LAIK: A Library for Fault Tolerant Distribution of Global Data for Parallel Applications

Josef Weidendorfer[1] Dai Yang[2] Carsten Trinitis[3]

Abstract: HPC applications usually are not written in a way that they can cope with dynamic changes in the execution environment, such as removing or integrating new nodes or node components. However, for higher flexibility with regard to scheduling and fault tolerance strategies, adequate application-integrated reaction would be worthwhile. However, with legacy MPI codes, this is difficult to achieve. In this paper, we present Lightweight Application-Integrated data distribution for parallel worKers (LAIK), a lightweight library for distributed index spaces and associated data containers for parallel programs supporting fault tolerance features. By giving LAIK control over data and its partitioning, the library can free compute nodes before they fail and do replication for rollback schemes on demand. Applications become more adaptive to changes of available resources. We show a simple example which integrates our LAIK library and present first results on a prototype implementation.

Keywords: High Performance Computing, Parallel Data Containers, Parallel Programming Models, Data Partitioning

1 Introduction

Modern High Performance Computing (HPC) systems become more and more parallel nowadays. To achieve the goal of exascale computing, technologies like on-chip parallelism are increasingly used in these supercomputers [SDM10]. This development greatly challenges programmers, especially on managing the data across all these distributed compute nodes. For future systems, Defense Advance Research Projects Agency (DARPA) expects that existing technology such as *Checkpoint & Restart* will require extensive amount of resources, which contradicts to the expected high efficiency of HPC. Therefore, it is no longer sufficient to handle the evolving requirements on fault tolerance and reliability [BBea08]. Towards exascale computing, different approaches are possible to increase the system reliability. For example, attempts[4] to add fault tolerant components [FD00] to the 4th version of Message Passing Interface (MPI) exist. Furthermore, mechanisms like process level migration [Wa12] or virtual machines [Na07] are presented to achieve a fault tolerant environment which is transparent to the application programmer. However, these solutions often require a significant amount of resources similar to the classical *Checkpoint & Restart* technique and limit an application's scalability, providing only limited applicability to emerging exascale requirements.

[1] TU München, Fakultät für Informatik, Boltzmannstr. 3, 85748 Garching, josef.weidendorfer@tum.de
[2] TU München, Fakultät für Informatik, Boltzmannstr. 3, 85748 Garching, dai.yang@tum.de
[3] TU München, Fakultät für Informatik, Boltzmannstr. 3, 85748 Garching, carsten.trinitis@tum.de
[4] `https://svn.mpi-forum.ort/trac/MPI-Forum-Web/wiki/FaultToleranceWikiPage`

In extension of the application-transparent strategies mentioned above, we point out that there is another possibility to make applications fault tolerant. Instead of speculating on the programmers' intention and applications' execution, one can ask the programmer to write the program in a manner that includes fault tolerance features. This results not only in reduced resource management overhead by the framework that provides fault tolerance, but also keeps this framework lightweight, with the application keeping control over any additional overhead required e.g. by redundancy. This way, we can achieve simple, lightweight and scalable fault tolerance on modern exascale HPC.

However, one cannot expect the programmer to write fault-tolerance features from scratch. To this end, in this paper we propose a lightweight library to achieve exactly that: LAIK focuses on providing fault tolerance by taking over control of the partitioning of data containers. If a node or node component is expected to fail, the library can be instructed to free the corresponding part of an HPC system from any compute load and application data. This is done by repartitioning within LAIK. Furthermore, to help with spontaneous failures, LAIK can be asked to maintain redundant copies of important data structures by updating copies on nearside nodes (local checkpointing). The update frequency can be controlled by a predicted probability of failure. If a fault-tolerant MPI implementation detects a node fault, a local rollback and repartitioning can be done with the help of LAIK.

In this work, we present our requirements for LAIK, an API proposal, and a first prototype implementation using an MPI communication backend. The latter is available as open-source on Github[5].

