

DASH: Distributed Data Structures and Parallel Algorithms in a Global Address Space



Karl Frlinger, Jos Gracia, Andreas Knpfer, Tobias Fuchs, Denis Hnich, Pascal Jungblut, Roger Kowalewski, and Joseph Schuchart

Abstract DASH is a new programming approach offering distributed data structures and parallel algorithms in the form of a C++ template library. This article describes recent developments in the context of DASH concerning the ability to execute tasks with remote dependencies, the exploitation of dynamic hardware locality, smart data structures, and advanced algorithms. We also present a performance and productivity study where we compare DASH with a set of established parallel programming models.

1 Introduction

DASH is a parallel programming approach that realizes the PGAS (partitioned global address space) model and is implemented as a C++ template library. DASH tries to reconcile the productivity advantages of shared memory parallel programming with the physical realities of distributed memory hardware. To achieve this goal, DASH provides the abstraction of globally accessible memory that spans multiple interconnected nodes. For performance reasons, this global memory is partitioned and data locality is not hidden but explicitly exploitable by the application developer.

DASH is realized as a C++ template library, obviating the need for a custom language and compiler. By relying on modern C++ abstraction and implementation techniques, a productive programming environment can be built solely based on

K. Frlinger (✉) · T. Fuchs · P. Jungblut · R. Kowalewski
Ludwig-Maximilians-Universitt Mnchen, Munich, Germany
e-mail: Karl.Fuerlinger@ifi.lmu.de

J. Gracia · J. Schuchart
University of Stuttgart, Stuttgart, Germany

A. Knpfer · D. Hnich
TU Dresden, Dresden, Germany

standard components. For many application developers in HPC and in general, a big part of the appeal of the C++ programming language stems from the availability of high performance generic data structures and algorithms in the C++ standard template library (STL).

DASH can be seen as a generalization of concepts found in the STL to the distributed memory case and efforts have been made to keep DASH compatible with components of the STL. In many cases it is thus possible to mix and match algorithms and data structures freely between DASH and the STL.

DASH is developed in the context of the SPPEXA priority programme for Exascale computing since 2013. In this paper we give an overview of DASH and report on activities within the project focusing on the second half of the funding period. We first give an overview of the DASH Runtime System (DART) in Sect. 2, focusing on features related to task execution with global dependencies and dynamic hardware topology discovery. In Sect. 3 we describe two components of the DASH C++ template library, a smart data structure that offers support for productive development of stencil codes and an efficient implementation of parallel sorting. In Sect. 4 we provide an evaluation of DASH testing the feasibility of our approach. We provide an outlook on future developments in Sect. 5.

2 The DASH Runtime System

The DASH Runtime System (DART) is implemented in C and provides an abstraction layer on top of distributed computing hardware and one-sided communication substrates. The main functionality provided by DART is memory allocation and addressing as well as communication in a global address space. In DASH parlance the individual participants in an application are called *units* mapped to MPI processes in the MPI-3 remote memory access based implementation of DART.

Early versions of DASH/DART focused on data distribution and access and offered no explicit compute model. This has changed with the support for tasks in DASH and DART. We start with a discussion of these new features, followed by a description of efforts to tackle increasing hardware complexity in Sect. 2.2.

2.1 Tasks with Global Dependencies

The benefit of decoupled transfer and synchronization in the PGAS programming model promises to provide improved scalability and better exploit hardware capabilities. However, proper synchronization of local and global memory accesses is essential for the development of correct applications. So far, the synchronization constructs in DASH were limited to collective synchronization using barriers and reduction operations as well as an implementation of the MCS lock. Using atomic RMA operations, users could also create custom synchronization schemes using

point-to-point signaling, i.e., by setting a flag at the target after completion of a transfer. While this approach might work for simple examples, it hardly scales to more complex examples where reads and writes from multiple processes need to be synchronized.

The need for a more fine-grained way of synchronization that allows to create more complex synchronization patterns was thus imminent. The data-centric programming model of DASH with the distributed data structure at its core lead motivated us to create a synchronization that centers around these global data structures, i.e., which is data-centric itself. At the same time, the essential property of PGAS needed to be preserved: the synchronization had to remain decoupled from data transfers, thus not forcing users to rely solely on the new synchronization mechanism for data transfers.

At the same time, the rise of task-based programming models inspired us to investigate the use of tasks as a synchronization vehicle, i.e., by encapsulating local and global memory accesses into tasks that are synchronized using a data-centric approach. Examples of widely known data-centric task-based synchronization models are OpenMPI with its task data dependencies, OmpSs, and PaRSEC. While PaRSEC uses data dependencies to express both synchronization and actual data flow between tasks, OpenMP and OmpSs use data dependencies solely for synchronization without affecting data movement. In contrast to PaRSEC, however, OpenMP and OmpSs only support shared memory parallelization.

A different approach has been taken by HPX, which facilitates synchronization through the use of future/promise pairs, which form a channel between two or more tasks and are a concept that has been well established in the C++ community. However, this synchronization concept with its inherent communication channel hardly fits into the concept of a PGAS abstraction built around data structures in the global memory space. Moreover, DASH provides a locality-aware programming, in which processes know their location in the global address and can diverge their control accordingly, whereas HPX is a locality-agnostic programming model.

We thus decided to focus our research efforts on distributed data dependencies, extending the shared memory capabilities of task data dependencies into the global memory space while keeping synchronization and data transfer decoupled.

2.1.1 Distributed Data Dependencies

Tasks in DASH are created using the `async` function call and passing it an action that will be executed by a worker thread at a later point in time. Additionally, the `async` function accepts an arbitrary number of dependency definitions of the form `in(memory_location)` and `out(memory_location)` to define input and output dependencies, respectively. In the DASH tasking model, each unit discovers its local task graph by creating tasks operating mainly in the local portion of the global memory space, i.e., tasks are never transferred to other units. This locality-awareness limits the number of tasks to be discovered to only the tasks that will eventually be executed in that unit: as depicted in Fig. 1.

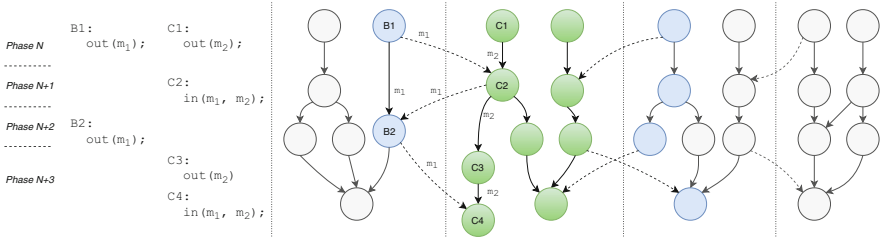


Fig. 1 Distributed task graph discovery: the unit in the middle only discovers its local tasks (green) and should only be concerned with tasks on other units that have dependencies to its local tasks (blue). All other tasks (gray) should not be considered

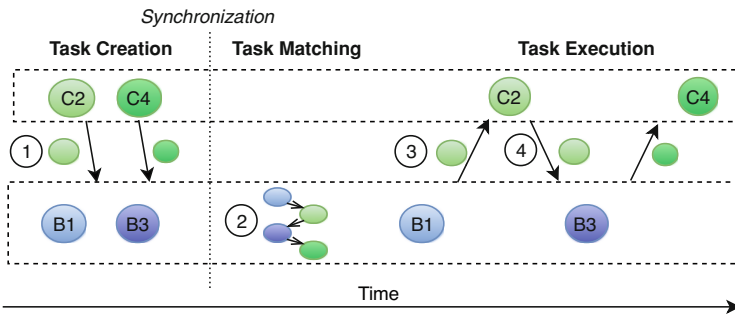


Fig. 2 Scheduler interaction required for handling remote task data dependencies

The discovery of the local task graphs thus happens in parallel and without immediate synchronization between the units, i.e., only the bold edges in Fig. 1 are immediately visible to the individual scheduler instances. In order to connect these trimmed task graphs, the schedulers need to exchange information on dependencies crossing its process boundary, i.e., dependencies referencing non-local global memory locations. The required interaction between the schedulers is depicted in Fig. 2. During the discovery of the trimmed local task graphs, schedulers communicate any encountered dependencies that reference non-local global memory to the unit owning the referenced memory ①. As soon as all dependencies have been communicated, the schedulers extend their local task graphs with the dependency information received from other units ②. A synchronization across all units is required to ensure that all relevant dependency information has been exchanged.

After the extension of the local task graphs, the units start the execution of the tasks. As soon as a task with a dependency to a task on a remote unit, e.g., through an input dependency previously communicated by the remote unit, has completed, the dependency release is communicated to the remote unit ③, where the task will eventually be executed. The completion of the execution is then communicated back to the first scheduler ④ to release any write-after-read dependency, e.g., the dependency $C_2 \rightarrow B_3$ in Fig. 1.

2.1.2 Ordering of Dependencies

As described in Sect. 2.1.1, the local task graphs are discovered by each unit separately. Local edges in the local task graph are discovered similar to the matching rules of OpenMP, i.e., an input dependency refers to the previous output dependency referencing the same memory location (read-after-write), which in turn match with previous input and output dependencies on the same memory location (write-after-read and write-after-write).

However, since the local task graphs are discovered in parallel, the schedulers cannot infer any partial ordering of tasks and dependencies across process boundaries. More specifically, the blue scheduler in Fig. 1 cannot determine the relationship between the dependencies of tasks B_1 , B_2 , C_2 , C_4 . The schedulers thus have to rely on additional information provided by the user in the form of *phases* (as depicted in Fig. 1). A task and its output dependencies are assigned to the current phase upon their discovery. Input dependencies always refer to the last matching output dependency in any previous phase while output dependencies match with any previous local input dependency in the same or earlier phase and any remote input dependency in any earlier phase, up to and including the previous output dependency on the same memory location.

As an example, the input dependency of C_2 is assigned the phase $N + 1$ whereas the input dependency of C_4 is assigned the phase $N + 3$. This information can be used to match the output dependency of B_1 in phase N to the input dependency of C_2 and the output dependency of B_2 in phase $N + 2$ to the input dependency of C_4 , creating the edges $B_1 \rightarrow C_2$ and $B_2 \rightarrow C_4$. The handling of write-after-read dependencies described in Sect. 2.1.1 creates the edge $C_2 \rightarrow B_2$. The handling of local dependencies happens independent of the phase.

In our model, conflicting remote dependencies in the same phase are erroneous as the scheduler is unable to reliably match the dependencies. Two dependencies are conflicting if at least one of them is non-local and at least one is an output dependency. This restriction allows the schedulers to detect synchronization errors such as underdefined phases and report them to the user. This is in contrast to the collective synchronization through barriers traditionally used in DASH, in which synchronization errors cannot be easily detected and often go unnoticed unless the resulting non-deterministic behavior leads to deviations in the application's results.

2.1.3 Implementation

A single task is created using the `async` function in DASH, which accepts both an action to be performed when the task is executed and a set of dependencies that describe the expected inputs and outputs of the task. In the example provided in Listing 1, every call to `async` (lines 5, 11, 19, and 27) is passed a C++ lambda in addition to input and output dependencies.

