Enabling Malleability for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics using LAIK

Amir Raoofy[1], Dai Yang[1], Josef Weidendorfer[2], Carsten Trinitis[1], Martin Schulz[1]

Abstract: Malleability, i.e., the ability for an application to release or acquire resources at runtime, has many benefits for current and future HPC systems. Implementing such functionality, however, is already difficult in newly written code and an even more daunting challenge when considering the enhancement of existing legacy code to support malleability. LAIK is a recent proposal for a dynamic and flexible parallel programming model that separates data and execution into two orthogonal concerns. These properties promise easier malleability as the runtime can partition resources dynamically as needed, as well as easier incremental porting of existing MPI code. In this paper, we explore the malleability of LAIK with the help of `laik-lulesh`, a LAIK-based port of LULESH, a proxy application from the CORAL benchmark suite. We show the steps required for porting the application to LAIK, and we present detailed scaling experiments that show promising results.

Keywords: LAIK, LULESH, MPI, Malleable Application, SPMD

1 Introduction

With the Sierra and Summit systems at LLNL and ORNL, respectively, High Performance Computing (HPC) has reached its last milestone before exascale computing. On this road, the environment of HPC systems has become more and more dynamic; most existing HPC applications, however, are rigid and lack support for malleability. In order to cope with the needed goals in efficiency, energy consumption, and fault tolerance [Ke11], we must overcome such rigidness and enhance HPC applications with more flexibility.

Much effort has been put into enabling malleability in HPC using a wide range of approaches. Some of them target application transparency. An example is MPI Sessions [Ho16], a proposed extension to the MPI standard that allows the instantiation of MPI multiple times at runtime. Other studies focus on the enhancement of HPC system software: Flux [Ah18] is a next-generation job scheduler that allows users to allocate and deallocate resources dynamically in a fine-grained way. Some of the existing work also focuses on the combination of minimal application modification and system software: Invasive MPI

[1] Technical University of Munich, Chair of Computer Architecture and Parallel Systems, Boltzmannstr. 3, 85748 Garching, Germany, {raoofy, yang, trinitis, schulzm}@in.tum.de
[2] Leibniz-Rechenzentrum, Boltzmannstr. 1, 85748 Garching, Germany, josef.weidendorfer@lrz.de

and Invasive Resource Manager [Ur12] provide a modified MPI and a modified SLURM resource manager, which allows for different phases in applications demanding different types and amounts of resources.

A pure application-oriented approach is LAIK [WYT17], a library assisting in dynamically scheduling the execution of HPC applications by separating the concerns for data location and computation. Initially designed for fault tolerance purposes, LAIK also enables elasticity that can be controlled from the outside. It supports incremental porting of an existing application, allowing the user to reuse most of the existing codebases. In this paper, we demonstrate the malleability properties of LAIK on a well-known proxy application - LULESH; an iterative solver for the Sedov Blast Problem, which is highly relevant in real life. LULESH is part of the CORAL benchmarks[3]. It was ported to different parallel programming models [Ka13] for investigation, which allows us to make interesting comparisons with our results.

The main contributions of this paper are: (1) we provide a fully functional port (`laik-lulesh`) of LULESH to LAIK with enhanced malleability features and; (2) we identify the limitations of LAIK with an in-depth performance analysis of our `laik-lulesh` application.

2 Related Work

2.1 LULESH Ports LULESH - as one of the Coral benchmarks - has been ported to a number of programming models and languages, including OpenMP, CUDA, AMP, OpenACC, Loci, Liszt, Chapel, and Charm++. A summary of all versions of LULESH is presented by Karlin et at. [Ka12]. An in-depth study of some ports of LULESH and their performance evaluation is presented by Karlin et at. [Ka13]. According to the authors, `Loci` [LG05] and `Charm++` provide comparable performance to the reference code using MPI.