2 Related Work

The currently most widely used de-facto parallel programming model for legacy HPC applications is MPI [Fo12], which allows to do message passing or one-sided communication via mapping of remote address spaces, both at a low level. For higher productivity, the Partitioned Global Address Space (PGAS) model was proposed which provides global address spaces and allows to program for good locality of accesses by making the fact whether an address is local or remote explicit. Implementations come either in new programming languages e.g. Chapel [ZCC07] and X10 [SBea14], or as libraries such as Global Array Toolkit (GAT) [Ni06], GASPI [ABea13], and DASH [FGea14, IFG16]. The drawback of Chapel or X10 is that programs have to be rewritten which can be painful for legacy code. The library GAT allows programmers to do put and get operations from local memory to data structures in a global address space. Similarly, GASPI provides a global address space with distributed data and allows access via RDMA (remote direct memory access) operations provided e.g. by Infiniband, which should result in better asynchronous communication than MPI. DASH uses C++ templates to provide a selection of standard data structures for the application programmer. The communication of DASH is built on top of DART [ZMea15] - a run-time system which provides abstraction of different communication libraries. All of the mentioned approaches provide application programmers a

[5] `https://github.com/envelope-project/laik`

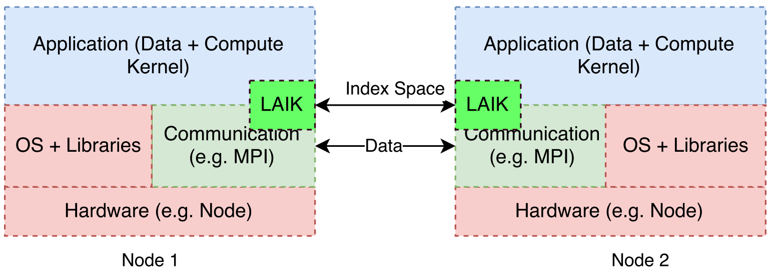

Figure 1: LAIK and Communication Backend

full interface for all possible communication needs of an application. In contrast to that, LAIK only provides one specific functionality and can be used in cooperation with existing communication libraries, by asking a communication backend to perform required actions for LAIK. The latter can be provided by the application itself to ensure correct embedding into already existing communication behavior of an application. Other approaches, such as *Legion* [TBA13], a region based programming system, handles data partitioning in a similar manner as LAIK. It shows the capability of keeping data coherence with non-disjoint data partitions by providing a such runtime with corresponding C++ API. However, it is not lightweight and incremental. An existing application cannot be ported without significant effort. FTC-Charm++ [ZSK04], a protocol for in-memory checkpointing and restart for MPI applications, provides scalable fault tolerance for applications written in Charm++ [KK93]. It provides very similar concepts which allows application to run with less processors after a crash. However, it relays on the Charm++ runtime. Thus, only a limited class of applications can benefit from this approach.

3 Features of the LAIK Library

The idea behind LAIK is to provide lightweight management for the distribution of global data containers for parallel applications using index spaces. By giving LAIK control over partitioning, it should be able to provide application-integrated fault tolerance features. To this end, the library is expected to sit in-between the application and the communication library used by the application, as shown in Figure 1.

First, we state our requirements guiding the design. Similar to MPI, LAIK should support SPMD-like programming with collective functions. That is, every participant in a parallel application, a LAIK task, executes the same code. Most LAIK functions are collective operations, this is all LAIK tasks have to call into this same LAIK function to ensure correct behavior. For highest flexibility of using LAIK with existing application codes, LAIK is expected to work in cooperation with other communication libraries e.g. MPI. Communication backends in LAIK should have a documented API and may be implemented by the application itself. For convenience, we will provide standard backends for MPI and multi-threaded shared memory (which mostly reduces communication to synchronization) which may be customized by applications. Furthermore, we want porting of existing codes

