. At the time of writing none of the executor proposals have been standardized. We extended one of them[1] to integrate an `AlpakaExecutor` that can be attached to an execution policy. An `AlpakaExecutor` can be created for each entity. It holds state, e.g. a thread pool, and may exist across algorithm invocations. Note that other executors (e.g. a `KokkosExecutor`) are possible just as well. However, as part of MEPHISTO, we only ship the `AlpakaExecutor`.

Inside the algorithm's implementation the three phases are executed as described in Sect. 2. We obtain the blocks with `blocks_for_entity` for each entity and start the kernels using Alpaka. In our implementation the execution devices from Alpaka are directly mapped to entities.

```
1  auto local_blocks   = pattern.blocks_for_entity(entity);
2  auto nlocal_blocks  = local_blocks.size();
3  std::vector<result_t> lres{nlocal_blocks};
4  for(int i = 0; i < nlocal_blocks; i++) {
5    executor.request_ownership(entity, local_block[i]);
6    executor.execute_kernel(reduction_kernel(user_func), entity,
7      local_block[i], &lres[i]); // releases ownership
8  }
9  executor.synchronize();
10 result_t lresult = std::min_element(lres.begin(), lres.end(),
11   result_t{}, user_func);
12 return reduce_global(lresult, user_func);
```

Listing 4: Simplified excerpt from the MEPHISTO-enabled `dash::min_element` algorithm internals.

Listing 4 shows a simple invocation of Alpaka inside of DASH. For our prototype we extended DASH's `for_each`, `transform`, `reduce` (and similar algorithms) and `transform_reduce` which have equivalent semantics as the STL variants. The prototype also provides three predefined Alpaka-enabled executors for CPU (serial, OpenMP) and CUDA. The code resembles the three phases outlined in Sect. 2.4: all blocks for the current entity (`entity`) are loaded and the ownership is requested. In this case this is a blocking call, but the request for ownership may be wrapped into a `std::future` to asynchronously wait and start executing the kernel after that. The call to `execute_kernel` hands over the control to Alpaka. It calculates a work division, i.e. the dimensions of elements per thread, threads per block and the number of blocks, based on the input size and the characteristics of the entity. For example, for entities that use offloading to GPUs the elements-per-thread extent will be set to one while for CPU-based OpenMP variants the elements per thread will be maximized. The algorithm synchronizes

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r12.html.

after all local blocks have been reduced by the entities. This is a node-local barrier. The results of all entities are then reduced again and finally a global result is calculated using the existing global `dash::min_element`.

3.4 Usage

Listings 2 and 4 show internals of the integration that are transparent to a user of the library. The interface of the containers and algorithms is very similar to DASH programs like the example in Listing 1. The changes required to offload the invocation of a DASH-algorithm to an entity are small:

- Specify a local pattern for the container.
- Extend the call to the algorithm with an execution policy.

To extend the example from Listing 1 we add a round-robin local pattern and pass the policy with the executor attached to the algorithms.

```
1  using Entity = mephisto::entity::CUDA;
2  dash::Pattern<2> blocked(8, 12, dash::BLOCKED, dash::NONE)
3  mephisto::local_pattern::round_robin<Entity> pattern(blocked);
4  mephisto::execution::par_unseq<Entity> par_unseq();
5  auto array = dash::array<double>(pattern);
6  dash::generate(par_unseq, array.begin(), array.end(), random_gen);
7  dash::sort(par_unseq, array.begin(), array.end(), array.begin());
```

Listing 5: MEPHISTO-enabled version of Listing 1.

Listing 5 shows the same example with MEPHISTO enabled. Line 1 specifies the entity that should be used within the executor. Only this line would need to be changed to offload to other entities. Lines 3 and 4 create the local pattern and initialize an execution policy. The execution policy is passed to both algorithms as a first parameter, similar to the standard C++ execution policies.

4 Evaluation

We evaluated the prototype using several systems:

SuperMUC-NG at LRZ is based on a Intel Xeon 8174 (Skylake-SP) with 48 cores at 2.7 GHz. The benchmarks were compiled with Intel ICC 19.0.5.281 and executed with Intel MPI 2019. Each node has 96GB of memory.

HAWK of the HLRS consists of 5,632 nodes with two AMD EPYC 7742 CPUs at 2.25 GHz with 64 cores each. We tested several compilers and OpenMP-implementations and found ICC 19.1.0.166 and AOCC (Clang) 2.1.0 with the best consistent results. The reported results were compiled with ICC.