2.2 Programming Models Many programming models support writing malleable applications. `Charm++`[KK93] is a machine independent parallel programming system. Its dynamic load balancing distributes workloads between different machines at runtime. Adaptive MPI (AMPI) [HLK04] is a flexible MPI implementation based on `Charm++`, with MPI ranks running on virtual Processing Units (PU). The mapping between physical and virtual PUs is done by the `Charm Runtime System (RTS)`, which benefits from the flexibility of Charm++. `Legion` [Ba12] is a data-centric programming model that provides automatic mechanisms for data handling and processing. Based on user-specified workflow mechanisms, the runtime takes care of all data movements during execution. The main difference between `Legion` and `LAIK` is that LAIK works with index spaces and partitionings abstracting data distribution, and it informs the user when changes need to be applied. The physical distribution and processing of the data still rely on the user's specification. Other task-based programming models such as OmpSs[Ma15] and StarPU [Au11] also provide fine granular control of resource and data mapping and processing. With a task-based programming model,

[3] `https://asc.llnl.gov/CORAL-benchmarks/`

malleability is ensured as both workload and data can be easily migrated over hardware resources across task boundaries.

2.3 Benchmarks Many benchmarks are used for performance evaluation of HPC libraries and programming models, e.g., the NAS Parallel Benchmarks[Ba91], the CORAL Benchmarks, and the Rodinia Benchmarks [Ch09].

3 LAIK - Library for Automatic Data Migration

LAIK[WYT17] is a lightweight library for automatic data migration in parallel applications featuring an SPMD approach similar to the one used in many MPI codes. The application programmer is required to transfer the responsibility of partitioning his or her application data to LAIK. However, the actual partitioning algorithm, called a **Partitioner**, which assigns portions of an abstract index space to processes, remains under application programmer control. The application programmer can specify a customized *Partitioner* using callbacks.

Examples provided as presets are the “all” partitioning (the whole index space is replicated to all processes, i.e., requesting complete local copies) or “disjunctive block distribution” (every participating process holds a portion of the index space).

The communication for data structures is implicitly specified as a set of transitions between different partitionings. A transition allows users to specify complex combinations of copy operations, broadcasts and reductions. Transitions can be declared in advance and result in a sequence of abstract communication actions on the index space, to be executed later by the user or by LAIK. The latter is done when asking LAIK to manage the data that is bound to an index space. Executing a transition for such data results in direct communication as required. We can use this approach in LAIK to react to internal and external requests to change the current partitioning by calculating a new partitioning, even on a modified process group, allowing the application to become malleable.

LAIK features several different API-levels, which provide different levels of abstraction, allowing programmers to port their application incrementally. The basic “index space” API only transfers the responsibility of index space partitioning to LAIK. The corresponding transitions and action sequences are calculated by LAIK, while the actual communication is carried out by the user application. The more sophisticated “data” API allows developers to hand over complete control of data structures to LAIK, allowing automatic communication. This way, all communication is hidden from the user, resulting in lean, purly data-oriented code. In addition, LAIK does internal optimization for different communication patterns, which reduces development costs.

Previous studies with LAIK [WYT17; Ya18] have already shown the effectiveness and efficiency of LAIK in both performance and basic malleability, but are limited to simple data structures with simple communication patterns. In this paper, we evaluate LAIK’s capability by porting the rather complex program LULESH to LAIK. For our experiments, we use the published open-source LAIK-Version on Github[4].

4 Porting The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) to LAIK

Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) is a benchmark that solves the Sedov blast problem using Lagrangian hydrodynamics [HKG].

In each time step of the simulation, a number of hydrodynamic fields are updated by the computational kernels. Data is stored either related to the initially cube-shaped finite elements covering a 3d domain ('elemental' data) or related to the vertices of the finite elements ('nodal' data). After the initialization of the fields, coordinates and boundary conditions the timestep loop executes until convergence.

For our work, we use the MPI/OpenMP hybrid implementation of LULESH 2.0[5], where MPI communication is explicitly coded. This is typical for many HPC applications: developers do not want to start from scratch for an existing application to add new functionality such as malleability.