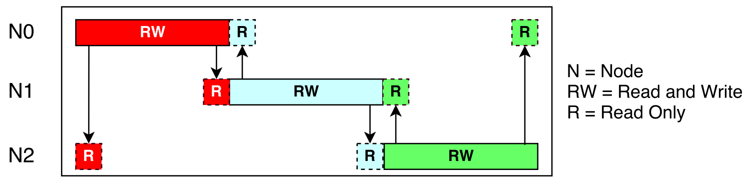

Figure 2: An example of a partitioning specification for a 1d data structure.

to LAIK to be done in an incremental way, step by step: the programmer can put one data structure after the other into LAIK's responsibility. This should not mean that the actual allocation of memory resources has entirely to be done by LAIK, as the programmer still may want to use allocator functions of another library. To this end, programmers should be able to specify an allocator interface (*alloc/realloc/free*) which LAIK will use to get real memory resources. Applications may use complex data structures such as compressed matrices in an application-specific format. It would be difficult to make LAIK aware of how to handle all kinds of such data structures. To still allow LAIK to take control over partitioning of such data, the programmer should be able to use LAIK's core abstraction, which is about distributed (possibly multi-dimensional) index spaces. Whenever the partitioning of the index space changes, the application can request to get a call-back with parameters specifying which parts of the index space should migrate among LAIK tasks.

With these requirements in mind, we now list the features we want LAIK to provide. The goal of LAIK is to provide a fault tolerant, yet efficient way of distributing data across different nodes. Towards this goal, the following functionalities must be supported:

3.1 Partitioning and Repartitioning of Data

For application data to be globally distributed among different parallel tasks, LAIK should be able to take control over data partitioning. Here, a data partition for a given task specifies which data should be available locally to the task using direct memory access. Different program phases or algorithms may expect different parts of data to be available locally. This requires the declaration of multiple partitions, between which the application may want to switch back and forth. For such switches, LAIK is expected to calculate and execute the data transfers required to satisfy the requirements for task-local data accesses as specified by the partitions. To allow for minimal data exchange, the application should specify the type of access done during the time a partitioning is active. E.g. switching to a partitioning where after the switch all data will be overwritten anyway does not need any communication at all. Access types are *read-only, read-write, write-only*. A data element may be required to be locally available by multiple tasks for reading, but usually only by one task for writing. However, multiple writers can be supported if a conflict resolution is provided which decides about the resulting value. Reduction operations such as Sum, Product, or Minimum, are typical conflict resolutions. The operations are automatically triggered when the program switches to a partitioning with read access to the given data elements, which decrease programming effort.

Figure 2 shows an example partitioning of 1d data among nodes N0, N1, and N2 (rows). For each element, exactly one node is the owner with read and write access. As shown in the figure, we want LAIK to provide copies for reading of the neighbor elements at the border of owned regions. This is useful for e.g. stencil codes. This example shows that it is useful to "switch" to the same partitioning, which ensures consistency of the values of elements to be accessible at multiple nodes. Thus, instead of a switch, it is better to talk about *enforcing consistency requirements* of a given partitioning. In the example, this will result in copying data as shown by the arrows in the figure. To support communication asynchronous to computation, a program first has to acquire access to parts of a LAIK partition with different access permission right after a switch: in the example separately for the RW and R parts. This may allow to already do computations on the inner parts of the owned regions while communication for the border elements is still going on, which probably speeds up the computation.

3.2 Coupling Partitions of Data Structures

Applications often use multiple different data structures at the same time. Thus, if LAIK decides that only a given portion of a data structure will be available locally to one task, the application may want simultaneous access to elements in other data structures. To provide flexible coupling of partitions, the user actually should be able to specify coupling of (partial) index spaces according to the data pattern of an algorithm. E.g. for matrix vector multiplication, we want to couple the row dimension of the 2d index space of the matrix to the 1d space of the vector. Another example are compute kernels which, for writing to an element in one data structure, may need read access to corresponding data elements in another structure and its neighbors in the index space (for so-called ghost layers in stencil codes). Switching between partitions of one data structure thus may result in automatic switching also for other data structures.