Instead of pure input dependencies, the example uses `copyin` dependencies, which combine an input dependency with the transfer of the remote memory range

into a local buffer. This allows for both a more precise expression of the algorithms and allows the scheduler to map the transfer onto two-sided MPI communication primitives, which may be beneficial on systems that do not efficiently support MPI RMA. The action performed by the task could still access any global memory location, keeping communication and synchronization decoupled in principle and retaining a one-sided programming model while allowing the use of two-sided communication in the background.

```

1 dash::Matrix<2, double> matrix{N, N, dash::TILE(NB), dash::TILE(NB)};
2
3 for (int k = 0; k < num_blocks; ++k) {
4   if (mat.block(k,k).is_local()) {
5     dash::tasks::async([&]{ potrf(matrix.block(k,k)); },
6     dash::tasks::out(mat.block(k,k)));
7   }
8   dash::tasks::async_fence(); // <- advance to next phase
9   for (int i = k+1; i < num_blocks; ++i)
10    if (mat.block(k,i).is_local())
11      dash::tasks::async([&]{
12        trsm(cache[k], matrix.block(k,i)); },
13      dash::tasks::copyin(mat.block(k,k), cache[k]),
14      dash::tasks::out(mat.block(k,i)));
15   dash::tasks::async_fence(); // <- advance to next phase
16   for (int i = k+1; i < num_blocks; ++i) {
17     for (int j = k+1; j < i; ++j) {
18       if (mat.block(j,i).is_local()) {
19         dash::tasks::async([&]{
20           ↪      gemm(cache[i], cache[j], mat.block(j,i)); },
21         dash::tasks::copyin(mat.block(k,i), cache[i]),
22         dash::tasks::copyin(mat.block(k,j), cache[j]),
23         dash::tasks::out(mat.block(j,i)));
24       }
25     }
26     if (mat.block(i,i).is_local()) {
27       dash::tasks::async([&]{
28         syrk(cache[i], mat.block(i,i)); },
29       dash::tasks::copyin(mat.block(k,i), cache[i]),
30       dash::tasks::out(mat.block(i,i)));
31     }
32   }
33   dash::tasks::async_fence(); // <- advance to next phase
34 }
35 dash::tasks::complete(); // <- wait for all tasks to execute

```

Listing 1 Implementation of Blocked Cholesky Factorization using global task data dependencies in DASH. Some optimizations omitted for clarity

The specification of phases is done through calls to the `async_fence` function (lines 8, 15, and 33 in Listing 1). Similar to a barrier, it is the user’s responsibility to ensure that all units advance phases in lock-step. However, the phase transition triggered by `async_fence` does not incur any communication. Instead, the call causes an increment of the phase counter, whose new value will be assigned to all ensuing tasks.

Eventually, the application waits for the completion of the execution of the global task graph in the call to `complete()`. Due to the required internal synchronization and the matching of remote task dependencies, the execution of all but the tasks in

the first phase has to be post-poned until all its dependencies in the global task-graph are known. DASH, however, provides the option to trigger intermediate matching steps triggered by a phase increment and allows the specification of an upper bound on the number of active phases¹ to avoid the need for discovering the full local task graph before execution starts. This way the worker threads executing threads may be kept busy while the main thread continues discovering the next window in the task graph.

In addition to the single task creation construct described above, DASH also provides the `taskloop()` construct, for which an example is provided in Listing 2. The `taskloop` function divides the iteration space `[begin, end)` into chunks that are assigned to tasks, which perform the provided action on the assigned subrange (lines 7–9). The user may control the size of each chunk (or the overall number of chunks to be created) by passing an instance of `chunk_size` (or `num_chunks`) to the call (Line 5). In addition, the call accepts a second lambda that is used to specify the dependencies of each task assigned a chunk (lines 11–14), which allows a depth-first scheduler to chain the execution of chunks of multiple data-dependent loops, effectively improving cache locality without changing the structure of the loops.

```

1 dash::Array<int> arr(N);
2
3 if (dash::myid() == 0) {
4     dash::tasks::taskloop(
5         arr.begin(), arr.end(), dash::tasks::chunk_size(10),
6         // task action
7         [&] (auto begin, auto end) {
8             // perform action on elements in [begin, end)
9         },
10        // generate out dependencies on elements in [begin, end)
11        [&] (auto begin, auto end, auto deps) {
12            for (auto it = begin; it != end; ++it)
13                *deps = dash::tasks::out(it);
14        });
15 }

```

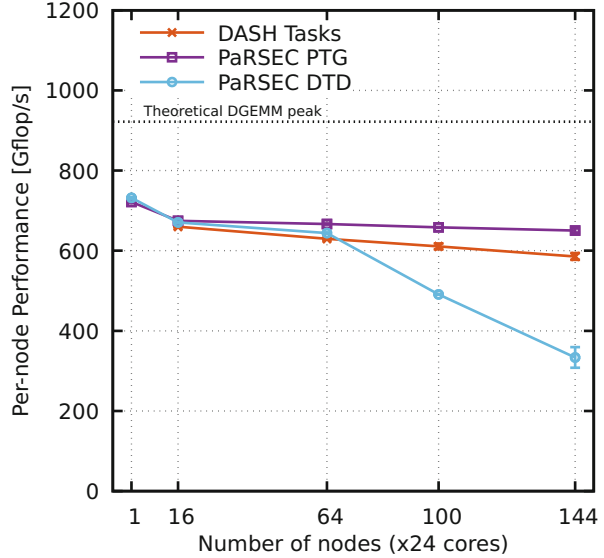
Listing 2 Example of using the `dash::taskloop` in combination with a dependency generator

2.1.4 Results: Blocked Cholesky Factorization

The implementation of the Blocked Cholesky Factorization discussed in Sect. 2.1.3 has been compared against two implementations in PaRSEC. The first implementation uses the parameterized task graph (PTG), in which the problem is described as an directed acyclic graph in a domain-specific language called JDF. In this version,

¹A phase is considered active while a task discovered in that phase has not completed execution.

Fig. 3 Per-node weak-scaling performance of Blocked Cholesky Factorization of a matrix with leading dimension $N = 25k/node$ and block size $NB = 320$ on a Cray XC40 (higher is better)



the task graph is not dynamically discovered but is instead inherently contained within the resulting binary.

The second PaRSEC version uses the Dynamic Task Discovery (DTD) interface of PaRSEC, in which problems are expressed with a global view, i.e., all processes discover the global task graph to discover the dependencies to tasks executing on remote processes.

For all runs, a background communication thread has been employed, each time running on a dedicated core, leading to one main thread and 22 worker threads executing the application tasks on the Cray XC40.

The results presented in Fig. 3 indicate that PaRSEC PTG outperforms both DTD and DASH, due to the missing discovery of tasks and their dependencies. DTD exhibits a drop in per-node performance above 64 nodes, which may be explained with the global task graph discovery. Although the per-node performance of DASH does not exhibit perfect scaling, it still achieves about 80% of the performance of PaRSEC PTG at 144 nodes.

2.1.5 Related Work

HPX [18] is an implementation of the ParalleX [17] programming paradigm, in which tasks are spawned dynamically and moved to the data, instead of the data being moved to where the task is being executed. HPX is locality-agnostic in that distributed parallelism capabilities are implicit in the programming model, rather than explicitly exposed to the user. An Active Global Address Space (AGAS) is used to transparently manage the locality of global objects. Synchronization of tasks

is expressed using futures and continuations, which are also used to exchange data between tasks.

In the Active Partitioned Global Address Space (APGAS) [27], in contrast, the locality of so-called places is explicitly exposed to the user, who is responsible for selecting the place at which a task is to be executed. Implementations of the APGAS model can be found in the X10 [9] and Chapel [8] languages as well as part of UPC and UPC++ [21].

The Charm++ programming system encapsulates computation in objects that can communicate using message objects and can be migrated between localities to achieve load balancing.

Several approaches have been proposed to facilitate dynamic synchronization by waiting for events to occur. AsyncShmem is an extension of the OpenShmem standard, which allows dynamic synchronization of tasks across process boundaries by blocking tasks waiting for a state change in the global address space [14]. The concept of phasers has been introduced into the X10 language to implement non-blocking barrier-like synchronization, with the distinction of readers and writers contributing to the phaser [29].

Tasklets have recently been introduced to the XcalableMP programming model [35]. The synchronization is modeled to resemble message-based communication, using data dependencies for tasks on the same location and notify-wait with explicitly specified target and tags.

Regent is a region- and task-based programming language that is compiled into C++ code using the Legion programming model to automatically partition the computation into logical regions [4, 30].

The PaRSEC programming system uses a domain specific language called JDF to express computational problems in the form of a parameterized task graph (PTG) [7]. The PTG is implicitly contained in the application and not discovered dynamically at runtime. In contrast to that, the dynamic task discovery (DTD) frontend of PaRSEC dynamically discovers the global task-graph, i.e., each process is aware of all nodes and edges in the graph.

A similar approach is taken by the sequential task flow (STF) frontend of StarPU, which complements the explicit MPI send/recv tasks to encapsulate communication in tasks and implicitly express dependencies across process boundaries [1].

Several task-based parallelization models have been proposed for shared memory concurrency, including OpenMP [3, 24], Intel thread building blocks (TBB) [25] and Cilk++ [26] as well as SuperGlue [34]. With ClusterSs, an approach has been made to introduce the APGAS model into OmpSs [32].

2.2 *Dynamic Hardware Topology*

Portable applications for heterogeneous hosts adapt communication schemes and virtual process topologies depending on system components and the algorithm scenario. This involves concepts of vertical and horizontal locality that are not based

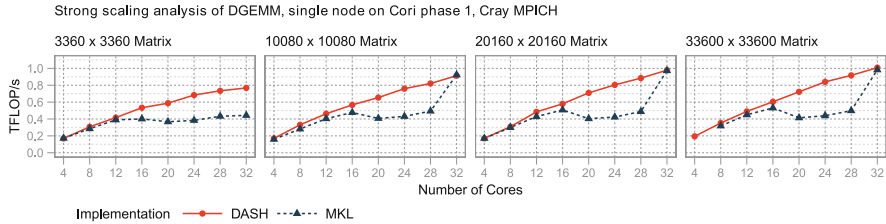


Fig. 4 Strong scaling of matrix multiplication on single node for 4 to 32 cores with increasing matrix size $N \times N$ on Cori phase 1, Cray MPICH

on latency and throughput as distance measure. For example in a typical accelerator-offloading scenario, data distribution to processes optimizes for horizontal locality to reduce communication distance between collaborating tasks. For communication in the reduction phase, distance is measured based on vertical locality.

The goal to provide a domain-agnostic replacement for the C++ Standard Template Library (STL) implies portability as a crucial criterion for every model and implementation of the DASH library. This includes additional programming abstractions provided in DASH, such as n-dimensional containers which are commonly used in HPC. These are not part of the C++ standard specifications but comply with its concepts. Achieving portable efficiency of PGAS algorithms and containers that satisfy semantics of their conventional counterparts is a multivariate, hard problem, even for the seemingly most simple use cases.

Performance evaluation of the of the DASH NArray and dense matrix-matrix multiplication abstractions on different system configurations substantiated the portable efficiency of DASH. The comparison also revealed drastic performance variance of the established solutions, for example node-local DGEMM of Intel MKL on Cori phase 1 shown in Fig. 4 which apparently expected a power of two amount of processing cores for multi-threaded scenarios.

The DASH variant of DGEMM internally uses the identical Intel MKL distribution for multiplication of partitioned matrix blocks but still achieves robust scaling. This is because DASH implements a custom, adaptive variant of the SUMMA algorithm for matrix-matrix multiplication and assigns one process per core, each using MKL in sequential mode. This finding motivated to find abstractions that allow expressions for domain decomposition and process placement depending on machine component topology. In this case: to group processes by NUMA domains with one process per physical core.

2.2.1 Locality-Aware Virtual Process Topology

In the DASH execution model, individual computation entities are called *units*. In the MPI-based implementation of the DASH runtime, a unit corresponds to an MPI rank but may occupy a locality domain containing several CPU cores or, in principle, multiple compute nodes.

Units are organized in hierarchical *teams* to match the logical structure of algorithms and machine components. Each unit is an immediate member of exactly one team at any time, initially in the predefined team ALL. Units in a team can be partitioned into child teams using the team’s `split` operation which also supports locality-aware parameters.

On systems with asymmetric or deep memory hierarchies, it is highly desirable to split a team such that locality of units within every child team is optimized. A locality-aware split at node level could group units by affinity to the same NUMA domain, for example. For this, locality discovery has been added to the DASH runtime. Local hardware information from `hwloc`, `PAPI`, `libnuma`, and `LIKWID` of all nodes is collected into a global, uniform data structure that allows to query locality information by process ID or scope in the memory hierarchy.

This query interface proved to be useful for static load balancing on heterogeneous systems where team are split depending on the machine component capacities and capabilities. These are stored in a hierarchy of domains with two *property maps*:

Capabilities invariant hardware locality properties that do not depend on the locality graph’s structure, like the number of threads per core, cache sizes, or SIMD width

Capacities derivative properties that might become invalid when the graph structure is modified, like memory in a NUMA domain available per unit

Figure 5 outlines the data structure and its concept of hardware abstraction in a simplified example of a topology-aware split. Domain capacities are accumulated from its subdomains and recalculated on restructuring. Team 1 and 2 both contain twelve cores but a different number of units. A specific unit’s maximum number of threads is determined by the number of cores assigned to the unit and the number of threads per core.