The selected implementation of LULESH is based on domain decomposition for data partitioning and uses non-blocking communication (`Isend/Irecv`) at various points in each time step. Two main kernels that require communication are executed in each iteration: (1) stencil-wise updates of data structures such as velocity gradients which require halo exchange at borders of domains. (2) aggregation of contributions from element quantities to the surrounding nodes, e.g., in the calculation of force fields, which also requires a halo exchange followed by an aggregation. Our goal in porting LULESH to LAIK is to keep the number of changes as small as possible: code accessing data structures as well as computational kernels should remain unchanged. LAIK is used for two tasks: a) it is responsible for regular value updates in iterations (e.g., for halo exchanges), replacing MPI in the original code; b) it has to migrate data for re-distribution to support malleability. Correspondingly, porting can be done in two steps: first, we replicate original communication by letting LAIK maintain the data structures that get updated in each iteration. Second, for malleability, also data structures used purely locally have to be maintained by LAIK, as it also needs to be migrated on re-distribution. Furthermore, small modifications are required in the main iteration loop to check for re-partitioning requests and trigger data repartitioning in LAIK. It is important to mention that *LULESH 2.0 only supports a cubic number of MPI processes*. This limitation still holds for our LAIK implementation as we are neither changing the partitioning algorithm nor the compute kernel. In the following, we present the steps for porting LULESH to LAIK. The major changes made to the LULESH program execution flow are illustrated in Pseudocode 1 vs. Pseudocode 2.

[4] `https://github.com/envelope-project/laik`, commit e504385
[5] The base version for our port is `https://github.com/LLNL/LULESH`, commit a328f79.

Pseudocode 1: Simplified Pseudocode for MPI Implementation of LULESH [HKG]

```
MPI_Init();
Domain locDom ← InitMeshDecomposition(rank, size, sz);
while (!endOfSimulation) do
    CalcTimeIncrement();
    LagrangeLeapFrog();
end
MPI_Finalize();
```

Pseudocode 2: Simplified Pseudocode for `laik-lulesh` [HKG]

```
Laik_init_mpi();
Domain locDom ← InitMeshDecomposition(rank, size, sz);
while (!endOfSimulation) do
    if (needRepart) then
        Domain newDom ← InitMeshDecomposition(newRank, newSize, sz);
        Laik_repartitioning_and_migrate(locDom, newDom);
        locDom ← newDom;
    end
    CalcTimeIncrement();
    LagrangeLeapFrog();
end
Laik_finalize();
```

Step 1: Adaptation of data structures requiring MPI communication. LULESH uses asynchronous communication for force fields, namely `fx`, `fy`, `fz` and `nodalMass` followed by aggregation. LAIK provides such communication patterns through so-called “transitions” between different partitionings, which are trigged by calling the API call `laik_switch`. For that, we create halo regions by introducing *overlapping* partitions on a global nodal index space, so that the neighboring tasks share one layer of nodes. As LAIK does not provide such partitioning out of the box, we implement the partitioner algorithm ourselves as a core part of our `laik-lulesh` port. While this partitioner is similar to the one in the reference code, it uses a different layout: the reference code uses `std::vector` with a compact `xyz` (Figure 1 right) layout, while our implementation of `laik-lulesh` relies on a non-compact `xyz` (Figure 1 left) layout. It divides a local domain into “slices”. The reason for this layout is that, although LULESH works on a 3D domain, it is mapped into a 1D data structure during execution. As we stick to the original kernels, we also need to provide 1D data storage.

From the perspective of LAIK this data is shared between the two neighboring tasks and updated by each task independently. After each iteration, we call `laik_switch`, which triggers the reduction operations on shared data, replacing any explicit communication code. LULESH uses an asynchronous halo exchange for velocity gradient fields, i.e., `delv_xi`, `delv_eta` and `delv_zeta`, and these data structures needed to be ported to LAIK as well. For

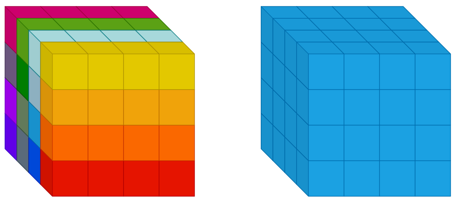
Fig. 1: Illustration of `laik_slices` for a problem with local domains of size $4 \times 4 \times 4$ elements. Each color represents one `laik_slice` in the partitioning. Current implementation of `laik_lulesh` relies on many slices (left) and the reference code relies on a compact allocation of data (right).