3.3 Communication backend

LAIK shall support different communication libraries to execute data migration demands due to switches between partitions. Standard backends to be supported are *MPI* and *Shared Memory*. The latter is wanted to allow multi-threaded programs to use LAIK for synchronization. Furthermore, LAIK should work together with any communication library used by an application. This should be supported by allowing application programmers to implement their own LAIK backend.

3.4 Efficient Fault Tolerance

Application data shall be recoverable upon system failure. The LAIK library shall support both recovery-based (e.g. *Checkpoint & Restart*) and proactive (e.g. *Fault Prediction*) fault tolerance. We want LAIK to support local recovery, that is on node failure, a near-side node

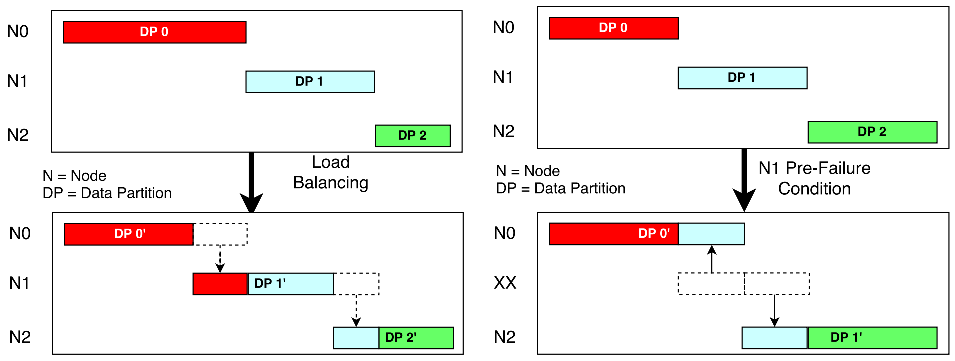

Figure 3: Load Balancing (left), Remove work from N1 due to failure prediction (right).

should be able to take over the computation of the failed node by using duplicated data. This way, LAIK can support a more efficient scheme than classical global *Checkpoint & Restart*. For failure prediction, we expect that there is a way to do online monitoring of hardware health via adequate sensors. However, this is out of scope for LAIK. Any support to react to outside sources should be implementable as a thin layer on top of LAIK. This may regularly poll (synchronous to program phases) for incoming messages using IPC mechanisms or protocols like Message Queue Telemetry Transport (MQTT).

3.5 Load Balancing

A task within a parallel application usually has a work load depending on the size of data it has access to. As LAIK provides partitions to tasks, this influences the work load distribution. We want LAIK to support automatic workload distribution via regular repartitioning of data structures. For that, LAIK should be able to make use of task-related profile data (such as time measurements of program phases). To be able to support different load balancing mechanisms, both static and dynamic, a key-value store may be needed to have historical data available. This must be persistent to allow static load balancing.

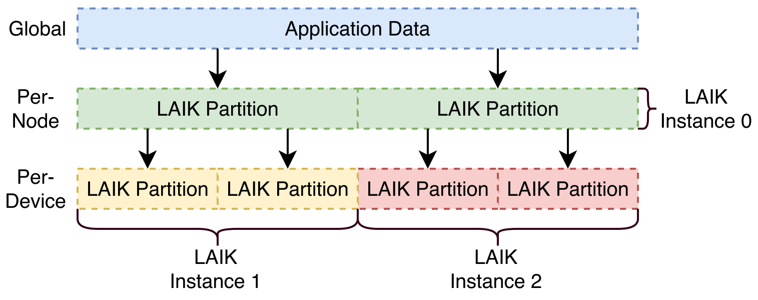

Figure 4: Hierarchical LAIK Instances

Figure 3 (left) shows an example of data re-distribution for load balancing. First, the data is distributed unequally across nodes. Upon a load balancing request, recalculation of partition borders from profile data results in a data redistributing with better load balancing. Figure 3 (right) shows an example of data reacting to failure prediction as part of a proactive fault tolerance scheme. A pre-failure condition for node N1 results in LAIK initiating a repartitioning such that the failing compute node is excluded from execution. Here we assume that a system health monitor with an integrated fault predictor exists. Related works such as [LZea06] shows the possibility of predicting system failure.

