Supercomputing Frontiers and Innovations

2014, Vol. 1, No. 1.

Scope

- *Parallel and distributed computing technologies.* Parallel programming languages and libraries. Methods, algorithms and tools for analysis, debugging and optimization of supercomputing applications. Co-design concepts.
- *Next-generation supercomputer architectures.* Advanced supercomputer architectures. Reconfigurable and hybrid supercomputing systems. Accelerator based supercomputing systems. Modeling and development of new high-performance computing hardware.
- *Supercomputing Applications.* Solving of computationally intensive problems using supercomputers in computational mathematics, computational physics, chemistry, hydro-gas-dynamics and heat transfer, nonlinear transient problems in mechanics, bioinformatics, engineering, nanotechnology, climate, ecology, cryptography, and other prospective areas.
- *Visualization.* Parallel methods and algorithms of visualizing of computational experiments.
- *Management, administration and monitoring of supercomputing systems.* Methods, algorithms and tools of management, technical support and monitoring of highly parallel supercomputing systems.
- *Parallel System Software and Tools.* Tools, performance evaluation, development tools, run-time systems, libraries.
- *Parallel database systems.* Development of parallel database management systems for multiprocessor (multicore) computing systems. Methods and algorithms of parallel database processing. High performance data mining.
- *Supercomputing Education.* Practice and experience of supercomputing education. Innovative approaches on teaching of HPC and parallel computing technologies.

Editorial Board

Contents

DOI: 10.14529/jsfi140101

Toward Exascale Resilience: 2014 Update

Franck Cappello,[1][2] *Al Geist,*[3] *William Gropp,*[2] *Sanjay Kale,*[2] *Bill Kramer,*[2] *Marc Snir*[1][2]

Resilience is a major roadblock for HPC executions on future exascale systems. These systems will typically gather millions of CPU cores running up to a billion threads. Projections from current large systems and technology evolution predict errors will happen in exascale systems many times per day. These errors will propagate and generate various kinds of malfunctions, from simple process crashes to result corruptions.

The past five years have seen extraordinary technical progress in many domains related to exascale resilience. Several technical options, initially considered inapplicable or unrealistic in the HPC context, have demonstrated surprising successes. Despite this progress, the exascale resilience problem is not solved, and the community is still facing the difficult challenge of ensuring that exascale applications complete and generate correct results while running on unstable systems. Since 2009, many workshops, studies, and reports have improved the definition of the resilience problem and provided refined recommendations. Some projections made during the previous decades and some priorities established from these projections need to be revised. This paper surveys what the community has learned in the past five years and summarizes the research problems still considered critical by the HPC community.

Keywords: Exascale, Resilience, Fault-tolerance techniques.

1. Introduction

We first briefly introduce the terminology that is used in this paper, following the taxonomy of Aviženis and others [3, 90]. System *faults* are causes of *errors*, which manifest themselves as incorrect system states. Errors may lead to *failures*, where the system provides an incorrect service (e.g., crashes or provides wrong answers). We deal with faults by predicting, preventing, removing, or tolerating them. *Fault tolerance* is achieved by detecting errors and notifying about errors and by recovering, or compensating for errors, for example, by using redundancy. *Error recovery* includes *rollback*, where the system is returned to a previous correct state (e.g., a checkpoint) and *rollforward*, where a new correct state is created. Faults can occur at all levels of the stack: facilities, hardware, system/runtime software, and application software. Fault tolerance similarly can involve a combination of hardware, system, and application software.

This paper deals with *resilience* for exascale platforms: the techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults. The lack of appropriate resilience solutions is expected to be a major problem at exascale: We discuss in Section 2 the many reasons errors are likely to be much more frequent in exascale systems. The current solutions used on petascale platforms, discussed in Section 3, may not scale up to exascale. We risk having systems that can perform quintillions of operations each second but never stay up long enough to progress in their computations, or produce results that cannot be trusted.

