Enabling Application-Integrated Proactive Fault Tolerance

Dai YANG, Josef WEIDENDORFER, Carsten TRINITIS, Tilman KÜSTNER, Sibylle ZIEGLER

TU München, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Boltzmannstr. 3, 85748 Garching, Germany
LMU München, Klinik und Poliklinik für Nuklearmedizin, Marchioninistraße 15, 81377 München, Germany

Abstract. Exascale computing is the next major milestone for the HPC community. Due to a steadily increasing probability of failures, current applications must be made malleable to be able to cope with dynamic resource changes. In this paper, we show first results with LAIK, a lightweight library for dynamically re-distributable application data. This allows to free compute nodes from workload before a predicted failure. For a real-world application, we show that LAIK adds negligible overhead. In addition, we show the effect of different re-distribution strategies.

Keywords. High Performance Computing, Parallel Programming Models, Application-Integrated Fault Tolerance, Data Distribution

1. Introduction

The performance increase of High Performance Computing (HPC) systems traditionally relies both on improvements of components as well as increased system size, with millions of compute units available nowadays. This results in increased importance of fault tolerance mechanisms, as the probability of single node failures becomes higher for applications running on larger systems. For future systems, Defense Advance Research Projects Agency (DARPA) expects that "traditional" fault tolerance technologies such as Checkpoint & Restart will require extensive amounts of resources, which is in clear contradiction to expected high efficiency [2]. An application-transparent approach for increased reliability is to provide fault tolerant environments using virtual machines [8] in combination with process-level live migration [10,16]. However, this often requires a significant amount of resources itself. Another option is to use fault tolerance techniques which cooperate with applications, e.g. by requiring them to dynamically adapt to resource change requests from the outside. This results in reduced overhead and better scalability.

In this paper we present a prototype of a library to help developers make their applications more dynamic, called LAIK (Lightweight Application-Integrated data distribution for parallel workers). It is based on our proposal presented in [17]. Given an “owner-computes” style, by taking over control of partitioning of data containers, LAIK can control the workload for compute units. This enables

1Preprint at: https://mediatum.ub.tum.de/1375185
an external proactive fault tolerance mechanism, which can predict failures of components, to instruct LAIK to free and remove corresponding units from an application. Deliberately, LAIK just provides this limited functionality; it can be used along with MPI or OpenMP, enabling incremental porting and composability. The source code of LAIK is available as open-source on Github\(^2\).

First, we give some related work, and we shortly present the LAIK API. We describe our current prototype with the implemented proactive fault tolerance features. A real-world medical imaging application is used as example, representing an important class of HPC applications. We describe the steps required for a port to LAIK. We show that LAIK has negligible overhead, and we compare different dynamic re-distribution strategies.

2. Related Work

The currently most widely used de-facto parallel programming model for legacy HPC applications is MPI [3], 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 as new programming languages, e.g. Chapel [19] and X10 [12], or as libraries such as Global Array Toolkit (GAT) [9], GASPI [1], and DASH [4,5]. 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. DASH’s communication is built on top of DART [18] - a run-time system which provides abstraction of different communication libraries. All of the abovementioned approaches provide application programmers a full interface for all communication needs of an application. In contrast to that, LAIK only provides one specific functionality, using a communication backend that cooperates with the application code. Other approaches, such as Legion [15], a region based programming system, or Charm++ [6], a C++ library encapsulating workload as so called chares, handles partitioning in a manner similar to LAIK. However, in contrast to LAIK, both come with their own runtimes and require significant effort to port existing applications.

3. The LAIK Library

Most HPC applications have an iterative structure and expose data parallelism. The same operations will be applied to a set of data items, multiple times in a row.

\(^2\)https://github.com/envelope-project/laik
int main(int argc, char* argv[])
{
    Laik_Instance* inst = laik_init_mpi(&argc, &argv);
    Laik_Data* a = laik_alloc_1d(laik_world(inst), 8, 1000000);
    laik_switchto(a, laik_Blockpartitioner, LAIK_DF_CopyOut);
    while (1) { // do iterations
        laik_map(a, (void**)&base, &count); // map to memory
        ...
        laik_allow_repartitioning(inst);
        if (laik_myid(laik_world(inst)) < 0) break; // removed?
    }
    laik_finalize(inst);
}

Figure 1. LAIK example with distributed data in iterative code allowing re-partitioning.

Typically, the data is distributed among the compute units which do computations on the data which is locally available according to the “owner computes” rule. Thus, the partitioning of data also decides on how much each compute unit has to do. The programmer is responsible to implement useful partitioning, taking the number and type of compute units allocated at runtime into account, as well as the knowledge about how much computational workload results from a given amount of data given to a compute unit.