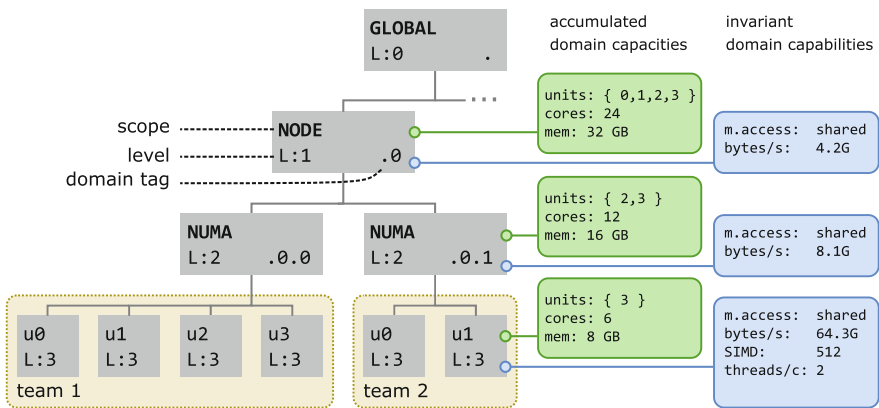


Fig. 5 Domains in a locality hierarchy with domain attributes in dynamically accumulated capacities and invariant capabilities

2.2.2 Locality Domain Graph

The machine component topology of the DASH runtime to support queries and topology-aware restructuring extends the tree-based hwloc topology model to represent properties and relations of machine components in a graph structure. It evolved to the *Locality Domain Graph* (LDG) concept which is available as the standalone library *dyloc*.²

In formal terms, a locality domain graph models hardware topology as directed, multi-indexed multigraph. In this, nodes represent *Locality Domains* that refer to any physical or logical component of a distributed system with memory and computation capabilities, corresponding to *places* in X10 or Chapel's *locales* [8]. Edges in the graph are directed and denote the following relationships, for example:

- **Containment** indicating that the target domain is logically or physically contained in the source domain
- **Alias** source and target domains are only logically separated and refer to the same physical domain; this is relevant when searching for a shortest path, for example
- **Leader** the source domain is restricted to communication with the target domain

2.2.3 Dynamic Hardware Locality

Dynamic locality support requires means to specify transformations on the physical topology graph as *views*. Views realize a projection but must not actually modify the original graph data. Invariant properties are therefore stored separately and assigned to domains by reference only.

Conceptually, multi-index graph algebra can express any operation on a locality domain graph, but complex to formulate. When a topology is projected to an acyclic hierarchy, transformations like partitioning, selection and grouping of domains can be expressed in conventional relational or set semantics. A partition or contraction of a topology graph can be projected to a tree data structure and converted to a hwloc topology object (Fig. 6).

A locality domain topology is specific to a team and only contains domains that are populated by the team's units. At initialization, the runtime initializes the default team ALL as root of the team hierarchy with all units and associates the team with the global locality graph containing all domains of the machine topology. When a team is split, its locality graph is partitioned among child teams such that a single partition is coherent and only contains domains with at least one leaf occupied by a unit in the child team.

In a map-reduce scenario, dynamic views on machine topology to express for domain decomposition and process placement depending on machine component

²<https://github.com/dash-project/dyloc>.

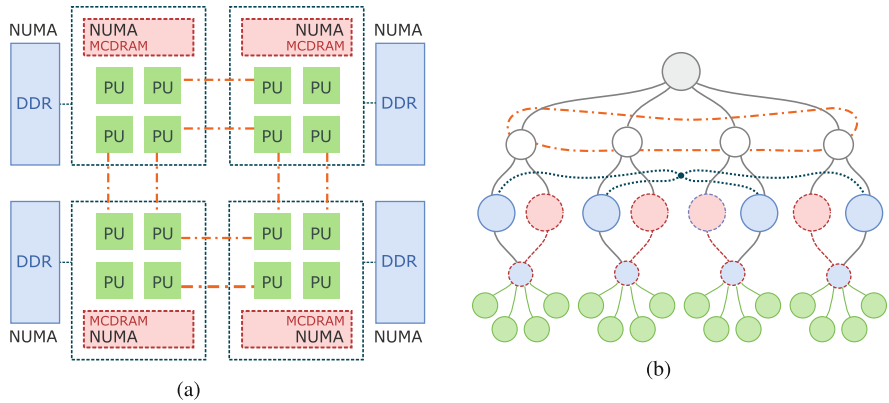


Fig. 6 Illustration of a hardware locality domain graph as a model of node-level system architectures that cannot be correctly or unambiguously represented in a single tree structure. **(a)** Cluster in Intel Knights Landing, configured in Sub-NUMA clustering, Hybrid mode. Contains a quarter of the processor’s cores, MCDRAM local memory, affine to DDR NUMA domain. **(b)** Exemplary graph representation of Knights Landing topology in **(a)**. Vertex categories model different aspects of component relationships, like cache-coherence and adjacency

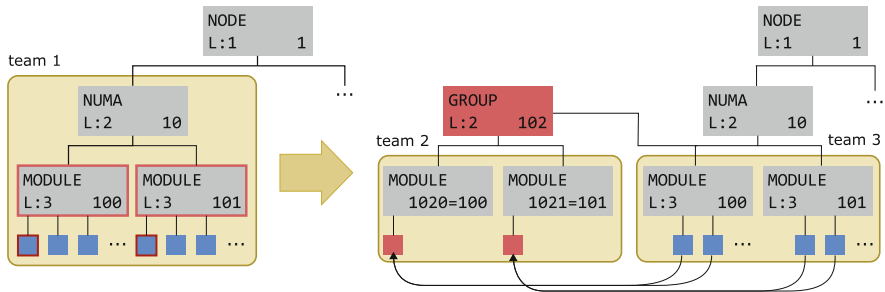


Fig. 7 Illustration of the domain grouping algorithm to define a leader group for vertical communication. One core is selected as leader in domains 100 and 110 and separated into a group. To preserve the original topology structure, the group includes their parent domains and is added as a subdomain of their lowest common ancestor

topology and improve portable efficiency. In the map phase, the algorithm is mostly concerned with *horizontal locality* in domain decomposition to distribute data according to the physical arrangement of cooperating processes. In the reduce phase, *vertical locality* of processes in the component topology determines efficient upwards communication of partial results. The locality domain graph can be used to project hardware topology to tree views for both cases. Figure 7 illustrates a locality-aware split of units in two modules such that one unit per module is selected for upwards communication. This principle is known as *leader communication scheme*. Partial results of units are then first reduced at the unit in the respective leader team. This drastically reduces communication overhead as the physical bus between the

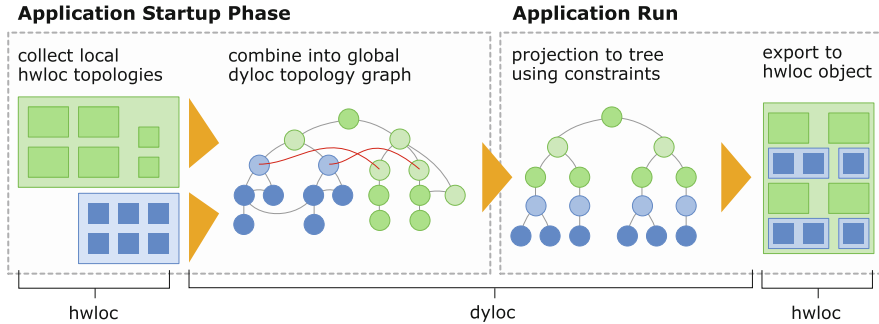


Fig. 8 Using dyloc as intermediate process in locality discovery

modules and their NUMA node is only shared by two leader processes instead of all processes in the modules (Fig. 8).

2.2.4 Supporting Portable Efficiency

As an example of both increased depth of the machine hierarchy and heterogeneous node-level architecture, the SuperMIC system³ consists of 32 compute nodes with symmetric hardware configuration of two NUMA domains, each containing an Ivy Bridge (8 cores) host processor and a Xeon Phi “Knights Corner” coprocessors (Intel MIC 5110P) as illustrated in Fig. 9.

For portable work load balancing on heterogeneous systems, domain decomposition and virtual process topology must dynamically adapt the machine components’ inter-connectivity, capacities and capabilities.

Capacities: Total memory capacity on MIC modules is 8 GB for 60 cores, significantly less than 64 GB for 32 cores on host level

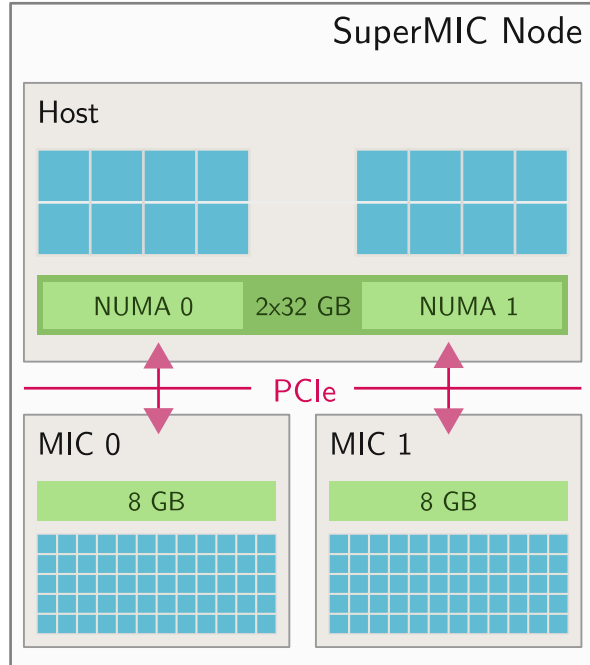
Capabilities: MIC cores have a base clock frequency of 1.1 GHz and 4 SMT threads, with 2.8 GHz and 2 SMT threads on host level

To illustrate the benefit of dynamic locality, we briefly discuss the implementation of the `min_element` algorithm in DASH. Its original variant is implemented as follows: domain decomposition divides the element range into contiguous blocks of identical size. All units then run a thread-parallel scan on their local block for a local minimum and enter a collective barrier once it has been found. Once all units finished their local work load, local results are reduced to the global minimum.

Listing 3 contains the abbreviated implementation of the `min_element` scenario utilizing runtime support based on a dynamic hardware locality graph.

³<https://www.lrz.de/services/compute/supermuc/supermic>.

Fig. 9 SuperMIC node



Dynamic topology queries are utilized in three essential ways to improve overall load-balance: In domain decomposition (lines 3–6), to determine the number of threads available to the respective unit (line 19) and for a simple leader-based communication scheme (lines 8–10, 26).