The problem of designing reliable computers out of unreliable components is as old as computing—as old as Babbage's analytical engine [15]. Frequent failures were a major problem in the earliest computers: ENIAC had an mean time to failure of two days [93]. Major advances in this area occurred in the 1950s and 1960s, for example, in the context of digital telephone

[1]Argonne National Laboratory
[2]University of Illinois at Urbana Champaign
[3]Oak Ridge National Laboratory

switches [35] and mainframes [91]. More recently, NASA examined the use of COTS (non-rad-hardened) processors for space missions, which requires tolerance of hardware errors [70]. Resilience for HPC is a harder problem, however, since it involves large systems performing tightly coupled computations: an error at one node can propagate to all the other nodes in microseconds.

Five years ago we published a survey of the state of the art on resilience for exascale platforms [19]. Since then, extraordinary technical progress has been made in many domains related to exascale resilience. Several technical options, initially considered inapplicable or unrealistic in the HPC context, have demonstrated surprising success. Despite this progress, the exascale resilience problem is not solved, and the community still faces the difficult challenge of ensuring that exascale applications complete and generate correct results while running on unstable systems. Since 2009, many workshops, studies, and reports have improved the definition of the resilience problem and provided refined recommendations [18, 19, 32, 38, 39, 58, 69, 90]. Some projections made during the previous decades and some priorities established from these projections now need to be revised. This paper surveys what the community has learned in the past five years (Section 4) and summarizes the research problems still considered critical by the HPC community (Section 5).

2. The Exascale Resilience Problem

Future exascale systems are expected to exhibit much higher fault rates than current systems do, for various reasons relating to both hardware and software.

2.1. Hardware Failures

Hardware faults are expected to be more frequent: since clock rates are not increasing, performance increases require a commensurate increase in the number of components. With everything else being equal, a system 1,000 times more powerful will have at least 1,000 times more components and will fail 1,000 times more frequently.

However, everything else is not equal: smaller transistors are more error prone. One major cause for transient hardware errors is cosmic radiation. High-energy neutrons occasionally interact with the silicon die, creating a secondary cascade of charged particles. These can create current pulses that change values stored in DRAM or values produced by combinatorial logic. Smaller circuits are more easily upset because they carry smaller charges. Furthermore, multiple upsets become more likely. Smaller feature sizes also result in larger manufacturing variances, hence larger variances in the properties of transistors, which can result in occasional incorrect or inconsistent behavior. Smaller transistors and wires will also age more rapidly and more unevenly so that permanent failures will become more frequent. Energy consumption is another major bottleneck for exascale. Subthreshold logic significantly reduces current leakage but also increases the probability of faults.

Vendors can compensate for the increase in fault rates with various techniques. For example, for regular memory arrays, one can use more powerful error correction codes and interleave coding blocks in order to reduce the likelihood of multiple bit errors in the same block. Buses are usually protected by using parity codes for error detection and by retries; it is relatively easy to use more powerful codes. Logic units that transform values can be protected by adding redundancy in the circuits. Researchers have estimated that an increase in the frequency of errors can be avoided at the expense of 20% more circuits and more energy consumption [90].

Whether such solutions will be pursued is unclear, however: the IC market is driven by mobile devices that are cost and energy sensitive and do not require high reliability levels. Most cloud applications are also cost sensitive but can tolerate higher error rates for individual components. The small market of high-end servers that require high reliability can be served by more costly solutions such as duplication or triplication of the transactions. This market is not growing in size or in the size of the systems used. Thus, if exascale systems will be built out of commodity components aimed at large markets, they are likely to have more frequent hardware errors that are not masked or not detected by hardware or software.

2.2. Software Failures

As hardware becomes more complex (heterogeneous cores, deep memory hierarchies, complex topologies, etc.), system software will become more complex and hence more error-prone. Failure and energy management also add complexity. Similarly, the increase use of open source layers means less coordinated design in software, which will increase the potential for software errors. In addition, the larger scale will add complexities as more services need to be decentralized, and complex failure modes that are rare and ignored today will become more prevalent.

Application codes are also becoming more complex. Multiphysics and multiscale codes couple an increasingly large number of distinct modules. Data assimilation, simulation, and analysis are coupled into increasingly complex workflows. Furthermore, the need to reduce communication, allow asynchrony, and tolerate failures results in more complex algorithms. Like system software, these more complex algorithms and application codes are more error-prone.

Researchers have predicted that large parallel jobs may fail as frequently as once every 30 minutes on exascale platforms [90]. Such failure rates will require new error-handling techniques. Furthermore, silent hardware errors may occur, requiring new error-detection techniques in (system and/or application) software.