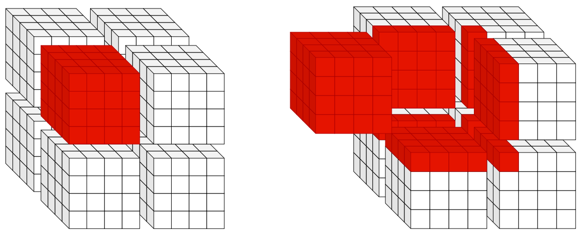
Fig. 2: Element partitionings: *exclusive* partitioning (left) and *halo* (right) partitioning. This figure is an illustration for a problem with $8 \times 8 \times 8$ elements using 8 tasks (8 sub-domains, e.g., cubes to be processes by each task). Each sub-domain (indicated in red) contains $4 \times 4 \times 4$ elements in exclusive `partitioning` and halo `partitioning` extends it with one layer.

this, we create two partitionings — *exclusive* and *halo* — again using custom partitioners. A switch between them triggers the communications for the halo exchange (see Figure 2).

Internally, LULESH uses `std::vector` as its data container and implements an accessor interface layer on top. This allows modifications of the underlying data structure by only replacing `std::vector`. For that, we introduce `laik_vector` encapsulating LAIK structures and implement the required accessor interfaces. In the end, all data structures with communication requirements are ported to LAIK and all MPI calls are eliminated.

Step 2: Enabling Malleability. LAIK can support malleability by having control over the underlying data structures. As LULESH uses a number of data structures in addition to those mentioned above, those need to be handed over to LAIK as well. For that, we provide additional partitions according to the needed data distribution before and after repartitioning. As above, calling `laik_switch` triggers the required MPI communication under the hood and re-distributes data according to the target partitioning. In addition, we modify the main loop to handle repartitioning requests. If a process is no longer part of an active calculation after repartitioning, this process is discarded by calling `laik_finalize`.

Additional Optimizations. We consider multiple optimizations to improve the performance of `laik-lulesh`. First, the transitions are executed in every iteration. Therefore, we use LAIK's advanced APIs in order to pre-calculate and cache of the `transitions` and corresponding `action_sequences` between the above-mentioned partitionings. Moreover, our implementation creates many `1D` slices and, as referencing data between `laik_switches` normally invalidates the pointers for all the slices, we use LAIK's reservation API, which guarantees the validity of addresses of data across "switch"es.

5 Evaluation

To show the performance of our ported LULESH code, we carried out strong and weak scaling tests on SuperMUC Phase II (SuperMUC) which consists of 3072 nodes, each equipped with 2 Intel Haswell Xeon Processor E5-2697v3 and 64 GB of main memory.

5.1 Weak Scaling We execute both the reference code `lulesh2.0` and our ported `laik-lulesh` five times with problem size 16^3 corresponding to $s = 16$ (parameter `-s`). We report the average runtime per iteration without initialization and finalization. The upper bound of the number of iteration is 10,000. The normalized runtime per iteration is represented in Figure 3 by the box plots and noted on the y-axis. On the x-axis, the number of MPI Tasks used in experiments is given. We can see an increase in iteration runtime with an increasing number of MPI tasks from our code. However, the reference code scales almost perfectly with only a slight increase. The blue line in Figure 3 represents overhead, which scales up with the number of processes. Our hypothesis for the source of this increasing overhead is the lack of support for asynchronous communication in LAIK and we, therefore, continue with strong scaling experiments to pinpoint the source of this overhead.