This implementation achieves portable efficiency across systems with different memory hierarchies and hardware component properties, and dynamically adapts to runtime-specific team size, range size, and available hardware components assigned to the team. Figure 10 shows timeline plots comparing time to completion and process idle time from a benchmark run executed on SuperMIC.

```

1 // Dynamic topology-aware domain decomposition depending on
2 // machine component properties and number of units in team:
3 TeamLocality      tloc(dash::Team::All());
4 LocBalancedPattern pattern(array_size, tloc);
5 dash::Array<T>    array(pattern);
6
7 GlobIt min_element(GlobIt first, GlobIt last) {
8     auto uloc      = UnitLocality(myid());
9     auto leader    = uloc.at_scope(scope::MODULE)
10                      .unit_ids()[0];
11     auto loc_min   = first;
12
13     // Allocate shared variable for reduction result at leader:
14     dash::Shared<GlobIt> glob_min(leader);
15     // Allocate shared array for local minimum values:

```

```

16  dash::Array<GlobIt>(dash::Team::All().size()) loc_mins;
17
18  // Dynamic query of locality runtime for number of threads:
19  auto nthreads = uloc.num_threads();
20  #pragma omp parallel for num_threads(nthreads)
21  for (...) { /* ... find local result ... */ }
22  // Local write, no communication
23  loc_mins[my_id] = loc_min;
24  dash::barrier();
25
26  if (myid() == leader) {
27      // leader reduces local results (instead of all-to-all
28      // reduction)
29      glob_min = std::min_element(loc_mins.begin(),
30                                loc_mins.end());
31
32  }
33  // ...
34  }

```

Listing 3 Code excerpt of the modified min_element algorithm

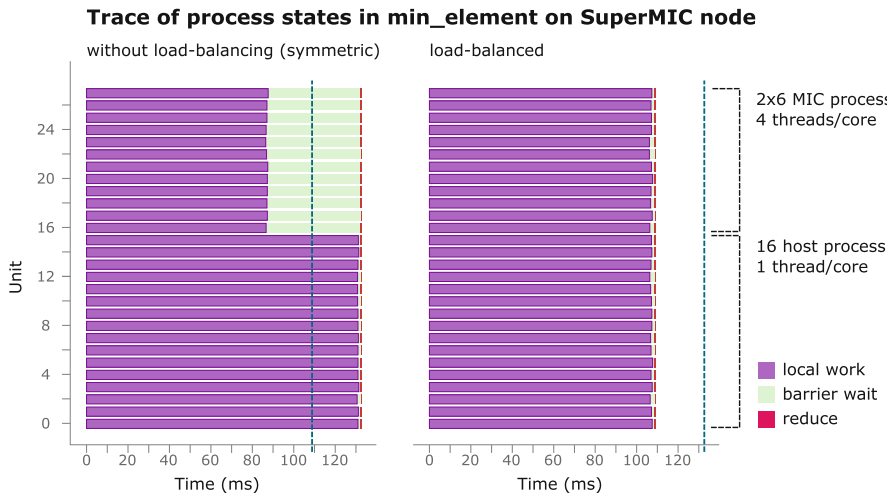


Fig. 10 Trace of process activities in the min_element algorithm exposing the effect of load balancing based on dynamic hardware locality

3 DASH C++ Data Structures and Algorithms

The core of DASH is formed by data structures and algorithms implemented as C++ templates. These components are conceptually modeled after their equivalents in the C++ standard template library, shortening learning curves and increasing programmer productivity. A basic DASH data structure is the distributed array with configurable data distribution (`dash::Array`) which closely follows the functionality of a STL `vector` except for the lack of runtime resizing. DASH also offers a multidimensional array and supports a rich variety of data distribution patterns [11]. A focus of the second half of the funding period was placed on *smart* data structures which are more specialized and support users in the development of certain types of applications. One such data structures for the development of stencil-based applications is described in Sect. 3.1.

Similar to data structures, DASH also offers generalized parallel algorithms. Many of the over 100 generic algorithms contained in the STL have an equivalent in DASH (e.g., `dash::fill`). One of the most useful but also challenging algorithms is sorting and Sect. 3.2 describes our implementation of scalable distributed sorting in the DASH library.

3.1 *Smart Data Structures: Halo*

Typical data structure used in ODE/PDE solvers or 2D/3D image analyzers are multi-dimensional arrays. The DASH `NArray` distributes data elements of a structured data `grid` and can be used similar to STL containers. But PDE solvers use stencil operations, not using the current data elements (center), but surrounding data elements (neighbors) as well. The use of the `NArray` itself is highly inefficient with stencil operations, because neighbors located in another sub-arrays may require remote access (via RDMA or otherwise). A more efficient approach is the use of so called “halo areas”. These areas contain copies of all required neighbor elements located on other compute nodes. The halo area width depends on the shape of the stencils and is determined by the largest distance from the center (per dimension). The stencil shape defines all participating data elements—center and neighbors. Figure 11 shows two 9-point stencils with different shapes. The first stencil shape (a) accesses ± 2 data elements in both horizontal and vertical direction and the second one (b) accesses ± 1 stencil point in each direction. While the stencil shape Fig. 11a needs four halo areas with a width of two data elements. The other stencil shape requires eight halo areas with a width one data elements. Using halo areas ensures local data access for all stencil operations used on each sub-array.

The `Dash NArray Halo Wrapper` wraps the local part of the `NArray` and automatically sets up a halo environment for stencil codes and halo accesses. Figure 12 shows an overview about all main components, which are explained in the following.

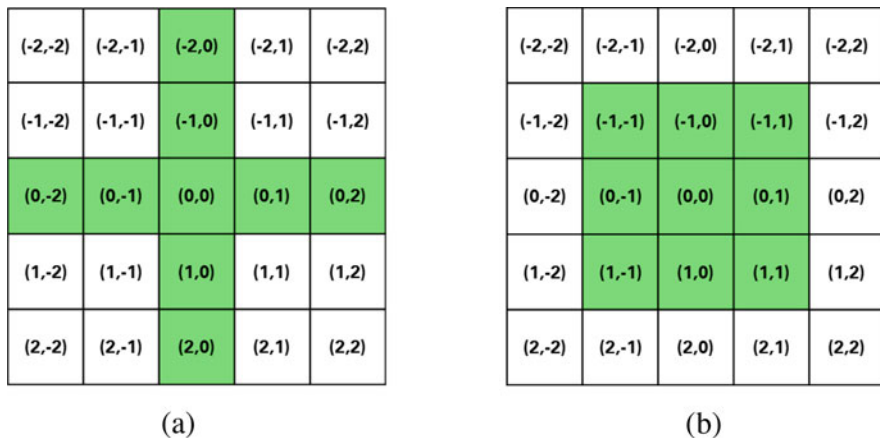


Fig. 11 Two shapes of a 9-point stencil. (a) ± 2 center stencil in horizontal and vertical directions. (b) Center ± 1 stencil point in each direction

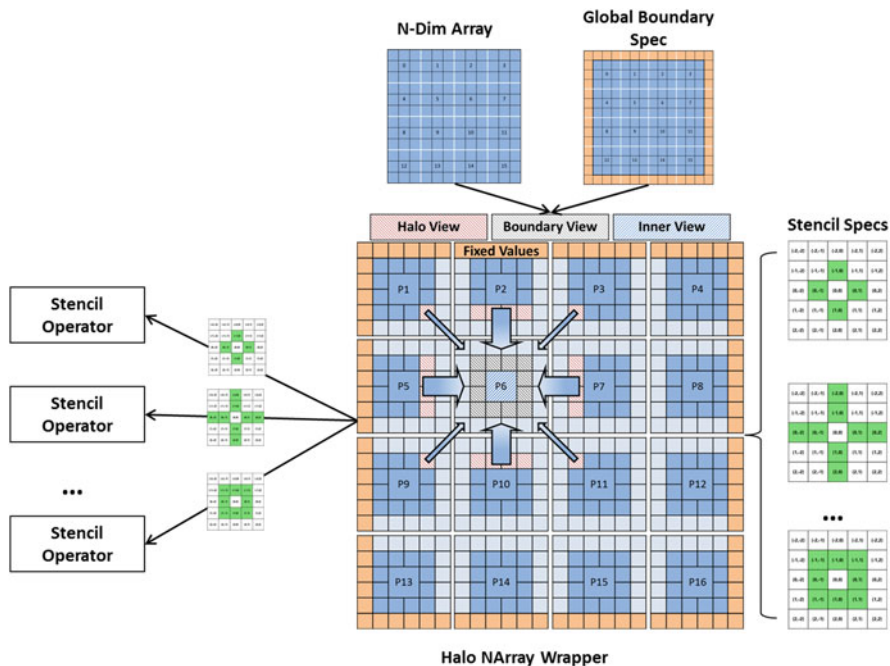


Fig. 12 Architecture of the DASH Halo NArray Wrapper

3.1.1 Stencil Specification

The discretization of the problem to be solved always determines the stencil shape to be applied on the structured grid. For a DASH based code this has to be specified as a stencil specification (StencilSpec), which is a collection of stencil points (StencilPoint). A StencilPoint consists of coordinates relative to the center and an optional weight (coefficient). The StencilSpec specified in Listing 4 describes the stencil shown in Fig. 11b. The center doesn't have to be declared directly.

```

1 using PointT = dash::halo::StencilPoint<2>;
2 dash::halo::StencilSpec<PointT,6> stencil_spec(
3     PointT(-1,-1), PointT(-1, 0), PointT(-1,1),
4     PointT( 0,-1),           , PointT( 0,1),
5     PointT( 1,-1), PointT( 1, 0), PointT( 1,1));

```

Listing 4 Stencil specification for an 9-point stencil

3.1.2 Region and Halo Specifications

The region specification (RegionSpec) defines the location of all neighboring partitions. Every unit keeps 3ⁿ regions representing neighbor partitions for “left”, “middle”, and “right” in each of the n dimensions. All regions are identified by a region index and a corresponding region coordinate. Indexing are done with the Row Major linearization (last index grows fastest). Figure 13 shows all possible regions with its indexes and coordinates for a two dimensional scenario. Region 4 with the coordinates (1,1) is mapped to the center region and represents the local partition. Region 6 (2,0) points to a remote partition located in the south west.

Fig. 13 Mapping of region coordinates and indexes

<p>0 (0,0) NW</p>	<p>1 (0,1) N</p>	<p>2 (0,2) NE</p>
<p>3 (1,0) W</p>	<p>4 (1,1) Local View</p>	<p>5 (1,2) E</p>
<p>6 (2,0) SW</p>	<p>7 (2,1) S</p>	<p>8 (2,2) SE</p>

The halo specification (HaloSpec) uses the RegionSpec to map neighbor partitions which are the origins for halo copies to the local halo areas. From one or multiple StencilSpecs it infers which neighbor partitions are necessary. In case no StencilPoint has a negative offset from the center in horizontal direction, no halo regions for the ‘NW’, ‘W’, and ‘SW’ (Fig. 13) need to be created. If no StencilPoint has diagonal offsets (i.e. only one non-zero coordinate in the offsets) the diagonal regions ‘NW’, ‘NE’, ‘SW’, and ‘SE’ can be omitted.

3.1.3 Global Boundary Specification

Additionally, the global boundary specification (GlobalBoundarySpec) allows to control the behavior at the outside of the global grid. For convenience, three different scenarios are supported. The default setting is `NONE` meaning that there are no halo areas in this direction. Therefore, the stencil operations are not applied in the respective boundary region where the stencil would require the halo to be present. As an alternative, the setting `CYCLIC` can be set. This will wrap around the simulation grid, so that logically the minimum coordinate becomes a neighbor to the maximum coordinate. Furthermore, the setting `CUSTOM` creates a halo area but never performs automatic update of its elements from any neighbors. Instead, this special halo area can be written by the simulation (initially only or updated regularly). This offers a convenient way to provide boundary conditions to a PDE solver. The GlobalBoundarySpec can be defined separately per dimension.

3.1.4 Halo Wrapper

Finally, using the aforementioned specifications as inputs, the halo wrapper (HaloWrapper) creates HaloBlocks for all local sub-arrays. The halo area extension is derived from all declared StencilSpecs by determining the maximum offset of any StencilPoint in the given direction.

The mapping between the local HaloBlocks and the halo regions pointing to the remote neighbor data elements is subjected to the HaloWrapper, as well as the orchestration of efficient data transfers for halo data element updates. The data transfer has to be done block-wise instead of element-wise to gain decent performance. While the HaloBlock can access contiguous memory, the part of the neighbor partition marked as halo area, usually can’t be accessed contiguously—compare Fig. 14. Therefore, the HaloWrapper relies on DART’s support for efficient strided data transfers.

The halo data exchange can be done per region or for all regions at once. It can be called asynchronously and operates independent between all processes and doesn’t use process synchronization. The required subsequent wait operation waits for local completion only.