3. Lessons learned from Petascale

Current petascale systems have multiple component failures each day. For example, a study of the Blue Waters system [1] during its 2013 preproduction period showed that, across all categories of events, an event that required remedial repair action occurred on average every 4.2 hours and that systemwide events occurred approximately every 160 hours [34]. The reported rates included events that failed transparently to applications or were proactively detected before application use but required remedial actions. Events with performance inconsistency and long failover times were reported as errors even if applications and the failover operation eventually completed successfully. Hence the events that were actually distuptive to applications were about half as frequent. In the first year of Blue Waters' full production, the rates improved by approximately a factor of 2. Similar rates are reported on other systems. A significant portion of failures are due to software—in particular, file and resource management systems. Furthermore, software errors take longer than hardware errors to recover from and account for the majority of downtime.

Software errors could be avoided by more rigorous testing. Testing HPC software at scale is hard and expensive, however. Since very large systems are one of a kind, each with a unique configuration, they are expensive and require unique machine rooms. Usually, vendors are not able to deploy and test a full system ahead of installation at the customer site, and customers cannot afford long testing periods ahead of actual use. Some scalability bugs will occur only intermittently at full scale. Subtle, complex interactions of many components may take a long time to occur. Many of the software products deployed on a large platform are produced by small companies or by teams of researchers that have limited resources for thorough testing on multiple large platforms.

The standard error-handling method on current platforms is periodic application checkpoint. If a job fails, it is restarted from its last checkpoint. Checkpoint and restart logic is part of the application code. A user checkpoint can be much smaller than a system checkpoint that would save all the application's memory; the checkpoint information can be also used as the output from a simulation; and a user checkpoint can be used to continue a computation on another system.

For single-level checkpointing, the checkpoint interval can be computed by using the formula developed by Young [97] or Daly [29]. Young's formula is particularly simple: $Interval = \sqrt{2 \times MTTI \times checkpt}$, where $MTTI$ is the mean time to interrupt and $checkpt$ is the checkpoint time. This formula approximates the optimum checkpoint interval, assuming a memoryless model for failures. For large jobs, this typically implies a checkpoint about every hour or multiple times per hour.

Root cause analysis of failures, especially software failures, is hard. Error logs may report which component signaled a failure, but failures can be due to a complex chain of events. For illustration, consider the actual case of a system crash due to a fan failure: The failure of one fan caused multiple hardware components to stop working; the cascade of errors reported from all these components overwhelmed the monitoring network and crashed the monitoring system; this crash, in turn, caused the entire system to crash [57]. The volume of system activity and event processing for large-scale systems is on the order of several tens of gigabytes and hundreds of millions of events per day during normal activity.

Current systems do not have an integrated approach to fault tolerance: the different subsystems (hardware, parallel environment software, parallel file system) have their own mechanisms for error detection, notification, recovery, and logging. Also, current systems do not have good error isolation mechanisms. For example, the failure of any component in a parallel job results in the failure of the entire job; the failure of the hardware monitoring infrastructure may impact the entire machine.

We note that all reports on failure rates focus on hardware and system software and ignore application software. From the viewpoint of the supercomputing center, a failure due to a bug in the application code is not a failure. Such failures are not reported, and no statistics of their frequency exist. Also, supercomputing centers have no information on errors that were not detected by hardware or system software. An incorrect result may be detected by the user once the run is complete, but it will be practically impossible to correctly attribute such an error to a root cause. Anecdotal evidence suggests that undetected hardware errors do occur on current systems [57].

4. Progress toward Exascale Resilience

We present in this section the most significant research progress in resilience since 2009.

4.1. System Software Approaches

We first discuss the progress in handling fail-top errors by checkpointing. We then describe other approaches, including forward recovery, replication, failure prediction, and silent data corruption mitigation.