The overhead in our `laik-lulesh` is in an acceptable range for a mid-sized run (e.g., 10% at 512 processes), which is a realistic use case scenario for **a malleable application**. For extreme scaling, however, LAIK must be further adapted and tuned.

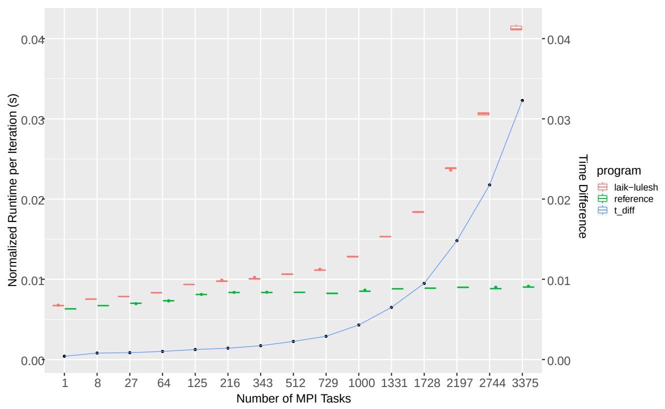

Fig. 3: Weak scaling comparison of `laik-lulesh` with reference LULESH

5.2 Strong Scaling Due to the limited support for an only cubic number of processes, the following limitation applies: let $C = s^3 * p$ be the global 3-dimensional problem size to be

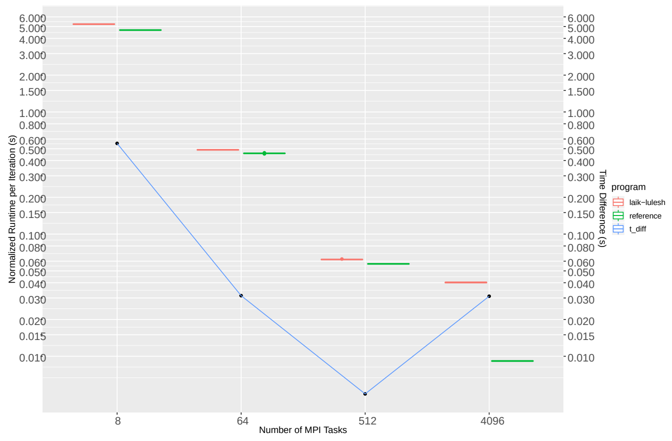

Fig. 4: Strong Scaling Comparison of `laik-lulesh` with reference LULESH

held constant for all strong scaling experiments, with s being the local one-dimensional problem size (parameter `-s i`); furthermore, let p be the number of MPI processes and $S = C^{\frac{1}{3}}$; the following implication applies: $(C = s^3 * p) => (S = s * p^{\frac{1}{3}})$. As $p^{\frac{1}{3}}$ and s must be natural numbers we set up our strong scaling experiments with $p^{\frac{1}{3}}$ being powers of 2 and $S = 256$. The resulting corresponding tuples of (p, s) used in this paper for strong scaling are $(1^3 = 1, 256)$, $(2^3 = 8, 128)$, $(4^3 = 64, 64)$, $(8^3 = 512, 32)$ and $(16^3 = 4096, 16)$. The results from these experiments are illustrated in Figure 4. Note that the y-axis is log(2)-scaled. As expected, similar scaling behavior can be observed for both the LAIK version and the reference version with up to 512 processes. With 4096 processes, our port shows significant overhead (factor 2x slower than the reference code). In addition, the overhead curve first decreases then increases with a large number of processes. Figure 4 shows a relatively constant overhead for experiments with 8, 64, and 512 processes. This is most likely due to a constant overhead of using 1D slices in the LAIK implementation. In addition, Figure 4 shows a problem similar to weak scaling for `laik-lulesh` with a large number of processes. This is very likely the result of lack of support for asynchronous communication in LAIK, which scales with the number of point-to-point communication (and the number of processes).