3.6 Hierarchical Data Partitioning

In order to make the LAIK library even more usable, LAIK shall be able to run in a nested multi-instance mode. It should be possible to connect different LAIK instances at different hierarchy levels. Thus, a partition in one LAIK instance should map to a top-level index space in a lower LAIK instance. To this end, LAIK index spaces must be able to shrink or expand. Data migration request may be passed between instances. Thus, a repartitioning happening in a lower LAIK instance is ignored in a higher LAIK instance, but a repartitioning of a higher instance will potentially result in size changes of index spaces in lower LAIK instances, resulting in forced repartition of the lower instances. The expected use for this nesting functionality is that large HPC systems may have a hierarchical topology with different resource capabilities (network, processors, accelerators) at different levels. This can be represented by adequate LAIK nesting. In the simplest case this would be inter-node and intra-node. Figure 4 shows a hierarchical configuration using different nested LAIK instances. Application data is first divided into LAIK data partitions node-wise. These data partitions then are mapped to top-level spaces if the inner LAIK instances. These divide the data again to be assigned to different devices (for example CPU cores or multiple GPUs) within the node.

4 Examples

Figure 5 shows the implementation of a parallel vector sum using LAIK with the MPI backend. The compiled binary can be run with "*mpirun*" using the number of LAIK tasks as requested as number of MPI tasks. First, the vector is initialized on the master node using a **master partitioning**, giving all elements write access on the master node only. To actually write the values, the partitions have to be "mapped" to local memory using a canonical ordering: the index into the 1d array is equal to the offset in memory (more data layout options are planned for 2d or 3d data). Afterwards, we switch to a **stripe partitioning** with equal size for all tasks, resulting in MPI messages from master to other nodes. After another mapping request for direct access, partial sums are generated with the result written to another LAIK array with only one element per task. This shows how LAIK can be asked to do a sum reduction, which actually happens when this array is switched to a partitioning, in which only master having access to print out the final result.

```
#include "laik-backend-mpi.h"
int main(int argc, char* argv[])
{
Laik_Instance* inst = laik_init_mpi(&argc, &argv);
Laik_Group* world = laik_world(inst);

// allocate global 1d double (8 bytes) array: 1 mio entries
Laik_Data* a = laik_alloc_1d(world, 8, 1000000);

// initialize at master (others do nothing)
laik_set_new_partitioning(a, LAIK_PT_Master, LAIK_AP_WriteOnly);
laik_map(a, LAIK_DL_CANONICAL, (void**) &base, &count);
double* base; uint64_t count;
for(uint64_t i = 0; i < count; i++) base[i] = (double) i;

// distribute data equally among all tasks, do partial sums
laik_set_new_partitioning(a, LAIK_PT_Stripe, LAIK_AP_ReadWrite);
laik_map(a, LAIK_DL_CANONICAL, (void**) &base, &count);
double mysum = 0.0;
for(uint64_t i = 0; i < count; i++) mysum += base[i];

// write partial sums as input for sum reduction
Laik_Data* sum = laik_alloc_1d(world, 8, 1);
laik_set_new_partitioning(sum, LAIK_PT_All, LAIK_AP_SUM);
laik_map(sum, LAIK_DL_CANONICAL, (void**) &base, &count);
*base = mysum;
// master-only: does sum reduction to be read at master
laik_set_new_partitioning(sum, LAIK_PT_Master, LAIK_AP_ReadOnly);
if (laik_myid(world) == 0) {
laik_map(sum, LAIK_DL_CANONICAL, (void**) &base, &count);
printf("Sum:␣\%.0f", *base[0]);
}
laik_finalize(inst);
}
```

Figure 5: Example using LAIK for parallel vector initialization and sum.