4.1.1. Checkpointing

To tolerate fail-stop errors (node, OS or process crash, network disconnections, etc.), all current production-quality technologies rely on the classic rollback recovery approach using checkpoint restart (application-level or system-level). Solutions focus on applications using MPI and Charm++. The most popular approach is application-level checkpointing, where the programmer defines the state that needs to be stored in order to guarantee a correct recovery in case of failure. The programmer adds some specific functions in the application to save essential state and restore from this state in case of failure. One of the drawbacks of checkpointing in general, and of application-level checkpointing in particular, is the nonoptimality of checkpoint intervals. Another drawback is the burden placed on the I/O infrastructure, since the checkpoint I/O may actually interfere with communication and I/O of other applications. The impact of suboptimal checkpoint intervals is investigated in [68]. System-level checkpoint can also be used in production, with technologies such as BLCR [63]. Some recent research in this domain focuses on the integration of incremental checkpointing in BLCR. In the past few years, research teams have developed mechanisms to checkpoint accelerators [81, 83].

The norm in 2009 was to store the application state on remote storage, generally a parallel file system, through I/O nodes. Checkpoint time was significant (often 15–30 minutes), because of the limited bandwidth of the parallel file system. When checkpoint time is close to the MTBF, the system spends all its time checkpointing and restarting, with little forward progress. Since the MTJI may be an hour or less on exascale platforms, new techniques are needed in order to reduce checkpoint time.

One way of achieving such a reduction is to reduce the checkpoint size. Techniques such as memory exclusion, data compression, and compiler analysis to detect dead variables have been proposed. More recently some researchers have explored hybrid checkpointing [94], data aggregation [67], incremental checkpointing in the context of OpenMP [17], and data deduplication with the hope that processes involved in parallel HPC executions have enough similarities in memory [6] to reduce the size of the data sets necessary to be saved. However, what we mentioned five years ago is still valid: Programmers are in the best position to know the critical data of their application but they cannot use adaptive data protection other than by doing it manually. Annotations about ways to protect or check key data, computations, or communications are still a relevant direction.

Another way of reducing checkpoint time is to reduce the usage of disks for checkpoint storage. In-memory checkpointing has been demonstrated to be fast and scalable [99]. In addition, multilevel checkpointing technologies such as FTI [4] and SCR [78] are increasingly popular. Multilevel checkpointing involves combining several storage technologies (multiple levels) to store the checkpoint, each level offering specific overhead and reliability trade-offs. The checkpoint is first stored in the highest performance storage technology, which generally is the local memory of a local SSD. This level supports process failure but not node failure. The second level stores the checkpoint in remote memory of remote SSD. This second level supports a single-node failure.

The third level corresponds to the encoding the checkpoints in blocks and in distributing the blocks in multiple nodes. This approach supports multinode failures. Generally, the last level of checkpointing is still the parallel file system. This last level supports catastrophic failures such as full system outage. Charm++ provides similar multilevel checkpointing mechanisms [98]. The checkpoint period can be defined in different ways. Checkpoints also can be moved between levels in various ways, for example, by using a dedicated thread [4] or agents running on additional nodes [87]). A new semi-blocking checkpoint protocol leverages multiple levels of checkpoint to decrease checkpoint time [80]. A recent result computes the optimal combination of checkpoint levels and the optimal checkpoint intervals for all levels given the checkpoint size, the performance of each level, and the failure distributions for each level [82]. Other recent research concerns the understanding of the energy footprint of multilevel checkpointing and the exploration of trade-offs between energy optimization and performance overhead. The progress in multilevel checkpointing is outstanding, and some initiatives are looking at the definition of a standard interface.

By offering fast checkpointing at the first level, multilevel checkpointing also offers the opportunity to increase the checkpoint frequency and checkpoint at a smaller granularity. For example instead of checkpointing at the outermost loop of an iterative method, multilevel checkpointing could be used in some inner loops of the iterative method. Reducing the granularity of checkpointing is also the objective of task-based fault tolerance. The principle is to consider the application code as a graph of tasks and checkpoint the input parameter of each task on local storage with a notion of transaction: either a task is completed successfully and its results are committed in memory, or the task is restarted from its checkpointed input parameters. This approach is particularly relevant for future nonvolatile memory on computing nodes. The nonvolatile memory would store the input parameters checkpoints and the committed results. The OMPsS environment offers a first prototype of this approach [92]. One of the key questions is how to ensure a consistent restart in case of a node failure. Solutions need to consider how to store the execution graph and the local memory content in order to tolerate a diversity of failure scenarios.