DGX-1 P100 from NVIDIA contains eight Tesla P100 GPUs with 16 GB HBM2 cross-connected with NVLink. The host CPUs are two 20-core Intel Xeon E5-2698. **DGX-1 V100** is very similar to DGX-1 P100: it contains Tesla V100 with 16 GB HBM2 instead. The code was compiled with ICC 19.0 for the host code and the CUDA back-end with version 10.2 on both systems.

Rome consists of two nodes with two AMD EPYC 7742 with 64 cores each at 2.25 GHz. Each node contains two AMD Radeon MI-50 connected via PCIe4.

4.1 Reduction

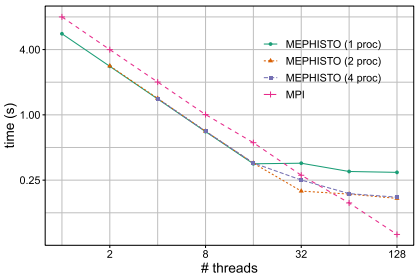
As a micro-benchmark we implemented a reduction using `dash::transform_reduce` and observed the scaling behavior as well as the portability across architectures. The operation takes a unary function `unary(elem)` for the transform and a binary operation `binary(accum, elem)` for the reduction. One can lower the number of slow memory accesses by computing `binary(acc, unary(elem))` for each element in a block and use a tree reduction for the block, grid, entity level and finally the global level for the global result. For a given array *arr* of size *a* in this scenario the `transform_reduce` computes $\sum_{i=0}^{a-1} \frac{arr_i}{arr_i^2+1}$.

Figure 4 shows the total run time of a reduction for 10 GB total (strong scaling) and 20 GB per process (weak scaling) on up to 256 nodes in HAWK and SuperMUC-NG. In both strong scaling studies the overhead of MEPHISTO's backend (OpenMP in this case) becomes visible as we add more threads per process. Due to Alpaka's optimized reduction kernel and its zero-overhead abstractions the total run time is slightly lower than a pure MPI implementation up until 32 threads per process.

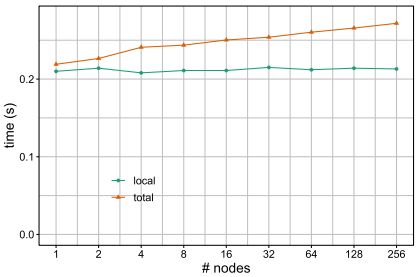
For HAWK (Fig. 4b) we chose the fastest combination of 4 processes each with 32 threads per process. The graph also shows the purely local portion of the computation, up until the global reduction. As expected it stays nearly constant regardless of the number of total nodes. The same effect can be seen on SuperMUC in Fig. 4d. Here we used 32 threads and one process per node so all node-level parallelism was managed by Alpaka. For the final global reduction MEPHISTO uses an `MPI_Allreduce` so the growing overhead can be traced back to that call.

One central motivation of MEPHISTO is the optimal support for different architectures and thus portability. To demonstrate the feasibility and portability of the approach we also evaluated the transform-reduce implementation on Nvidia's DGX-1 nodes with P100 and V100. Figure 4f shows the execution time for a transform-reduce over the problem size on one P100. We compare MEPHISTO (with and without prefetching) to Thrust [2]. This library comes bundled with CUDA and implements STL-like algorithms optimized for the execution on CUDA-enabled GPUs. We used `thrust::transform_reduce` to perform the same operation as with MEPHISTO.

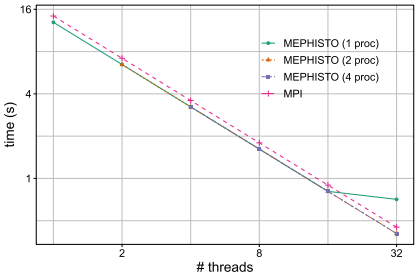
Block-wise prefetching has a positive effect on the performance across all problem sizes. However, especially for small problem sizes Thrust's implementation is the fastest alternative. Note that Thrust's containers manage memory on their own and do not rely on unified memory. Therefore we include data movement between host and device but no allocation in the run time which reduces Thrust's run time compared to MEPHISTO. Scaling up from one GPU per node to up to six as depicted in Fig. 4g reveals two characteristics. First, prefetching does not offer a speed-up in all scenarios. Especially when the number of concurrently used GPUs is large, prefetching has a clear negative effect on the performance. Second, the



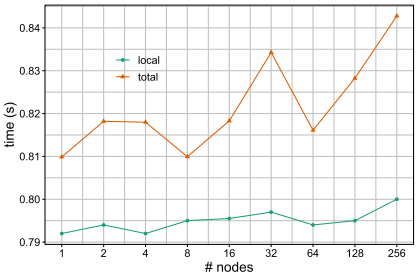
(a) HAWK: strong scaling with 10 GB.