Figure 6 shows excerpts from an example using LAIK to perform computation on a sparse matrix distributed in row dimension for partitions containing a similar number of non-zero elements. For this, we use an index space which also is bound to vector "*r*", which has the same number of elements as there are rows in the sparse matrix. To calculate the requested **weighted stripe partitioning**, LAIK will call into an application provided function (*getEW*) that returns a weight for each element in a 1d index space, equal to the number of elements in that row. Here the structure type "*SpM*" encapsulates a matrix stored in CSR format. Afterwards, each task performs a loop over all matrix rows belonging to its partition.

```
double getEW(uint64_t i, void* d) {
  SpM* m = (SpM*) d;
  return (double) (m->row[i + 1] - m->row[i]); }

main {
  ...
  Laik_Data* r = laik_alloc_1d(world, 8, SIZE);
  Laik_Space* s = laik_get_space(r);
  Laik_Partitioning* p = laik_new_base_partitioning(s,
    LAIK_PT_Stripe, LAIK_AP_ReadWrite);
  laik_set_index_weight(p, getEW, matrix);
  laik_set_partitioning(resD, p);

  laik_my_slice(p, &from, &to);
  laik_map(r, LAIK_DL_CANONICAL, (void**) &res, &count);
  for(int r = from; r < to; r++) { res[r-from] += ... }
  ...
}
```

Figure 6: Using element-wise weighting balancing workload involving a sparse matrix.

```
0/3 partng-0: 0:[0-999999], 1:(empty), 2:(empty)
...
0/3 partng-1: 0:[0-333332], 1:[333333-666665], 2:[666666-999999]
0/3 partng-0 => partng-1: locl: [0-333332]
       send: [333333-666665] => T1, [666666-999999] => T2
1/3 partng-0 => partng-1: recv: T0 => [333333-666665]
2/3 partng-0 => partng-1: recv: T0 => [666666-999999]
```

Figure 7: Excerpt of debug output from Example 1 (*vsum*). This shows the computed migration of indexes for switching between master and Stripe partitioning.

5 Prototype Implementation and First Results

Our current prototype of the LAIK library[6] comes with an MPI backend, support for 1D data and different types of partitioning. The most important one is a **stripe partitioning**, which slices a 1D space into an ordered sequence of consecutive partitions, one for each task. This partitioning type supports element-wise and task-wise weighting. Element-wise weighting is shown in Fig. 6; task-wise weighting similarly uses a function called by LAIK e.g. returning time measurements for load balancing.

The LAIK prototype source code provides examples as shown in the previous section (*vsum* and *spmv*, respectively). As expected, with enough parallel work load, we can achieve the same scalability as if an example would have used MPI directly. Optionally, debug output can be printed which includes the calculated partitions of index spaces as well as the transition actions needed to migrate from one partitioning to another. This is shown in Fig. 7.

[6] Commit 790ea403 at `https://github.com/envelope-project/laik`

6 Conclusion and Future Work

In this paper, we presented our design and a prototypical implementation of LAIK, a lightweight library for the distribution of data containers among tasks of a parallel application. The idea is to separate the decision making of how to best partition application data from the application code. This allows partitioning strategies which not only take program-internal information (such as profiling data for load balancing) but also external sources into account. This enables support for application-integrated fault tolerance features, such as pro-active requests for removing computation from nodes predicted to fail soon. As future work, we will support real-world applications which need coupling of index spaces of used data structures, as well as support for multi-dimensional data. Further, we want to show how a local checkpoint/restart functionality involving rollback only for data recovered with the help of redundant data duplication within LAIK. Furthermore, we will implement a mechanism to adapt fault predictions from outside. Although adapted applications using MPI in this paper, it is to be expected that legacy MPI application cannot continue the computation, if one of the ranks suffers from failure. Therefore, we will have a closer look into potential fault tolerant MPI implementations, which allows to change the size of MPI communicator and exclude some ranks at runtime.

Acknowledgment This work is a part of collaborative project ENVELOPE, which is funded by German Federal Ministry of Education and Research (BMBF) under grant title 01IH16010D.