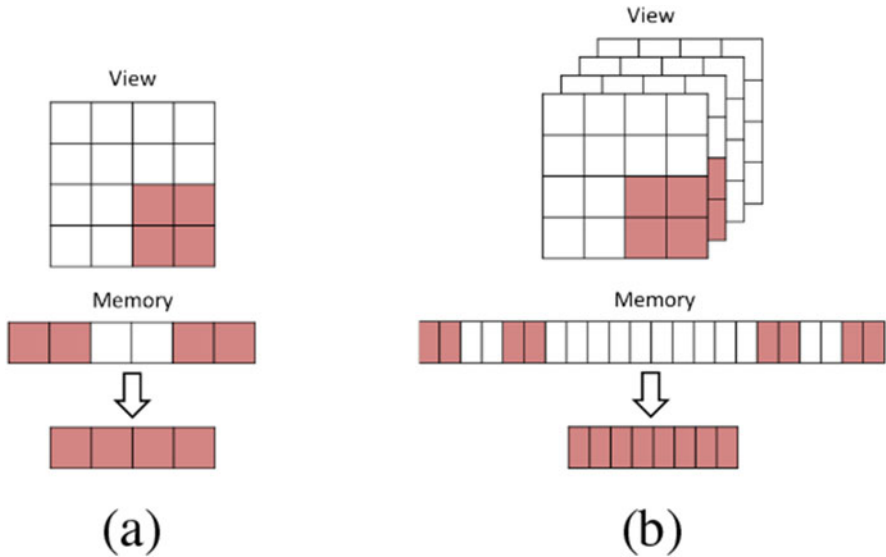


Fig. 14 Halo data exchange of remote strided data to contiguous memory: (a) in 2D a corner halo has a fixed stride, whereas (b) in 3D the corner halo region has two different strides

3.1.5 Stencil Operator and Stencil Iterator

So far, the HaloWrapper was used to create HaloBlocks to fit all given StencilSpecs. Besides that, the HaloWrapper also provides specific views and operations for each StencilSpec.

First, for every StencilSpec the HaloWrapper provides a StencilOperator with adapted *inner* and *boundary* views. The inner view contains all data elements that don't need a halo area when using the given stencil operation. All other data elements are marked via the boundary view. These two kind of views are necessary to overlap the halo data transfer with the inner computation. The individual view per StencilSpec allows to make the inner view as large as possible, regardless of other StencilSpecs.

Second, the HaloWrapper offers StencilSpec specific StencilIterators. They iterate over all elements assigned by a given view (inner) or a set of views (boundary). With these iterators center elements can be accessed—equivalent to STL iterators—via the dereference operator. Neighboring data elements can be accessed with a provided method. Stencil points pointing to elements within a halo area, are resolved automatically without conditionals in the surrounding code. StencilIterators can be used with random access, but are optimized for the increment operator.

3.1.6 Performance Comparison

A code abstraction hiding complexity is useful only, if no or minor performance impact is added. Therefore, a plain MPI implementation of a heat equation was compared to a DASH based one regarding weak and strong scaling behavior. All measurements were performed on the Bull HPC-Cluster ‘‘Taurus’’ at ZIH, TU Dresden. Each compute node has two Haswell E5-2680 v3 CPUs at 2.50 GHz with 12 physical cores each and 64 GB memory. Both implementations were built with gcc 7.1.0 and OpenMPI 3.0.0.

The weak scaling scenario increases the number of grid elements proportional to the number of compute nodes. The accumulated main memory is almost entirely used up by each compute grid. Figure 15 shows that both implementations almost have identical and perfect weak scaling behavior. Note that the accumulated waiting times differ significantly. This is due to two effects. One is contiguous halo areas (north and south) vs. strided halo areas (east and west). The other is intra node vs. inter node communication.

The strong scaling scenario uses $55,000^2$ grid elements to fit into the main memory of a single compute node. It is solved with 1 to 768 *CPU cores* (== MPI ranks), where 24 cores equals to one full compute node and 768 cores to 32 compute nodes. Figure 16 shows again an almost identical performance behavior between DASH and MPI for the total runtime. Notably, both show the same performance artifact around 16 to 24 cores. This can be ascribed to an increased number of last level cache load misses which indicates that both implementations are memory bound at this number of cores per node.

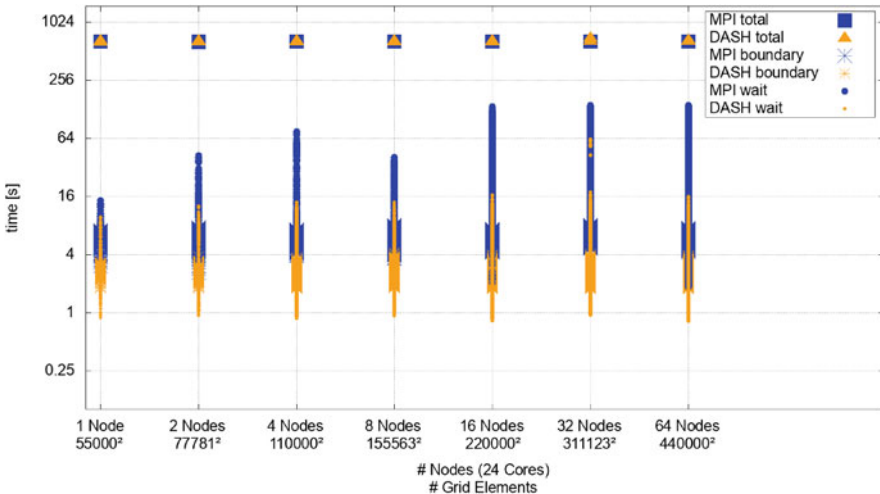


Fig. 15 Weak scaling in DASH vs. MPI

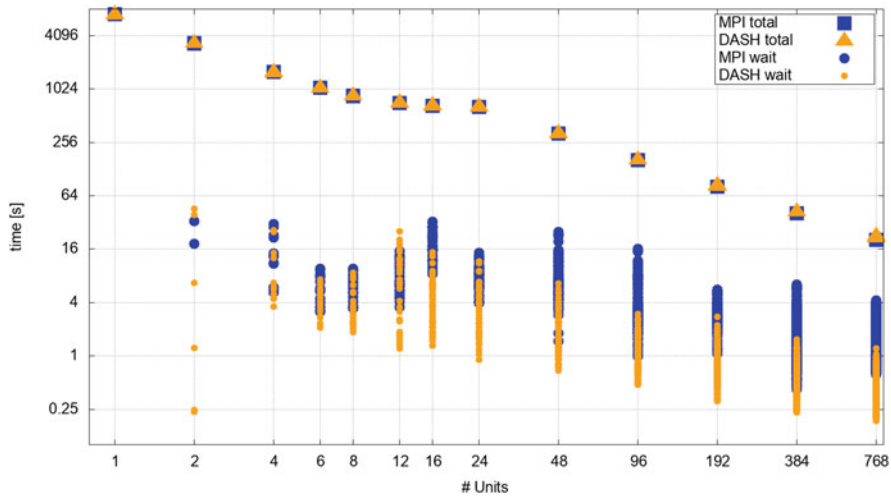


Fig. 16 Strong scaling for $55,000 \times 55,000$ elements in DASH vs. MPI. Effective wait time for asynchronous halo exchanges is shown in addition

3.2 Parallel Algorithms: Sort

Sorting is one of the most important and well studied non-numerical algorithms in computer science and serves as a basic building block in a wide spectrum of applications. A notable example in the scientific domain are N-Body particle simulations which are inherently communication bound due to load imbalance. Common strategies to mitigate this problem include redistributing particles according to a space filling curve (e.g., Morton Order) which can be achieved with sorting. Other interesting use cases which can be addressed using DASH are Big Data applications, e.g., Google PageRank.

Key to achieve performance is obviously to minimize communication. This applies not only to distributed memory machines but to shared memory architectures as well. Current supercomputers facilitate nodes with large memory hierarchies organized in a growing number of NUMA domains. Depending on the data distribution, sorting is subject to a high fraction of data movement and the more we communicate across NUMA boundaries the more negative the result performance impact becomes.

In the remainder of this section we briefly describe the problem of sorting in a more formal manner and summarize the basic approaches in related work. It follows a more detailed elaboration of our sorting algorithm [20]. Case studies on both distributed and shared memory demonstrate our performance efficiency. Results reveal that we can outperform state of the art implementations with our PGAS algorithm.

3.2.1 Preliminaries

Let X be a set of N keys evenly partitioned among P processors, thus, each processor contributes $n_i \sim N/P$ keys. We further assume there are no duplicate keys which can technically be achieved in a straightforward manner. Sorting permutes all keys by a predicate which is a binary relation in set X . Recursively applying this predicate to any ordered pair (x, y) drawn from X enables to determine the rank of an element $I(x) = k$ with x as the k -th order statistic in X . Assuming our predicate is *less than* (i.e., $<$) the output invariant after sorting guarantees that for any two subsequent elements $x, y \in X$

$$x < y \Leftrightarrow I(x) < I(y).$$

Scientific applications usually require a balanced load to maximize performance. Given a load balance threshold ϵ , *local balancing* means that in the sorted sequence each processor P_i owns at most $N(1 + \epsilon)/P$ keys. This does not always result in a *globally balanced* load which is an even stronger guarantee.

Definition 1 For all $i \in \{1..P\}$ we have to determine splitter S_i to partition the input sequence into P subsequences such that

$$\frac{Ni}{P} - \frac{N\epsilon}{2P} \leq I(s_i) \leq \frac{Ni}{P} + \frac{N\epsilon}{2P}$$

Determining these splitters boils down to the *k-way selection* problem which is a core algorithm in this work. If $\epsilon = 0$ we need to perfectly partition the input which increases communication complexity. However, it often is the easiest solution in terms of programming productivity which is a major goal of the DASH library.

3.2.2 Related Work

Sorting large inputs can be achieved through parallel sample sort which is a generalization of *Quicksort* with multiple pivots [5]. Each processor partitions local elements into p pieces which are obtained out of a sufficiently large sample of the input. Then, all processors exchange elements among each other such that piece i is copied to processor i . In a final step, all processors sort received pieces locally, resulting in a globally sorted sequence. *Perfect partitioning* can be difficult to achieve as splitter selection is based only on a sample of the input.

In parallel p -way mergesort each processor first sorts the local data portion and subsequently partitions it, similar to sample sort, into p pieces. Using an ALL-TO-ALL exchange all pieces are copied to the destination processors which finally merge them to obtain a globally sorted sequence. Although this algorithm has worse ioefficiency due to the partitioning overhead compared to sample sort, *perfect partitioning* becomes feasible since data is locally sorted.

Scalable sorting algorithms are compromises between these two extremes and apply various strategies to mitigate negative performance impacts of splitter selection (partitioning) and ALL-TO-ALL communication [2, 15, 16, 31]. Instead of communicating data only once, partitioning is done recursively from a coarse-grained to a more fine-grained solution. Each recursion leads to independent subpartitions until the solution is found. Ideally, the level of recursion maps to the underlying hardware resources and network topology.

This work presents two contributions. First, we generalize a distributed selection algorithm to achieve scalable partitioning [28]. Second, we address the problem of communication-computation overlap in the ALL-TO-ALL exchange, which is conceptually limited in MPI as the underlying communication substrate.

3.2.3 Histogram Sort

The presented sorting algorithm consists of four supersteps as delineated in Fig. 17.

- Local Sort** Sorts the local portion using a fast shared memory algorithm.
- Splitting** Each processor partitions the local array into p pieces. We generalize distributed selection to a p -way multiselect.
- Data Exchange** Each processor exchanges piece i with processor i according to the splitter boundaries.
- Local Merge** Each processor merges the received sorted pieces.

Splitting is based on distributed selection [28]. Instead of finding one pivot we collect multiple pivots (splitters) in a single iteration, one for each *active* range. If a pivot matches a specific rank we do not consider this range anymore and discard it from the set of active ranges. Otherwise, we examine each of the two resulting

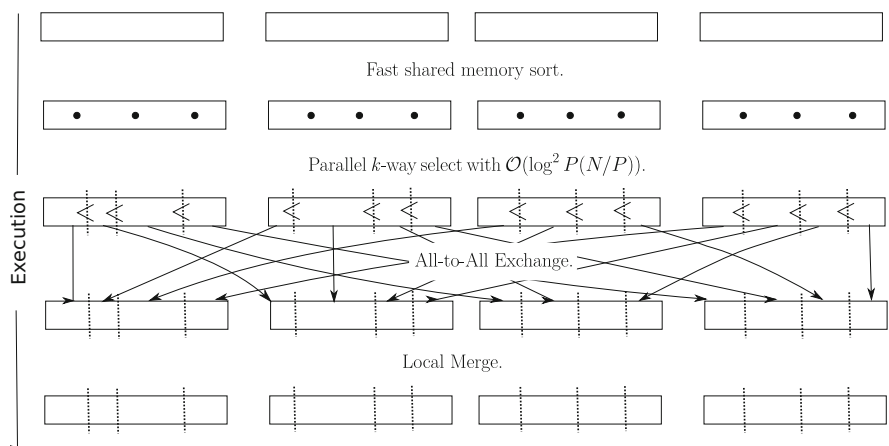


Fig. 17 Algorithmic schema of `dash::sort` with four processors ($P = 4$)

subranges whether they need to be considered in future iterations and add them to the set of active ranges. The vector of splitters follows from Definition 1 (page 126) and is the result of a prefix sum over the local capacities of all processors.