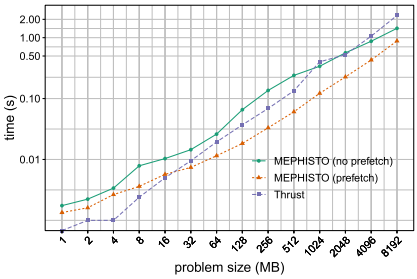
(b) HAWK: weak scaling with 4 processes and 20 GB per process.



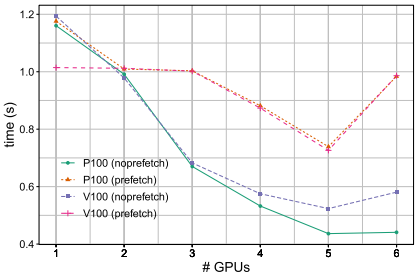
(c) SuperMUC-NG: strong scaling with 10 GB.



(d) SuperMUC-NG: weak scaling with 4 processes and 20 GB per process.



(e) One Nvidia P100



(f) DGX-1 V100 vs P100

Fig. 4 Evaluation of MEPHISTO's transform-reduce implementation. The top figures show shows strong scaling on a shared memory node of HAWK (**a**) and weak scaling with 20GB per process on the same system (**b**). The figures below show the same benchmarks on SuperMUC-NG. **e** shows a comparison between Thrust and MEPHISTO with and without prefetch enabled whereas **f** shows the scaling on a different number of GPUs

parallel efficiency is not ideal with this problem size of 10GB. Further investigation indicates that the PCI-e link between the host memory and the GPUs becomes congested. The effect is amplified when all GPUs start prefetching whole blocks during the execution.

During the evaluation it became clear that the architecture of DGX-1 limits the throughput for a single process. On DGX-1 each CPU is connected to four P100/

V100 via two PCI-e switches. The CPUs in turn are connected with Intel's QuickPath Interconnect (QPI) and the GPUs with NVLink. With one process pinned to CPU_0, all blocks assigned to GPU 4, 5, 6 and 7 must be sent over QPI to the second socket and PCI-e to the GPUs. With the *x-per-process*-pattern the benchmark can be started with one, two, four or eight processes and with eight, four, two or one GPU per process. The evaluation in Fig. 5 shows the benefit of each configuration, especially when working with prefetching. With one process assigned to each GPU the PCI-e lanes become congested. Even with two processes (one per CPU) there is still a higher runtime than with a 4×2 configuration. This matches the topology directly and shows the need for flexible process-to-accelerator assignments.

4.2 Heat Equation

We used a 2-dimensional heat equation calculation to further evaluate different entities with local and global communication. The algorithm is a double buffered 5-point stencil application that uses blocking for temporal data locality within each thread. Again we use DASH to span a global array across all processes and execute the kernels using Alpaka. Thus, the global communication for neighbor-exchange across processes is handled by DASH and node-local this is done with Alpaka. The tests were conducted on the Rome system with AMD AOMP 11.9 as the compiler for OpenMP 2 and 5 (offloading) and hipcc 4.1 (Clang 12), because Alpaka requires HIP version 4.0 and up.