References

[ABea13] Alrutz, Thomas; Backhaus, Jan; et. al.: GASPI: A Partitioned Global Address Space Programming Interface. In: Facing the Multicore-Challenge III. volume 7686 of Lecture notes in computer science, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 135 – 136, 2013.

[BBea08] Bergman, Keren; Borkar, Shekhar; et. al.: Exascale computing study: Technology challenges in achieving exascale systems. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep, 15, 2008.

[FD00] Fagg, Graham E; Dongarra, Jack J: FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In: European Parallel Virtual Machine/Message Passing Interface User Group Meeting. Springer, pp. 346–353, 2000.

[FGea14] Fürlinger, Karl; Glass, Colin; et. al.: DASH: Data Structures and Algorithms with Support for Hierarchical Locality. In: Euro-Par 2014 Workshops (Porto, Portugal). 2014.

[Fo12] Forum, Message Passing Interface: , MPI: A Message-Passing Interface Standard Version 3.0, 2012.

[IFG16] Idrees, Kamran; Fuchs, Tobias; Glass, Colin W: Effective use of the PGAS Paradigm: Driving Transformations and Self-Adaptive Behavior in DASH-Applications. In: Proceedings of the First International Workshop on Program Transformation for Programmability in Heterogeneous Architectures, held in conjunction with the CGO Conference. arXiv preprint arXiv:1603.01536, 2016.

[KK93] Kale, Laxmikant V; Krishnan, Sanjeev: CHARM++: a portable concurrent object oriented system based on C++. In: ACM Sigplan Notices. volume 28. ACM, pp. 91–108, 1993.

[LZea06] Liang, Yinglung; Zhang, Yanyong; et. al.: Bluegene/l failure analysis and prediction models. In: Dependable Systems and Networks, 2006. DSN 2006. International Conference on. IEEE, pp. 425–434, 2006.

[Na07] Nagarajan, Arun Babu; Mueller, Frank; Engelmann, Christian; Scott, Stephen L: Proactive fault tolerance for HPC with Xen virtualization. In: Proceedings of the 21st annual international conference on Supercomputing. ACM, pp. 23–32, 2007.

[Ni06] Nieplocha, Jarek; Palmer, Bruce; Tipparaju, Vinod; Krishnan, Manojkumar; Trease, Harold; Aprà, Edoardo: Advances, applications and performance of the global arrays shared memory programming toolkit. The International Journal of High Performance Computing Applications, 20(2):203–231, 2006.

[SBea14] Saraswat, Vijay; Bloom, Bard; et. al.: , X10 Language Specification Version 2.5, 2014.

[SDM10] Shalf, John; Dosanjh, Sudip; Morrison, John: Exascale computing technology challenges. In: International Conference on High Performance Computing for Computational Science. Springer, pp. 1–25, 2010.

[TBA13] Treichler, Sean; Bauer, Michael; Aiken, Alex: Language support for dynamic, hierarchical data partitioning. In: ACM SIGPLAN Notices. volume 48. ACM, pp. 495–514, 2013.

[Wa12] Wang, Chao; Mueller, Frank; Engelmann, Christian; Scott, Stephen L: Proactive process-level live migration and back migration in HPC environments. Journal of Parallel and Distributed Computing, 72(2):254–267, 2012.

[ZCC07] Zima, Hans; Chamberlain, B. L.; Callahan, D.: Parallel Programmability and the Chapel Language. International Journal on HPC Applications, Special Issue on High Productivity Languages and Models, 21(3):291–312, 2007.

[ZMea15] Zhou, Huan; Mhedheb, Yousri; et. al.: DART-MPI: An MPI-based Implementation of a PGAS Runtime System. CoRR, abs/1507.01773, 2015.

[ZSK04] Zheng, Gengbin; Shi, Lixia; Kalé, Laxmikant V: FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In: Cluster Computing, 2004 IEEE International Conference on. IEEE, pp. 93–103, 2004.