To enable applications to react to external requests to withdraw themselves from given compute units, LAIK requires the developer to make the data distribution mechanism explicit, enabling it to take over control of partitioning and to trigger dynamic redistribution on demand. The programmer specifies where re-distribution is allowed, e.g. whenever a new iteration starts. Compute units with empty partitions are no longer required and can be shut down. LAIK uses the SPMD model (similar to MPI). It defines a C API for easy integration with existing C/C++ and Fortran code. An example is given in Fig. 1. With the LAIK initialization, the type of communication backend used by LAIK is specified (here MPI). The programmer defines mappings between application data to be distributed and abstract dense index spaces. In the example, we ask LAIK to allocate 1 million entries of 8 bytes, mapped implicitly to a 1d index space. The index space and coupled data is partitioned using the built-in block partitioner which distributes the data equally among all tasks. Access to data controlled by LAIK is done in access phases. For each access phase, the programmer specifies a partitioning object to use as well as whether data needs to be preserved among neighbor phases. In the example, we enter an access phase starting with uninitialized data, to be preserved when switching to the next phase (using a CopyOut data flow request). To actually access data managed by LAIK, it explicitly has to be mapped to memory. Access permission ends with the end of the current access phase. Re-partitioning may be allowed by the developer at any time (as in the example), but terminates the current access phase. Then, LAIK checks for external requests which may ask to set partitions of tasks to be freed to zero. A previously defined partitioner (in the example, the block partitioner) is used to re-distribute indexes among remaining tasks. If data should be preserved (as in
the example), such a change implicitly results in communication, as the locality of indexes in the partitioned space changes and thus, the location of coupled data, too. Afterwards, tasks can check whether they got freed, and may terminate early (as shown in Fig. 1).

Actually, for each data structure in an application, LAIK can be used in two ways: either, the application manages the data itself. Then LAIK provides the information about index slices assigned to a task, as well as which slices in the index space need to be sent to or received from other tasks when the partitioning changes. This way, complex mappings from the dense index spaces known to LAIK to application-specific sparse or irregular data structures are possible. The other way of using LAIK is to let LAIK manage memory allocation for data, as shown in the example above. A computational kernel within an application may read and write data from various data structures. Typically, there is a primary partitioning which decides about how the data structure is distributed where results are written to. The partitioning of other data structures often need to depend on and can be derived from this primary partitioning, to allow computations to be executed by only accessing locally available data. LAIK allows to define such derivative partitionings, resulting in synchronized changes to them whenever the primary partitioning changes. It is important to allow applications to provide their own partitioning algorithms to let programmers include their knowledge about how much computational work is required for one index of the index space. To reduce communication on re-partitioning, a partitioning algorithm can have access to previous partition boundaries, allowing for incremental algorithms. Our LAIK prototype provides a set of useful built-in algorithms.

LAIK is designed to allow incremental porting of existing applications. The partitioning of different data structures can be moved step-by-step to become controlled by LAIK. Furthermore, LAIK is expected to be composable with existing parallel programming models such as MPI or OpenMP, by documenting the API calls which work locally vs. need global synchronization. For flexibility, LAIK uses a defined interface to trigger data transfers among tasks, implemented by communication backends (such as using MPI). The creation and any change of a partitioning of an index space requires global synchronization, as partitions of all involved tasks may change. This also holds true whenever the program allows for eventual re-partitioning, even if no actual change is done, as all tasks need to agree on what to do. However, switching between known partitionings only requires communication as actually needed. This should be the regular case for applications, resulting in no further overhead than really needed.

**LAIK Prototype**

The current prototype implements basic functionality with an MPI backend. Next to the block partitioner, to allow a master task to broadcast data to all others, we provide predefined All and Master partitionings. Requests for preserving data at the same indexes by all tasks triggers reductions to be executed when leaving the current access phase. The current prototype supports 1d partitioning on 1–3 dimensional index spaces and data. For switching between partitionings, we first calculate transition actions on index sets: (1) indexes which remain local; (2) indexes to be sent to other tasks; (3) indexes to be received from another task;
Figure 2. (Left) Two different repartitioning methods: (1) Data on failing node $N1$ is equally distributed across remaining nodes. (2) Upon node failure, a global redistribution of data is initiated. (Right) Agent Interface of LAIK.

(4) indexes to be initialized with the neutral element of a reduction operation; and (5) a reduction to calculate for given indexes. For coupled data managed by LAIK, actions 1 and 4 are processed internally, and actions 2/3/5 are passed to the communication backend. Our MPI backend currently uses synchronous MPI operations.