Figure 6a shows the per-node scaling on two nodes on the Rome system for a different number of threads using the OpenMP 2 entity. It is visible that the scaling is limited due to the neighbor exchange as the surface between the processes grows with its number. The shared memory parallelism performs well, because the neighbor exchange can be done in-memory. Figure 6b shows the execution of the same algorithm in three different MEPHISTO configurations: purely on the CPU (equivalent to Fig. 6a) using the OpenMP 2 backend, one with OpenMP 5 with offloading to the GPUs and one HIP backend. The OMP 5 backend has considerably longer runtimes compared with both the OMP 2 version and the HIP backend

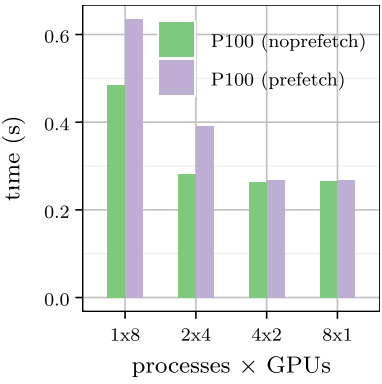
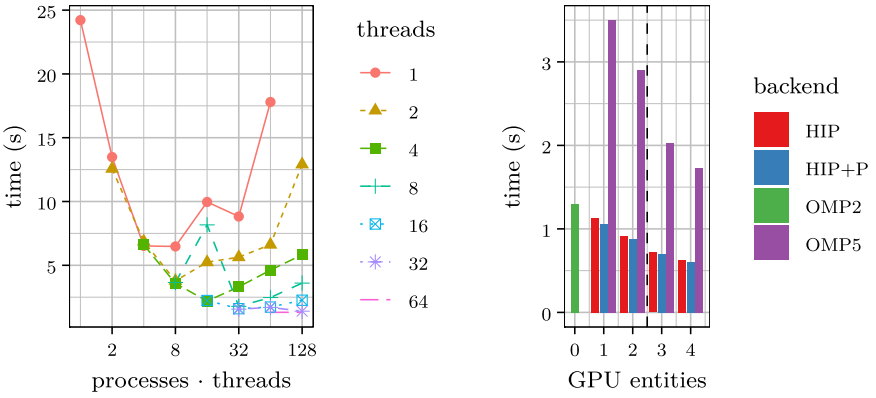


Fig. 5 Runtime of MEPHISTO's transform-reduce combinations of the number of processes and GPUs per process



(a) Combination of CPU-threads and processes for the heat equation on a Rome node.

(b) Strong scaling across two nodes with global communication.

Fig. 6 Heat equation computed on the Rome systems

(HIP+P with prefetching). We were able to reproduce similar effects with other workloads outside of MEPHISTO. Even with simultaneous execution on the CPU-entity the benefit of OMP 5-offloading was marginal, whereas the HIP backend performed better than the CPU-based version. Prefetching of the buffer holding the exchanged neighbors yielded a slight improvement. OpenMP 5 does not support explicit prefetching.

4.3 MiniMD

As a more complex case we ported MiniMD [5] to DASH and then used MEPHISTO to add shared memory parallelism. For that less than 1% of the 1500 LOC needed to be changed:

- Choose one of the pre-configured entities (e.g. using OpenMP) to be used.
- Specify which calls to DASH's algorithms should be performed in parallel using executors.
- Optional: change the data layout using DASH's patterns.

MiniMD is a molecular dynamic proxy application that mimics the workload of LAMMPS [16]. The 3d-space is decomposed into a configurable number of cells. Each cell holds a dynamic number of atoms. To update the position and velocity of each atom, the force of the surrounding atoms is calculated. MiniMD configures a cut-off distance r_{cutoff} and only calculates forces of neighbors inside this radius. Because the space domain is already split into cells, only the cells within r_{cutoff} have to be considered, reducing the complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ for n atoms. For our evaluation we used the default configuration of the reference implementation that requires each cell to consider at maximum 27 neighbor cells. The atoms are stored in a `dash::NArray` with four dimensions: three dimensions representing cells

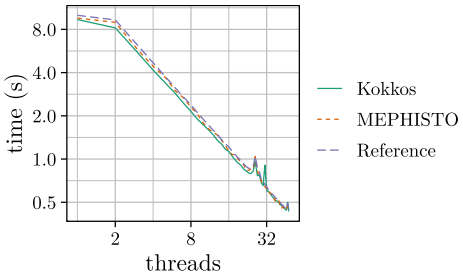


Fig. 7 Comparison of the shared memory runtime of the reference, Kokkos and MEPHISTO implementations on SuperMUC-NG

and one for atoms in each cell. An `NArray` is static in size so we re-allocate when a cell overflows during the binning process.

Figure 7 shows the comparison of the reference implementation and the Kokkos and MEPHISTO ports on one node of SuperMUC-NG with an increasing number of threads. The anomalies at around 24 threads point to a NUMA-related issue, although we observed similar spikes for two socket-pinned MPI processes (with half the number of threads) on all variants. Here all variants perform similar. We tested several other combinations of processes and threads and all performed similar or minimally slower.