Another difference compared to the original algorithm [28] is that in our case we replace the local partition with binary search which is possible due to our initial sorting step in all processors. Thus, to determine a local histogram over p pieces requires logarithmic computation complexity instead of linear complexity. A global histogram to determine if all splitters are valid (or if we need to refine the boundaries) requires a single REDUCE over all processors with logarithmic communication complexity.

The question how many iterations we need is answered as follows. We know that for the base case with only two processors (i.e., only one splitter) distributed selection has a recursive depth of $O(\log p)$. This follows from the weighted median for the pivot selection which guarantees a reduction of the working set by at least one quarter each iteration. As described earlier instead of a single pivot we collect multiple pivots in a single iteration which we achieve by a list of active ranges. Although the local computation complexity increases by a factor of $O(\log p)$ the recursion depth does not change. Hence, the overall communication complexity is $O(\log^2 p)$ including the REDUCE call each iteration.

After successfully determining the splitters all processors communicate the locally partitioned pieces with an ALL-TO-ALL exchange. Merging all received pieces leads to a globally sorted sequence over all processors. Due to the high communication volume communication-computation overlap is required to achieve good scaling efficiency. However, for collective operations MPI provides a very limited interface. While we can use a non-blocking ALL-TO-ALL we cannot operate on partially received pieces. For this reason we designed our own ALL-TO-ALL algorithm to pipeline communication and merging. Similar to the Butterfly algorithm processor i sends to destination $(i + r) \pmod p$ and receives from $(i - r) \pmod p$ in round r [33]. However we schedule communication requests only as long as a communication buffer of a fixed length is not completely allocated. As soon as *some* communication requests complete we schedule new requests while merging the received chunks. PGAS provides additional optimizations. For communication within a shared memory node we use a cache-efficient ALL-TO-ALL algorithm to minimize negative cache effects among the involved processors. Instead of scheduling send receive pairs processor ranks are reordered according to a Morton order. Data transfer is performed using one-sided communication mechanisms. Similar optimizations can be applied to processor pairs running on nearby nodes. We are preparing a paper to describe the involved optimizations in more detail. First experimental evaluations reveal that we can achieve up to 22% speedup compared to a single ALL-TO-ALL followed by a p -way local merge.

In the next section we demonstrate our performance scalability against a state-of-the-art implementation in Charm++.

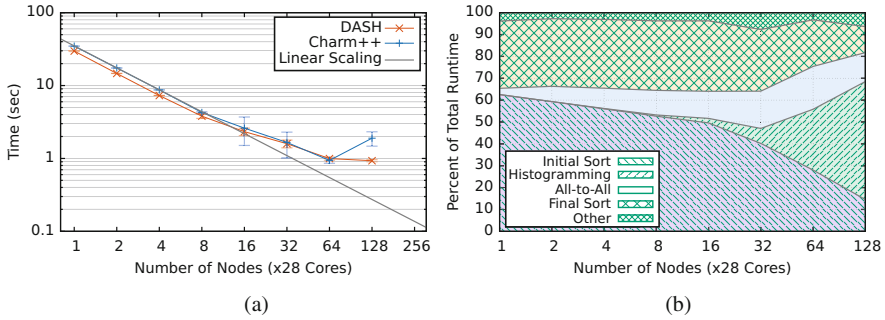


Fig. 18 Strong scaling study with Charm++ and DASH. (a) Median execution time. (b) Strong scaling behavior of `dash::sort`

3.2.4 Evaluation and Conclusion

We conducted the experiments on SuperMUC Phase 2 hosted at the Leibniz Supercomputing Center. This system is an island-based computing cluster, each equipped with 512 nodes. Each node has two Intel Xeon E5-2697v3 14-core processors with a nominal frequency of 2.6 GHz and 64 GB of memory, although only 56 GB are usable due to the operating system. Computation nodes are interconnected in a non-blocking fat tree with Infiniband FDR14 which achieves a peak bisection bandwidth of 5.1 TB/s. We compiled our binaries with Intel ICC 18.0.2 and linked the Intel MPI 2018.2 library for communication. The Charm++ implementation was executed using the most recent stable release.⁴ On each node we scheduled only 16 MPI ranks (28 cores available) because the Charm++ implementation requires the number of ranks to be a power of two. We emphasize that our implementation in DASH does not rely on such constraints.

The strong scaling performance results are depicted in Fig. 18a. We sort 28 GBytes of uniformly distributed 64-bit signed integers. This is the maximum memory capacity on a single node because our algorithm is not in-place. We always report the median time out of 10 executions along with the 95% confidence interval, excluding an initial warmup run. For Charm++ we can see wider confidence intervals. We attribute this to a volatile histogramming phase which we can see after analyzing generated log files in the Charm++ experiments. Overall, we observe that both implementations achieve nearly linear speedup with a low number of cores. Starting from 32–64 nodes scalability gets worse. DASH still achieves a scaling efficiency of ≈ 0.6 on 3500 cores while Charm++ is slightly below. Figure 18b visualizes the relative fraction of the most relevant algorithm phases in a single run. It clearly identifies histogramming as the bottleneck if we scale up the number of processors. This is not surprising because with 128 nodes (2048 ranks) each rank operates on only 8 MB of memory.

⁴v6.9.0, <http://charmplusplus.org/download/>.

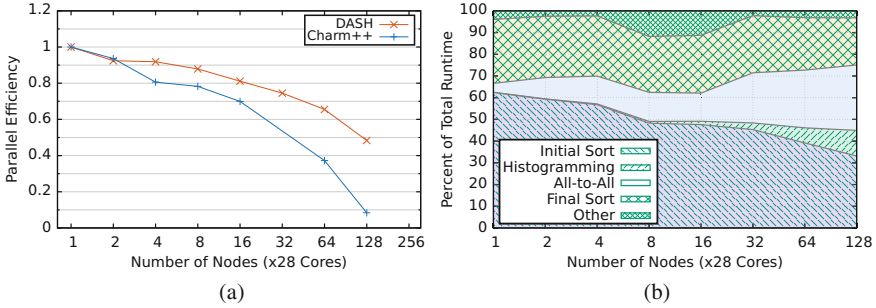


Fig. 19 Weak scaling study with Charm++ and DASH. (a) Weak scaling efficiency. (b) Weak scaling behavior of `dash::sort`

Figure 19a depicts the weak scaling efficiency. The absolute median execution time for DASH started from 2.3s on one node and ended with 4.6s if we scale to 128 nodes (3584 cores). As expected, the largest fraction of time is consumed in local sorting and the ALL-TO-ALL data exchange because we have to communicate 256 GB across the network. Figure 19b confirms this. The collective ALLREDUCE of $P - 1$ splitters among all processors in histogramming overhead is almost amortized from the data exchange which gives an overall good scalability for DASH. The Charm++ histogramming algorithm again shows high volatility with running times from 5–25 s, resulting in drastic performance degradation.

Our implementation shows good scalability on parallel machines with a large processor count. Compared to other algorithms we do not pose any assumptions on the number of ranks, the globally allocated memory volume or the key distribution. Performance measurements reveal that our general purpose approach does not result in performance degradation compare to other state-of-the-art algorithms. Our optimized MPI ALL-TO-ALL exchange with advanced PGAS techniques shows how we can significantly improve communication-computation overlap. Finally, the STL compliant interface enables programmers to easily integrate a scalable sorting algorithm into scientific implementations.

4 Use Cases and Applications

4.1 A Productivity Study: The Cowichan Benchmarks

In this section we present an evaluation of DASH focusing on productivity and performance by comparison with four established parallel programming approaches (Go, Chapel, Cilk, TBB) using the Cowichan set of benchmark kernels.

4.1.1 The Cowichan Problems

The Cowichan problems [36], named after a tribal area in the Canadian Northwest, is a set of small benchmark kernels that have been developed primarily for the purpose of assessing the usability of parallel programming systems. There are two versions of the Cowichan problems and here we restrict ourselves to a subset of the problems found in the second set. The comparison presented in this section is based on previous work by Nanz et al. [23] as we use their publicly available code⁵ to compare with our own implementation of the Cowichan benchmarks using DASH. The code developed as part of a study by Nanz et al. has been created by expert programmers in Go, Chapel, Cilk and TBB and can thus be regarded as idiomatic for each approach and free of obvious performance defects.

The five (plus one) problems we consider in our comparison are the following:

randmat: Generate a $(nrows \times ncols)$ matrix `mat` of random integers in the range $0, \dots, max - 1$ using a deterministic pseudo-random number generator (PRNG).

thresh: Given an integer matrix `mat`, and a thresholding percentage `p`, compute a boolean matrix `mask` of similar size, such that `mask` selects `p` percent of the largest values of `mat`.

winnow: Given an integer matrix `mat`, a boolean matrix `mask`, and a desired number of target elements `nelem`, perform a *weighted point selection* operation using sorting and selection.

outer: Given a vector of `nelem` (*row, col*) points, compute an $(nelem \times nelem)$ *outer product* matrix `omat` and a vector `vec` of floating point values based on the Euclidean distance between the points and the origin, respectively.

matvec: Given an $nelem \times nelem$ matrix `mat` and a vector `vec`, compute the matrix-vector product (row-by-row inner product) `res`.

chain: Combine the kernels in a sequence such that the output of one becomes the input for the next. I.e., `chain = randmat \circ thresh \circ winnow \circ outer \circ matvec`.

4.1.2 The Parallel Programming Approaches Compared

We compare our implementation of the Cowichan problems with existing solutions in the following four programming approaches.

Chapel [8] is an object-oriented partitioned global address space (PGAS) programming language developed since the early 2000s by Cray, originally as part of DARPA's High Productivity Computing Systems (HPCS) program. We have used Chapel version 1.15.0 in our experiments.

Go [10] is a general-purpose systems-level programming language developed at Google in the late 2000s that focuses on concurrency as a first-class concern. Go supports lightweight threads called *goroutines* which are invoked by prefixing

⁵<https://bitbucket.org/nanzs/multicore-languages/src>.

a function call with the `go` keyword. Channels provide the idiomatic way for communication between goroutines but since all goroutines share a single address space, pointers can also be used for data sharing. We have used Go version 1.8 in our experiments.

Cilk [6] started as an academic project at MIT in the 1990s. Since the late 2000s the technology has been extended and integrated as Cilk Plus into the commercial compiler offerings from Intel and more recently open source implementations for the GNU Compiler Collection (GCC) and LLVM became available. Cilk's initial focus was on lightweight tasks invoked using the `spawn` keyword and dynamic workstealing. Later a parallel loop construct (`cilk_for`) was added. We have used Cilk as integrated in Intel C/C++ compilers version 18.0.2.

Intel Threading Building Blocks (TBB) [25] is a C++ template library for parallel programming that provides tasks, parallel algorithms and containers using a work-stealing approach that was inspired by the early work on Cilk. We have used TBB version 2018.0 in our experiments, which is part of Intel Parallel Studio XE 2018.

4.1.3 Implementation Challenges and DASH Features Used

In this section we briefly describe the challenges encountered when implementing the Cowichan problems, a more detailed discussion can be found in a recent publication [13]. Naturally this small set of benchmarks only exercises a limited set of the features offered by either programming approach. However, we believe that the requirements embedded in the Cowichan problems are relevant to a wide set of other uses cases, including the classic HPC application areas.