5.3 Repartitioning Using LAIK, we can now shrink the number of MPI processes during the execution of `laik-lulesh`. To test how this shrinking affects the scaling behavior before and after repartitioning, we conduct a series of scaling experiments. We set up the repartitioning experiments with $p^{\frac{1}{3}}$ being powers of 2 and $S = 64$ and enforce a repartitioning to the smaller, next supported number of MPI processes in the strong scaling tests. This results in the following repartitioning experiments: from 8 to 1, from 64 to 8 and from 512 to 64. We fix the number of iterations (parameter `-i`) to 2000 iterations for all experiments and execute the kernel for 250 iterations with the initial number of MPI processes and then perform the

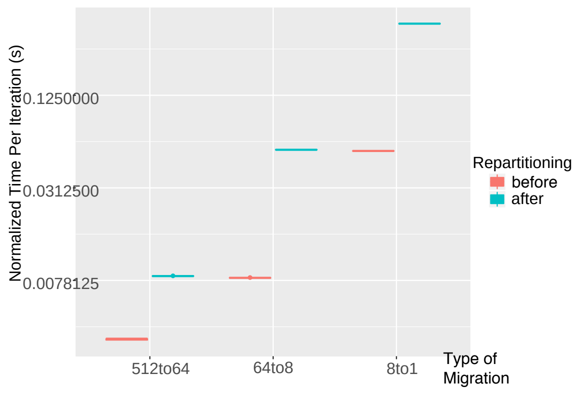

Fig. 5: Strong scaling result for `laik-lulesh` with enforced repartitioning

repartitioning. Finally, the kernel is executed another 1750 iterations until completion. We execute a total of five runs per configuration on SuperMUC.

The results are provided in Figure 5. On the x-axis, we list the type of migration, and on the y-axis the normalized time per iteration in log scale, respectively. Both curves show the same trend. In addition, the runtime for a given scale (e.g., p=64) is almost the same, regardless of whether it is the initial number of processes or the state after repartitioning. The required time for repartitioning is presented in Table 1.

As for the effectiveness of the repartitioning function, our test shows little to no overhead on the normalized kernel execution time of `laik-lulesh` and also a migration has little impact.

Tab. 1: Required Time for `laik-lulesh` with Enforced Repartitioning

Configuration	Time for Repartitioning
512to64	~1.5678s
64to8	~0.8803s
8to1	~1.6979s

6 Conclusions and Future Work

We presented `laik-lulesh`, a port of the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) to LAIK. The ported code gains enhanced malleability at runtime. Moreover, it is capable of repartitioning its data as needed. All data structures, as well as all MPI communication, are transferred to LAIK's responsibility, while the actual kernel did not have to be modified. Results from weak and strong scaling experiments show a low constant overhead and an increasing overhead when scaling the number of processes. The constant part of the overhead stems from the additional abstraction introduced by LAIK. The overall overhead stays acceptable for up to 1000 MPI processes, which is a typical configuration for malleable applications. Further, the repartitioning experiments show promising performance result, as no additional overhead from repartitioning can be observed. This shows that LAIK is a useful approach to assist programmers to enable malleability for existing HPC applications.

As next steps, we will enhance LAIK's asynchronous communication behavior. In addition, we will work on the reduction of the constant overhead by using a proposed layout interface from LAIK. Finally, we plan to work with the original LULESH team to overcome the limitation of only allowing a cubic number of processes.

Acknowledgment This work is partially funded by the German Federal Ministry for Education and Research under grant title 01|H16010D (Project ENVELOPE). Compute resources on SuperMUC are sponsored by Leibniz Supercomputer Centre under grant title pr27ne.

References

[Ah18] Ahn, D. H.; Bass, N.; Chu, A.; Garlick, J.; Grondona, M.; Herbein, S.; Koning, J.; Patki, T.; Scogland, T. R. W.; Springmeyer, B.; Taufer, M.: Flux: Overcoming Scheduling Challenges for Exascale Workflows. In: 2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS). Pp. 10–19, Nov. 2018.

[Au11] Augonnet, C.; Thibault, S.; Namyst, R.; Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23/2, pp. 187–198, 2011.