5 Related Work

Similar to Alpaka is Kokkos [6], an open source abstract interface for shared memory programming. For the details on the differences between both, refer to [20]. SYCL[2] is cross platform and built on top of OpenCL that supports potentially a wide range of accelerators. In contrast to OpenCL, it is also single-source. SYCL is a part of Intel's OneAPI approach for heterogeneous, parallel programming. Thrust [2] is a library that implements STL-like algorithms with a CUDA, Thread Building Blocks or OpenMP back-end. Because it ships its own data structures, the compatibility with most PGAS implementations will be limited. OpenMP itself is an open standard for shared memory programming. Version 4.5 introduced the *target* environment with data regions which can be used to allocate device memory and offload computation to accelerators. It does not yet support the allocation and usage of block-wide memory on CUDA devices, potentially impairing the performance on these devices. Very similar to OpenMP is OpenACC[3], also an open standard, which does provide access to block-shared memory. All of these offer no integration with distributed memory paradigms. We support OpenMP 2, OpenMP 5 and TBB through Alpaka as backends. PACXX is an abstraction for many-core systems that uses an extension to Clang and LLVM to provide similar, high-level features in C++ [9]. Conceptionally, MEPHISTO could integrate PACXX instead of Alpaka. However, PACXX requires a custom compiler whereas Alpaka works with a wide range of C++ compliant compilers and backends.

[2] https://www.khronos.org/sycl/.

[3] https://www.openacc.org/.

There are also higher-level skeleton programming environments that can use multiple backends. SkePU supports parallel OpenMP execution on the CPU and offloading to GPUs with OpenCL and CUDA. Because SkePU is also pure C++, an integration in MEPHISTO is conceptionally possible [7]. Other GPU-capable approaches are the Muenster Skeleton Library [19], StarPU [1] and SkelCL [17]. The latter extends OpenCL for better support of multiple GPUs but is limited to one node. StarPU is task-based and works on distributed systems through MPI. TuCCompi provides a multi-level programming environment for distributed and heterogeneous systems [15]. The global communication is handled via message passing (as opposed to PGAS). The local computation is done with OpenMP and CUDA. It requires the programmer to write specific code for CPUs and GPUs. Skeleton frameworks decouple the description of the algorithm from its mapping to the hardware. MEPHISTO is more transparent about this mapping, e.g. the local part of a PGAS container is often explicitly addressed by a user.

PGAS is either implemented as a programming language or in a library for an existing language. UPC [18], Co-array Fortran [14], Chapel [3], and X10 [4] are languages with built-in PGAS support. All have configurable data placement with respect to the nodes, making it usable for our approach. However, these languages require a special compiler that limits the portability. High-level libraries such as DASH, Global Arrays or UPC++ provide PGAS support without the need for a special compiler. They ship with distributed data structures and algorithms. There are also more low-level ones such as GASPI or one-sided MPI. These are used as a provider by the aforementioned libraries. The integration of the shared memory abstractions with our approach is possible with all of these, because the only requirements for the PGAS implementation are user-defined data layouts and a form of synchronization between processes. All of the above offer both.

There has been some work on combining shared memory parallelism in a PGAS environment, much like MPI+X with the message passing model. The MiniMD application was ported to PGAS in [12] using UPC and POSIX threads. Jose et al. implement a PGAS runtime to achieve considerable speedups due to shared memory support [10] These focus on specific implementation whereas MEPHISTO is agnostic to the back-end. Most similar to our work is [13]. The authors combine already mentioned directive-based OpenACC with XcalableMP to XcalableACC, enabling offloading to accelerators. It requires a dedicated compiler but offers accelerator-to-accelerator communication which we do not explicitly support. HPX [11] supports distributed job scheduling but uses *Active Global Address Space*, thus violating our requirement that allocated memory is not moved by the runtime.

6 Conclusion

6.1 Summary

In this paper we present our approach to integrate node-level abstractions for parallelism with the PGAS programming model in a user-friendly way. It extends existing methods for data distribution to include *local patterns* that map contiguous

memory blocks to processing resources. Further, it includes a simple execution model using these patterns to execute kernels on *entities*, processing elements supported by various back-ends. We combine two existing projects as MEPHISTO to achieve flexible kernel acceleration and offloading in distributed systems with partitioned global memory.