Memory Allocation and Data Structure Instantiation The Cowichan problems use one- and two-dimensional arrays as the main data structures. 1D arrays are widely supported by all programming systems. True multidimensional arrays, however, are not universally available and as a consequence workarounds are commonly used. The Cilk and TBB implementation both adopt a linearized representation of the 2D matrix and use a single `malloc` call to allocate the whole matrix. Element-wise access is performed by explicitly computing the offset of the element in the linearized representation by `mat[i * ncols + j]`. Go uses a similar approach but bundles the dimensions together with the allocated memory in a custom type. In contrast, Chapel and DASH support a concise and elegant syntax for the allocation and direct element-wise access of their built-in multidimensional arrays. In the case of DASH, the distributed multidimensional array is realized as a C++ template class that follows the container concept of the standard template library (STL) [11].

Work Sharing In all benchmarks, work has to be distributed between multiple processes or threads, for example when computing the random values in *randmat* in parallel. *randmat* requires that the result be independent of the degree of parallelism used and all implementations solve this issue by using a separate deterministic seed value for each row of the matrix. A whole matrix row is the unit of work that is

distributed among the processes or threads. The same strategy is also used for *outer* and *product*.

Cilk uses `cilk_for` to automatically partition the matrix rows and TBB uses C++ template mechanisms to achieve a similar goal. Go does not offer built-in constructs for simple work sharing and the functionality has to be laboriously created manually using goroutines, channels, and ranges.

In Chapel this type of work distribution can simply be expressed as a parallel loop (`forall`).

In DASH, the work distribution follows the data distribution. I.e., each unit is responsible for computing on the data that is locally resident, the *owner computes* model. Each unit determines its locally stored portion of the matrix (guaranteed to be a set of rows by the data distribution pattern used) and works on it independently.

Global Max Reduction In *thresh*, the largest matrix entry has to be determined to initialize other data structures to their correct size. The Go reference implementation doesn't actually perform this step and instead just uses a default size of 100, Go is thus not discussed further in this section.

In Cilk a `reducer_max` object together with a parallel loop over the rows is employed to find the maximum. Local maximal values are computed in parallel and then the global maximum is found using the reducer object. In TBB a similar construct is used (`tbb::parallel_reduce`). In these approaches finding the local maximum and computing the global maximum are separate steps that require a considerable amount of code (several 10s of lines of code).

Chapel again has the most concise syntax of all approaches, the maximum value is found simply by `nmax = max reduce matrix`. The code in the DASH solution is nearly as compact, by using the `max_element()` algorithm to find the maximum. Instead of specifying the matrix object directly, in DASH we have to couple the algorithm and the container using the iterator interface by passing `mat.begin()` and `mat.end()` to denote the range of elements to be processed.

Parallel Histogramming *thresh* requires the computation of a global cumulative histogram over an integer matrix. Thus, for each integer value $0, \dots, nmax - 1$ we need to determine the number of occurrences in the given matrix in parallel. The strategy used by all implementations is to compute one or multiple histograms by each thread in parallel and to later combine them into a single global histogram.

In DASH we use a distributed array to compute the histogram. First, each unit computes the histogram for the locally stored data, by simply iterating over all local matrix elements and updating the local histogram (`histo.local`). Then `dash::transform` is used to combine the local histograms into a single global histogram located at unit 0. `dash::transform` is modeled after `std::transform`, a mutating sequence algorithm. Like the STL variant, the algorithm works with two input ranges that are combined using the specified operation into an output range.

Parallel Sorting *winnow* requires the sorting of 3-tuples using a custom comparison operator. Cilk and TBB use a parallel shared memory sort. Go and Chapel

call their respective internal sort implementations (quicksort in the case of Chapel) which appears to be unparallelized. DASH can take advantage of a parallel and distributed sort implementation based on histogram sort cf. 3.2.

4.1.4 Evaluation

We first compare the productivity of DASH compared to the other parallel programming approaches before analyzing the performance differences.

Productivity We evaluate programmer productivity by analyzing source code complexity. Table 1 shows the lines of code (LOC) used in the implementation for each kernel, counting only lines that are not empty or comments. Of course, LOC is a crude approximation for source code complexity but few other metrics are universally accepted or available for different programming languages. LOC at least gives a rough idea for source code size, and, as a proxy, development time, likelihood for programming errors and productivity. The overall winners in the productivity category are Chapel and Cilk, which achieve the smallest source code size for three benchmark kernels. For most kernels, DASH also achieves a remarkably small source code size considering that the same source code can run on shared memory as well as on distributed memory machines.

Performance As the hardware platform for our experiments we have used one or more nodes of SuperMUC Phase 2 (SuperMUC-HW) with Intel Xeon E5-2697 (Haswell) CPUs with 2.6 GHz, 28 cores and 64 GB of main memory per node.

Single Node Comparison We first investigate the performance differences between DASH and the four established parallel programming models on a single Haswell node. We select a problem size of $nrows = ncols = 30,000$ because this is the largest size that successfully ran with all programming approaches.

Table 2 shows the absolute performance (runtime in seconds) and relative runtime (compared to DASH) when using all 28 cores of a single node. Evidently, DASH is the fastest implementation, followed by Cilk and TBB. Chapel and especially Go can not deliver competitive performance in this setting.

Analyzing the scaling behavior in more detail (not shown graphically due to space restrictions) by increasing the number of cores used on the system from 1 to 28 reveals a few interesting trends. For outer and randmat, DASH, Cilk and TBB

Table 1 Lines-of-code (LOC) measure for each kernel and programming approach, counting non-empty and non-comment lines only

	DASH	Go	Chapel	TBB	Cilk
Randmat	18	29	14	15	12
Thresh	31	63	30	56	52
Winnow	67	94	31	74	78
Outer	23	38	15	19	15
Product	19	27	11	14	10

Table 2 Performance comparison for each kernel and programming approach using all cores on one node of SuperMUC-HW

	Absolute runtime (sec.)					Relative runtime				
	DASH	Go	Chapel	TBB	Cilk	DASH	Go	Chapel	TBB	Cilk
Randmat	0.12	5.84	0.36	0.15	0.19	1.00	48.69	3.03	1.26	1.58
Thresh	0.20	0.64	0.53	0.41	0.40	1.00	3.19	2.64	2.06	2.00
Winnow	2.99	366.13	256.40	9.45	4.28	1.00	122.45	85.75	3.16	1.43
Outer	0.25	1.18	0.39	0.27	0.31	1.00	4.70	1.56	1.06	1.24
Product	0.06	0.46	0.19	0.12	0.13	1.00	7.66	3.15	2.01	2.16

behave nearly identical in terms of absolute performance and scaling behavior. The small performance advantage of DASH can be attributed to better NUMA locality of DASH, where all work is done on process-local data. For thresh and winnow DASH can take advantage of optimized parallel algorithms (for global max reduction and sorting, respectively) whereas these operations are performed sequentially in some of the other approaches. For product the DASH implementation takes advantage of a local copy optimization to improve data locality. The scaling study reveals that this optimization at first costs performance but pays off at larger core counts.

Multinode Scaling We next investigate the scaling of the DASH implementation on up to 16 nodes (448 total cores) of SuperMUC-HW. None of the other approaches can be compared with DASH in this scenario. Cilk and TBB are naturally restricted to shared memory systems by their threading-based nature. Go realizes the CSP (communicating sequential processes) model that would, in principle, allow for a distributed memory implementation but since data sharing via pointers is allowed, Go is also restricted to a single shared memory node. Finally, Chapel targets both shared and distributed memory systems, but the implementation of the Cowichan problems available in this study is not prepared to be used with multiple locales and cannot make use of multiple nodes (it lacks the `dmapped` specification for data distribution).

The scaling results are shown in Fig. 20 for two data set sizes. In Fig. 20 (left) we show the speedup relative to one node for a small problem size ($nrows = ncols = 30,000$) and in Fig. 20 (right) we show the speedup of a larger problem size ($nrows = ncols = 80,000$) relative to two nodes, since this problem is too big to fit into the memory of a single node.

Evidently for the smaller problem size, the benchmark implementations reach their scaling limit at about 10 nodes, whereas the larger problem sizes manage to scale well even to 16 nodes, with the exception of the product benchmark which shows the worst scaling behavior. This behavior can be explained by the relatively large communication requirement of the product benchmark kernel.

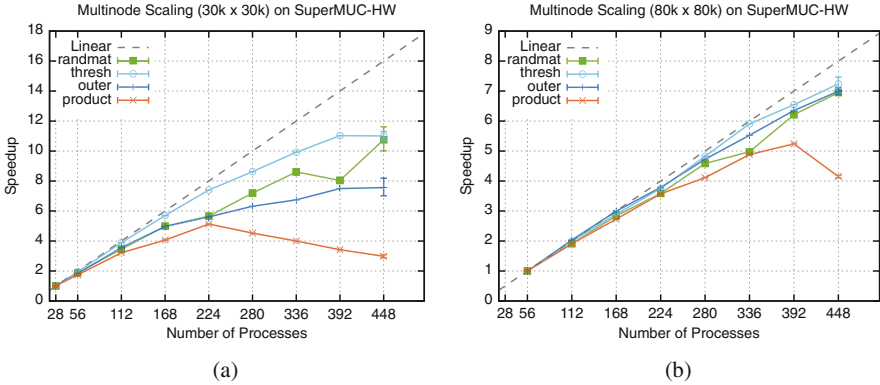


Fig. 20 Scaling behavior of the Cowichan benchmarks with up to 16 nodes on SuperMUC-HW. (a) Multinode Scaling, $30 \times 30k$ Matrix. (b) Multinode Scaling: $80k \times 80k$ Matrix

4.1.5 Summary

In this section we have evaluated DASH, a new realization of the PGAS approach in the form of a C++ template library by comparing our implementation of the Cowichan problems with those developed by expert programmers in Cilk, TBB, Chapel, and Go. We were able to show that DASH achieves both remarkable performance and productivity that is comparable with established shared memory programming approaches. DASH is also the only approach in our study where the same source code can be used both on shared memory systems and on scalable distributed memory systems. This step, from shared memory to distributed memory systems is often the most difficult for parallel programmers because it frequently goes hand in hand with a re-structuring of the entire data distribution layout of the application. With DASH the same application can seamlessly scale from a single shared memory node to multiple interconnecting nodes.

4.2 Task-Based Application Study: LULESH

The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) is part of the Department of Energy’s Coral proxy application benchmark suite [19]. The domain typically scales with the number of processes and is divided into a grid of nodes, elements, and regions of elements. The distribution and data exchange follows a 27-point stencil with three communication steps per timestep.

The reference implementation of LULESH uses MPI send/rcv communication to facilitate the boundary exchange between neighboring processes and OpenMP worksharing constructs are used for shared memory parallelization. Instead, we

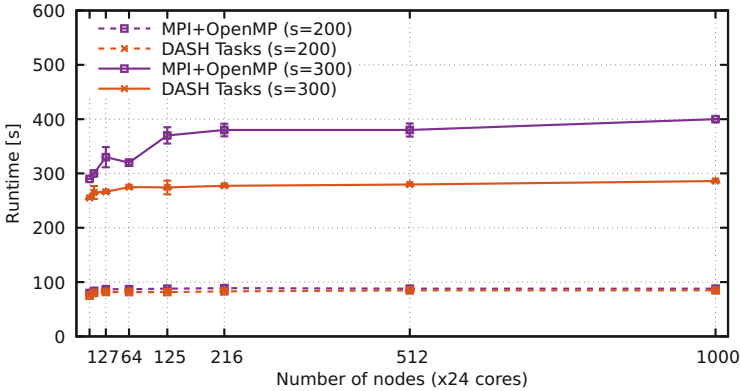


Fig. 21 Lulesh reference implementation and Lulesh using DASH tasks running on a Cray XC40

iteratively ported LULESH to use DASH distributed data structures for neighbor communication and DASH tasks for shared memory parallelization, with the ability to partially overlap communication and computation.

The first step was to adapt the communication to use DASH distributed data structures instead of MPI [12]. In order to gradually introduce tasks, we started by porting the OpenMP worksharing loop constructs with to use the DASH task-loop construct discussed in Sect. 2.1.3. In a second step, the iteration chunks of the taskloops were connected through dependencies, allowing a breadth-first scheduler to execute tasks from different task-loop statements concurrently. In a last step, a set of high-level tasks has been introduced that encapsulate the task-loop statements and coordinate the computation and communication tasks.