[Ba12] Bauer, M.; Treichler, S.; Slaughter, E.; Aiken, A.: Legion: Expressing locality and independence with logical regions. In: High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for. IEEE, pp. 1–11, 2012.

[Ba91] Bailey, D. H.; Barszcz, E.; Barton, J. T.; Browning, D. S.; Carter, R. L.; Dagum, L.; Fatoohi, R. A.; Frederickson, P. O.; Lasinski, T. A.; Schreiber, R. S., et al.: The NAS parallel benchmarks. The International Journal of Supercomputing Applications 5/3, pp. 63–73, 1991.

[Ch09] Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J. W.; Lee, S.-H.; Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, pp. 44–54, 2009.

[HKG] Hornung, R. D.; Keasler, J. A.; Gokhale, M. B.: Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory, tech. rep. LLNL-TR-490254, Livermore, CA, USA, pp. 1–17.

[HLK04] Huang, C.; Lawlor, O.; Kalé, L. V.: Adaptive MPI. In (Rauchwerger, L., ed.): Languages and Compilers for Parallel Computing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 306–322, 2004.

[Ho16] Holmes, D.; Mohror, K.; Grant, R. E.; Skjellum, A.; Schulz, M.; Bland, W.; Squyres, J. M.: MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale. In: Proceedings of the 23rd European MPI Users' Group Meeting. ACM, pp. 121–129, 2016.

[Ka12] Karlin, I.; Bhatele, A.; Chamberlain, B. L.; Cohen, J.; Devito, Z.; Gokhale, M.; Haque, R.; Hornung, R.; Keasler, J.; Laney, D.; Luke, E.; Lloyd, S.; McGraw, J.; Neely, R.; Richards, D.; Schulz, M.; Still, C. H.; Wang, F.; Wong, D.: LULESH Programming Model and Performance Ports Overview, tech. rep. LLNL-TR-608824, Livermore, CA, USA, Dec. 2012, pp. 1–17.

[Ka13] Karlin, I.; Bhatele, A.; Keasler, J.; Chamberlain, B. L.; Cohen, J.; Devito, Z.; Haque, R.; Laney, D.; Luke, E.; Wang, F., et al.: Exploring traditional and emerging parallel programming models using a proxy application. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. IEEE, pp. 919–932, 2013.

[Ke11] Kerbyson, D.; Vishnu, A.; Barker, K.; Hoisie, A.: Codesign Challenges for Exascale Systems: Performance, Power, and Reliability. Computer 44/11, pp. 37–43, Nov. 2011.

[KK93] Kale, L. V.; Krishnan, S.: CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications. OOPSLA '93, ACM, Washington, D.C., USA, pp. 91–108, 1993.

[LG05] Luke, E. A.; George, T.: Loci: A rule-based framework for parallel multi-disciplinary simulation synthesis. Journal of Functional Programming 15/3, pp. 477–502, 2005.

[Ma15] Martsinkevich, T.; Subasi, O.; Unsal, O.; Cappello, F.; Labarta, J.: Fault-Tolerant Protocol for Hybrid Task-Parallel Message-Passing Applications. In: 2015 IEEE International Conference on Cluster Computing. Pp. 563–570, Sept. 2015.

[Ur12] Ureña, I. A. C.; Riepen, M.; Konow, M.; Gerndt, M.: Invasive MPI on Intel's Single-Chip Cloud Computer. In (Herkersdorf, A.; Römer, K.; Brinkschulte, U., eds.): Architecture of Computing Systems – ARCS 2012. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 74–85, 2012.

[WYT17] Weidendorfer, J.; Yang, D.; Trinitis, C.: LAIK: A Library for Fault Tolerant Distribution of Global Data for Parallel Applications. In: Konferenzband des PARS'17 Workshops. Hagen, Germany, 2017.

[Ya18] Yang, D.; Weidendorfer, J.; Kuestner, T.; Trinitis, C.; Ziegler, S.: Enabling Application-Integrated Proactive Fault Tolerance. In. Vol. 32, IOS Press, Bologna, Italy, pp. 475–484, 2018.