6.2 Limitations

The approach, aside from the restrictions outlined in Sect. 2, requires users to be aware of race conditions when working with a shared memory space. Currently there exists no explicit method to synchronize during the execution. However, it is possible to use either synchronization from both the PGAS and the node-level abstraction's implementation.

Further, our prototype implementation currently only implements unified memory for the CUDA back end. This in turn requires CUDA-aware MPI to provide a coherent view of the data to MPI.

6.3 Outlook

In the future we want to focus on automatic load balancing (auto tuning) between entities. For now we only tested the execution on one entity at a time, albeit with multiple instances (threads/GPUs) of each. When the host and multiple entities may execute during a single invocation of an algorithm, the benefits of smart load balancing seem worthwhile investigating to minimize the idle time for each entity.

Another more complex endeavor is the scheduling of kernels on executors independently of the invocation of their containing algorithms. This could bring benefits due to better cache usage and the removal of barriers.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., Thibault, S.: Harnessing clusters of hybrid nodes with a sequential task-based programming model. In: International Workshop on Parallel Matrix Algorithms and Applications (PMAA 2014), Lugano, Switzerland (July 2014)
2. Bell, N., Hoberock, J.: Chapter 26—Thrust: a productivity-oriented library for CUDA. In: Hwu, W., Mei, W. (eds.) GPU Computing Gems Jade Edition, Applications of GPU Computing Series, pp. 359–371. Morgan Kaufmann, Boston (2012)

3. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
4. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. ACM Sigplan Not. **40**(10), 519–538 (2005)
5. Crozier, P., Plimpton, S.: miniMD v. 1.0. Technical report, Sandia National Laboratories (2009)
6. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). (Domain-Specific Languages and High-Level Frameworks for High-Performance Computing)
7. Enmyren, J., Kessler, C.W.: SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications, pp. 5–14 (2010)
8. Fuchs, T., Fürlinger, K.: Expressing and exploiting multi-dimensional locality in DASH. In: Software for Exascale Computing-SPPEXA 2013-2015, pp. 341–359. Springer (2016)
9. Haidl, M., Gorlatch, S.: High-level programming for many-cores using C++14 and the STL. Int. J. Parallel Program. **46**(1), 23–41 (2018)
10. Jose, J., Potluri, S., Subramoni, H., Lu, X., Hamidouche, K., Schulz, K., Sundar, H., Panda, D.K.: Designing scalable out-of-core sorting with hybrid MPI+PGAS programming models. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, pp. 1–9 (2014)
11. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: Hpx: a task based programming model in a global address space. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, pp. 1–11 (2014)
12. Li, M., Lin, J., Lu, X., Hamidouche, K., Tomko, K., Panda, D.K.: Scalable MiniMD design with hybrid MPI and Open SHMEM. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, pp. 1–4 (2014)
13. Nakao, M., Murai, H., Shimosaka, T., Tabuchi, A., Hanawa, T., Kodama, Y., Boku, T., Sato, M.: XcalableACC: extension of XcalableMP PGAS language using OpenACC for accelerator clusters. In: 2014 First Workshop on Accelerator Programming using Directives, pp. 27–36. IEEE (2014)
14. Numrich, R.W., Reid, J.: Co-array Fortran for parallel programming. In: ACM Sigplan Fortran Forum, vol. 17, pp. 1–31. ACM, New York (1998)
15. Ortega-Arranz, H., Torres, Y., Gonzalez-Escribano, A., Llanos, D.R.: TuCCompi: a multi-layer model for distributed heterogeneous computing with tuning capabilities. Int. J. Parallel Program. **43**(5), 939–960 (2015)
16. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. Technical report, Sandia National Labs., Albuquerque, NM (United States) (1993)
17. Steuwer, M., Gorlatch, S.: Skelcl: enhancing opencl for high-level programming of multi-GPU systems. In: International Conference on Parallel Computing Technologies, pp. 258–272. Springer (2013)
18. UPC Consortium et al.: UPC language specifications v1. 2. Technical report, Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US) (2005)
19. Wrede, F., Ernsting, S.: Simultaneous CPU-GPU execution of data parallel algorithmic skeletons. Int. J. Parallel Program. **46**(1), 42–61 (2018)
20. Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W.E., Bussmann, M.: Alpaka-an abstraction library for parallel kernel acceleration. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 631–640. IEEE (2016)