The resulting performance at scale on a Cray XC40 is shown in Fig. 21. For larger problem sizes ($s = 300^3$ elements per node), the speedup at scale of the DASH port over the reference implementation is about 25%. For smaller problem sizes ($s = 200^3$), the speedup is significantly smaller at about 5%. We believe that further optimizations in the tasking scheduler may yield improvements even for smaller problem sizes.

5 Outlook and Conclusion

We have presented an overview of our parallel programming approach DASH, focusing on recent activities in the areas of support for task-based execution, dynamic locality, parallel algorithms, and smart data structures. Our results show that DASH offers a productive programming environment that also allows programmers to write highly efficient and scalable programs with performance on-par or exceeding solutions relying on established programming systems.

Work on DASH is not finished. The hardware landscape in high performance computing is getting still more complex, while the application areas are getting more diverse. Heterogeneous compute resources are common, nonvolatile memories are making their appearance and domain specific architectures are on the horizon. These and other challenges must be addressed by DASH to be a viable parallel programming approach for many users.

The challenge of utilizing heterogeneous computing resources, primarily in the form of graphics processing units (GPUs) used in high performance computing systems, is addressed in a project building on DASH funded by the German Federal Ministry of Education and Research (BMBF) called MEPHISTO. We close our report on DASH with a short discussion of MEPHISTO.

5.1 MEPHISTO

The PGAS model simplifies the implementation of programs for distributed memory platforms, but data locality plays an important role for performance critical applications. The *owner-computes* paradigm aims to maximize the performance by minimizing the intra-node data movement. This focus on locality also encourages the usage of shared memory parallelism. During the DASH project the partners already experimented with shared memory parallelism and how it can be integrated into the DASH ecosystem. OpenMP was integrated into suitable algorithms, experiments with Intel's Thread Building Blocks as well as conventional POSIX-thread-based parallelism were conducted. These, however, had to be fixed for a given algorithm: a user had no control over the acceleration and the library authors had to find sensible configurations (e.g. level of parallelism, striding, scheduling). Giving users of the DASH library more possibilities and flexibility is a focus of the MEPHISTO project.

The MEPHISTO project partly builds on the work that has been done in DASH. One of its goal is to integrate abstractions for better data locality and heterogeneous programming with DASH. Within the scope of MEPHISTO two further projects are being integrated with DASH:

- Abstraction Library for Parallel Kernel Acceleration (ALPAKA) [22]
- Low Level Abstraction of Memory Access⁶ (LLAMA)

ALPAKA is a library that adds abstractions for node-level parallelism for C++ programs. Once a kernel is written with ALPAKA, it can be easily ported to different accelerators and setups. For example a kernel written with ALPAKA can be executed in a multi-threaded CPU environment as well as on a GPU, for example using the CUDA backend. ALPAKA provides optimized code for each accelerator that can be further customized by developers. To switch from one back end to

⁶<https://github.com/mephisto-hpc/llama>.

another requires just the change of one type definition in the code so that the code is portable.

Integrating ALPAKA and DASH brings flexibility for node-level parallelism within a PGAS environment: switching an algorithm from a multi threaded CPU implementation to an accelerator now only requires passing one more parameter to the function call. It also gives the developer an interface to control *how* and *where* the computation should happen. The kernels of algorithms like `dash::transform_reduce` are currently extended to work with external *executors* like ALPAKA. Listing 5 shows how executors are used to offload computation using ALPAKA. Note that the interface allows offloading to other implementations as well. In the future DASH will support any accelerator that implements a simple standard interface for node-level parallelism.

```

1 policy.executor().bulk_tway_execute (
2     [=] (size_t      block_index,
3         size_t      element_index,
4         T*          res,
5         value_type* block_first) {
6     res[block_index] = binary_op (
7                             res[block_index],
8                             unary_op(block_first[element_index])
9         ↪ );
10 },
11 in_first, // a "shape"
12 [&]() -> std::vector<std::future<T>>& {
13     return results;
14 },
15 std::ignore); // shared state (unused)

```

Listing 5 Offloading using an executor inside `dash::transform_reduce`. The executor can be provided by MEPHISTO

LLAMA on the other hand focuses solely on the format of data in memory. DASH already provides a flexible *Pattern Concept* to define the data placement for a distributed container. However, LLAMA gives developers finer grained control over the data layout. DASH's patterns map elements of a container to locations in memory, but the layout of the elements itself is fixed. With LLAMA, developers can specify the data layout with a C++ Domain Specific Language (DSL) to fit the application's needs. A typical example is a conversion from Structure of Arrays (SoA) to Array of Structures (AoS) and vice versa. But also more complex transformations like projections are being evaluated.

Additionally to the integration of LLAMA, a more flexible, hierarchical and context-sensitive pattern concept is being evaluated. Since the current patterns map elements to memory locations in terms of units (i.e., MPI processes), using other sources for parallelism can be a complex task. Mapping elements to (possibly multiple) accelerators was not easily possible. Local patterns extend the existing patterns. By matching elements with *entities* (e.g. a GPU), the node-local data may be assigned to other compute units beside processes.

Both projects are currently evaluated in the context of DASH to explore how the PGAS programming model can be used more flexibly and efficiently.

Acknowledgments The authors would like to thank the many dedicated and talented individuals who contributed to the success of the DASH project. DASH started out as “just an idea” that was turned into reality with the help of Benedikt Bleimhofer, Daniel Rubio Bonilla, Daniel Diefenthaler, Stefan Effenberger, Colin Glass, Kamran Idrees, Benedikt Lehmann, Michael Maier, Matthias Maiterth, Yousri Mhedheb, Felix Mössbauer, Sebastian Oeste, Bernhard Saumweber, Jie Tao, Bert Wesarg, Huan Zhou, Lei Zhou.

We would also like to thank the German research foundation (DFG) for the funding received through the SPPEXA priority programme and initiators and managers of SPPEXA for their foresight and level-headed approach.

References

1. Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., Thibault, S.P.: Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Trans. Parallel Distrib. Syst.* (2018). <https://doi.org/10.1109/TPDS.2017.2766064>
2. Axtmann, M., Bingmann, T., Sanders, P., Schulz, C.: Practical massively parallel sorting. In: *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pp. 13–23. ACM, New York (2015)
3. Ayguade, E., Coptly, N., Duran, A., Hoefflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.* **20**, 404–418 (2009). <https://doi.org/10.1109/TPDS.2008.105>
4. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11 (2012). <https://doi.org/10.1109/SC.2012.71>
5. Blleloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagher, M.: A comparison of sorting algorithms for the connection machine cm-2. In: *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 3–16. ACM, New York (1991)
6. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: *Cilk: An Efficient Multithreaded Runtime System*, vol. 30. ACM, New York (1995)
7. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemariner, P., Dongarra, J.: Dague: a generic distributed dag engine for high performance computing. In: *Proceedings of the Workshops of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2011 Workshops)*, pp. 1151–1158 (2011)
8. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.* **21**, 291–312 (2007)
9. 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**, 519–538 (2005)
10. Donovan, A.A., Kernighan, B.W.: *The Go Programming Language*, 1st edn. Addison-Wesley Professional, Boston (2015)
11. Fuchs, T., Furlinger, K.: A multi-dimensional distributed array abstraction for PGAS. In: *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016), Sydney*, pp. 1061–1068 (2016). <https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0150>

12. Furlinger, K., Fuchs, T., Kowalewski, R.: DASH: a C++ PGAS library for distributed data structures and parallel algorithms. In: Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016), Sydney, pp. 983–990 (2016). <https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0140>
13. Furlinger, K., Kowalewski, R., Fuchs, T., Lehmann, B.: Investigating the performance and productivity of DASH using the cowichan problems. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops, Tokyo (2018). <https://doi.org/10.1145/3176364.3176366>
14. Grossman, M., Kumar, V., Budimlić, Z., Sarkar, V.: Integrating asynchronous task parallelism with OpenSHMEM. In: Workshop on OpenSHMEM and Related Technologies, pp. 3–17. Springer, Berlin (2016)
15. Harsh, V., Kalé, L.V., Solomonik, E.: Histogram sort with sampling. CoRR abs/1803.01237 (2018). <http://arxiv.org/abs/1803.01237>
16. Helman, D.R., JáJá, J., Bader, D.A.: A new deterministic parallel sorting algorithm with an experimental evaluation. *ACM J. Exp. Algorithmics* **3**, 4 (1998)
17. Kaiser, H., Brodowicz, M., Sterling, T.: Paralex an advanced parallel execution model for scaling-impaired applications. In: 2009 International Conference on Parallel Processing Workshops (2009). <https://doi.org/10.1109/ICPPW.2009.14>
18. 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, PGAS '14. ACM, New York (2014). <http://doi.acm.org/10.1145/2676870.2676883>
19. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Technical Report. LLNL-TR-641973 (2013)
20. Kowalewski, R., Jungblut, P., Furlinger, K.: Engineering a distributed histogram sort. In: 2019 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, Piscataway (2019)
21. Kumar, V., Zheng, Y., Cavé, V., Budimlić, Z., Sarkar, V.: HabaneroUPC++: a compiler-free PGAS library. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS '14. ACM, New York (2014). <https://doi.org/10.1145/2676870.2676879>
22. Matthes, A., Widera, R., Zenker, E., Worpitz, B., Huebl, A., Bussmann, M.: Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the alpaka library (2017). <https://arxiv.org/abs/1706.10086>
23. Nanz, S., West, S., Da Silveira, K.S., Meyer, B.: Benchmarking usability and performance of multicore languages. In: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 183–192. IEEE, Piscataway (2013)
24. OpenMP Architecture Review Board: OpenMP Application Programming Interface, Version 5.0 (2018). <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>
25. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly & Associates, Sebastopol (2007)
26. Robison, A.D.: Composable parallel patterns with intel cilk plus. *Comput. Sci. Eng.* **15**, 66–71 (2013). <https://doi.org/10.1109/MCSE.2013.21>
27. Saraswat, V., Almasi, G., Bikshandi, G., Cascaval, C., Cunningham, D., Grove, D., Kodali, S., Peshansky, I., Tardieu, O.: The asynchronous partitioned global address space model. In: The First Workshop on Advances in Message Passing, pp. 1–8 (2010)
28. Saukas, E., Song, S.: A note on parallel selection on coarse-grained multicomputers. *Algorithmica* **24**(3-4), 371–380 (1999)
29. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In: Proceedings of the 22nd Annual International Conference on Supercomputing, ICS '08. ACM, New York (2008). <https://doi.org/10.1145/1375527.1375568>
30. Slaughter, E., Lee, W., Treichler, S., Bauer, M., Aiken, A.: Regent: a high-productivity programming language for HPC with logical regions. In: SC15: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2015). <https://doi.org/10.1145/2807591.2807629>

31. Sundar, H., Malhotra, D., Biros, G.: Hyksort: a new variant of hypercube quicksort on distributed memory architectures. In: Proceedings of the 27th international ACM conference on International Conference on Supercomputing, pp. 293–302. ACM, New York (2013)
32. Tejedor, E., Farreras, M., Grove, D., Badia, R.M., Almasi, G., Labarta, J.: A high-productivity task-based programming model for clusters. *Concurr. Comp. Pract. Exp.* **24**, 2421–2448 (2012). <https://doi.org/10.1002/cpe.2831>
33. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. *Int. J. High Perform. Comput. Appl.* **19**(1), 49–66 (2005). <https://doi.org/10.1177/1094342005051521>
34. Tillenius, M.: SuperGlue: a shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM J. Sci. Comput.* **37**, C617–C642 (2015). <http://epubs.siam.org/doi/10.1137/140989716>
35. Tsugane, K., Lee, J., Murai, H., Sato, M.: Multi-tasking execution in pgas language xscalablemp and communication optimization on many-core clusters. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. ACM, New York (2018). <https://doi.org/10.1145/3149457.3154482>
36. Wilson, G.V., Irvin, R.B.: Assessing and comparing the usability of parallel programming systems. University of Toronto. Computer Systems Research Institute (1995)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