The major benefit achieved by using LAIK is the ability to dynamically redistribute application data upon request. We provide API functions for the application to trigger repartitioning itself, or my means of an external source using an agent interface. Currently, we provide two different repartitioning algorithms as shown in Fig. 2 (left). The first one is incremental, i.e. using the previous partitioning. Indexes of partitions from tasks running on nodes expected to fail are redistributed to remaining tasks. This should reduce the amount of data to be communicated, but requires the application to be able to cope with multiple separate slices of data, i.e. the global indexes of local data is not continuous. In the second algorithm, a global redistribution of data is calculated. This variant may require more effort in communication, but each task receives a continuous piece of data for calculation.

To obtain data about system health and potential spare nodes, LAIK exposes an agent interface. This supports the communication with external agents, e.g. providing data on upcoming node failure events. Agents can be health reporting systems (e.g. Splunk) or workload schedulers (e.g. SLURM). LAIK supports events to be passed to it within every task, but at one point in time, only one task on a node actually should be used for passing events for minimal overhead, with one agent daemon running per node. Fig. 2 (right) shows a schematic composition of the agent interface of LAIK. The agent interface supports agent adapters in the form of both statically and dynamically linked libraries. Dynamic libraries are loaded upon request by the application, while static ones are part of LAIK. Multiple agents can be attached to a LAIK instance.

4. Application Example: Image Reconstruction via MLEM

The Maximum Likelihood Expectation Maximization (MLEM) algorithm [13] is used to reconstruct 3D images from data obtained by positron emission tomography (PET) scanners. It is an iterative algorithm that mainly consists of two sparse matrix-vector multiplications (SpMV) per iteration. Positron emission to-
mography is a functional imaging technique in nuclear medicine. A radionuclide is injected into the subject’s body, which undergoes beta decay. After traveling a very short distance (typically less than 1 mm), the positron interacts with an electron. The result of this annihilation event, a pair of high-energy photons traveling in opposite directions, are detected by a ring of scintillator crystals positioned around the subject. Reconstructing a 3D image from the scanner readout (a list of detected events) is an inverse problem. While multiple (accelerated) algorithms exist, MLEM is commonly seen as the one providing the highest quality. The algorithm requires two inputs: the readout and the system matrix. The system matrix describes all scanner properties in a concise way, for example the spatial arrangement of the detectors and the physical effects during image acquisition. The matrix is sparse, as each pair of detectors can only detect events coming from a section (a tube) of the full field of view of the scanner. The system matrix can be obtained in different ways: direct measurement using a probe, analytical models like the detector response function (DRF) model [14], or Monte Carlo simulation of the imaging process. For our application example, we used a system matrix describing the small animal PET scanner Madpet-II [11], generated by the DRF model. The matrix has around $1.6 \times 10^9$ non-zero elements and takes around 12.8 GB of memory stored in compressed sparse row (CSR) format.

The MLEM algorithm starts with an estimate of the image, which is improved with each iteration. Per iteration, the algorithm performs the following steps:

1. Calculate the scanner readout which would result from the current image estimate. This is called a forward projection and corresponds to an SpMV of the system matrix $A$ and the image vector.
2. Calculate a correction vector by correlating the estimated readout with the actual readout. This is an element-wise vector operation.
3. Transfer the correction factor into the image domain. This is called back-projection and corresponds to an SpMV of the transposed system matrix $A^T$ and the correction vector.
4. Apply the correction factor and a normalization vector to the image results to come up with a new estimate. This is again a vector operation.

Although the transposed system matrix is required in the back-projection step, we do not actually have to store the transposed matrix, as $y = A^T x \Leftrightarrow y^T = x^T A$.

All calculations are performed in single precision. The two SpMV operations dominate the vector operations, so that the time per iteration is given by the speed of the SpMV operations. We expect that an optimally implemented SpMV operation is limited by the memory bandwidth of the system, which is used to read the system matrix. We built our implementation upon an MPI-parallelized version of MLEM developed earlier [7]. The SpMV operations are parallelized by cutting the sparse matrix into groups of rows containing approximately the same number of non-zero elements. Each group is then assigned to an MPI task and each MPI process only loads its part of the system matrix from disk. Due to this approach, MPI communication comprises 2 all to all reductions per iteration (regardless of the partitioning scheme).
Porting MLEM

As preparation for porting the MPI version of MLEM to LAIK, the code needed to cope with partitions assigned to tasks which consist of multiple separate slices, as created by the incremental partitioner. To this end, the sparse matrix class reading the data from a file had to be changed. Around computational kernels of MLEM, another loop over the slices had to be added. The old code did file-mapping of the sparse matrix data. For reproducible runs on NUMA architectures, we triggered first-touch memory allocation by explicit copy operations. For the actual port to LAIK, we used it to control the partitioning of the rows of the sparse matrix, using a blocked partitioner with support for weighted indexes to equally distribute non-zero elements instead of equal number of rows. The data of the sparse matrix remains in control of the application. However, LAIK data containers were used to hold the normalization, correction, and image vector, using All partitioning and reduction operations, as each task needs its own complete copy of these vectors. The port itself (apart from the preparations) required an effort of half a day for our programmer, who is not involved in the development of interfaces of LAIK. The modified MLEM code is available on Github.

5. Evaluation

As a testbed, we have chosen the CoolMUC2 cluster at Leibniz Supercomputing Centre. The compute node of CoolMUC2 features a two socket configuration with two Intel Xeon E5-2697v3 processors (“Haswell”, 35MB L3, 14 Cores) configured in “cluster-on-die” mode (two NUMA domains per chip) with HyperTreading disabled and 64GiB of RAM. FDR14 InfiniBand interconnect is installed as fabric interconnect. The NAS background storage is connected via a 2GiB/s connection. The software toolchain is Intel MPI with GCC 5, and binaries are compiled using “-O3”. Both our LAIK-MLEM and the native MPI MLEM is run using 2 to
Continuous Incremental

Figure 4. (Left) Comparison of LAIK-MLEM runtime before, during and after repartitioning with different repartitioners; (Right) Iteration runtime for 14 MPI tasks before and after repartitioning using incremental repartitioner

28 MPI tasks, pinned equally among the four NUMA domains of a node (using `LMPI_PIN_DOMAIN=numa`), and each of them with 10 MLEM iterations. Four independent measurements are captured for each task/binary combination. All MPI tasks are executed on the same node. The initial time of loading the sparse matrix from remote disk is not considered in any measurements.

As shown in Fig. 3, LAIK does not produce much overhead over the native MPI version of MLEM. Hence, a speedup of up to 12x can be achieved. This is also the case for the MPI only version of our MLEM implementation. Fig. 3 shows that the speedup flattens and oscillates with a higher number of MPI tasks. We checked that this is not due to load imbalance, but due to MLEM becoming memory bound with higher core counts. Furthermore, the architecture of our testbed consists of a dual socket system with two NUMA domains on each chip, resulting in four NUMA domains. This results in worse performance if the number of cores is not divisible by four. We conclude that the overhead of LAIK for MLEM is low.

We now compare different repartitioning algorithms: The first one is the incremental repartitioner, as stated in Fig. 2 (1), which appends the workload of leaving MPI tasks to the remaining ones. The second one is our continuous repartitioner, which initiates a new global partitioning once an MPI task is leaving. In this experiment, we shrink the number of working MPI tasks by one after the sixth iteration of MLEM execution. The results can be seen in Fig. 4 and Fig. 5. We see that the incremental repartitioner outperforms the continuous one by a factor of two. This is because the MLEM code has to load new parts of the sparse matrix in order to do the required computations. In the incremental case, a significantly lower amount of data has to be loaded, since the difference to the original loaded matrix regions is smaller. With the continuous repartitioner, the higher amount of data to be loaded results in increased time consumption after repartitioning. This performance gain becomes even clearer in Fig. 5 (left): Both the execution
time before and after repartitioning remain the same, while the time for the first iteration after repartitioning differs. In addition, one can see that the incremental partitioner scales better than the continuous one in case of MLEM. Fig. 5 (right) shows that the runtime per iteration increases only slightly after repartitioning, indicating that LAIK has redistributed the data equally across all remaining MPI tasks. Both partitioners are useful in different scenarios: some applications are less elastic and cannot simply be changed to accept multiple slices within each iteration. In this case, some applications need to be adapted accordingly while other applications (such as MLEM) may benefit from the incremental approach.

6. Conclusions and Future Work

In this paper we presented LAIK, a library to add elasticity to parallel codes by using partitioned index spaces as abstraction for workload. This allows to react to outside requests to free compute resources, enabling application-integrated proactive fault tolerance (e.g. based on predictions of future component failures).

We described how to port a real-world application to LAIK. While MLEM has a simple communication pattern, it already represents an important class of HPC applications. We showed that our prototype does not add not much overhead, and we compared two implemented redistribution strategies. To support automatic (re)partitioning of MLEM’s sparse matrix in CSR format, it proved useful to have LAIK only optionally control memory allocation as well as to support application-specific partitioning algorithms. LAIK’s interaction with existing parallel programming models enabled a straightforward step-wise porting of MLEM.

As future work, we need more experience with porting applications with more complex communication patterns, such as neighbor exchange in 2d or 3d stencil computations. We plan to implement backends supporting heterogenous systems, allowing hot-plug of components while applications are running, as well as combining multiple backends via nested LAIK instances. Regarding spontaneous failures, we want to implement automatic, LAIK controlled local checkpointing.
to neighbor nodes, which should be possible if all important application data is managed by LAIK. However, e.g. for MPI, it may need an implementation of MPI that supports fault tolerance features itself.

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