The past five years have also seen excellent advances in checkpointing protocols. Classic checkpointing protocols are used to capture the state of distributed executions so that, after a failure, the distributed executions (or a part of them) restart and produce a correct execution [44]. In the HPC domain, classic checkpointing protocols are rarely used in production environments because most of the applications use application-level checkpointing that implicitly coordinates the capture of the distributed state and guarantees the correctness of the execution after restart. Without additional support, however, application-level checkpointing imposes global restart even if a single process fails. Message-logging fault-tolerant protocols offer a solution to avoid global restart and to restart only the failed processes. By logging the messages during the execution and replaying messages during recovery, they allow the local state reconstruction of the failed processes (partial restart). The main limitation is the necessity to log all messages of the execution. Several novel fault tolerance protocols overcome this limitation by reducing significantly the number of messages to log. They fall into the class of hierarchical fault tolerance protocols, forming clusters of processes and using coordinated checkpointing inside clusters and and message logging between clusters [61, 77, 84]. Such protocols need to manage causal dependencies between processes in order to ensure correct recovery. Multiple approaches have been proposed for reducing the overhead of storing causal dependency information [11, 20].

Partial restart opens opportunities for accelerated recovery. Charm++ and AMPI accelerate recovery by redistributing recovering processes on nodes that are not restarting [22]. Message logging can by itself accelerate recovery because messages needed by recovering processes are immediately available and messages to be sent to nonrestarting processes are just canceled [84]. These two aspects reduce the communication time during recovery. A recent advance in hierarchical fault tolerance protocols is the formulation and solving of the optimal checkpoint interval in this context [9]. Partial restart also opens the opportunity to schedule other jobs (than the recovering one) on resources running nonrestarting processes during the recovery of the failed processes. While this principle does not improve the execution performance for the job affected by the failure, it significantly improved the throughput of the platform that execute more jobs in a given amount of time compared with restarting all the processes of a job when a failure occurs [12]. This new result demonstrates another benefit of message-logging protocols that was not known before.

4.1.2. Forward Recovery

In some cases, the application can handle the error and execute some actions to terminate cleanly (checkpoint on failure [7]) or follow some specific recovery procedure without relying on classic rollback recovery. Such applications use some form of algorithmic fault tolerance. The application needs to be notified of the error and runs forward recovery steps that may involve access to past or remote data to correct (sometimes partially) or compensate the error and its effect, depending on the latency of the error detection.

A prerequisite for rollforward recovery is that some application processes and the runtime environment stay alive. While standard MPI does not preclude an application continuing after a fault [60], the MPI standard does not provide any specification of the behavior of an MPI application after a fault. For that purpose, researchers have developed several resilient MPI designs and implementations. The FT-MPI (fault-tolerant MPI) library [50, 66] was a first implementation of that approach. As the latest development, ULFM [8] allows the application to get notifications of errors and to use specific functions to reorganize the execution for forward recovery. Standardization of resilient MPI is complex; and despite several attempts, the MPI Forum has not reached a consensus on the principles of a resilient MPI. The GVR [100] system developed at the University of Chicago also provides mechanisms for application-specified forward error recovery. GVR design features two key concepts: (1) a global view of distributed arrays to processes and (2) versioning of these distributed arrays for resilience. APIs allow navigation of multiple versions for flexible recovery. Several applications have been tested with GVR.

4.1.3. Replication

Understanding of replication has progressed significantly in the past five years. Several teams have developed MPI and Charm++ prototypes offering process-level replication. Process-level replication of parallel executions is more reliable than replicated parallel executions under the assumption that the replication scheme itself is reliable. Several challenges need to be solved in order to make replication an attractive approach for resilience in HPC. First, the overhead of replication on the application execution time should be negligible. Second, replication needs to guarantee equivalent state of process replicas,[4] which is not trivial because some MPI operations are not deterministic. A third, more complex challenge is the reduction of the resource and energy cost of replication. By default, replication needs twice the number of resources compared with nonreplicated execution.

One major replication overhead comes from the management of extra messages required for replication. Without specific optimization, when a replicated process sends a message to another replicated process, four communications of that message take place. rMPI addresses this problem by reducing the number of communications between replicas [51]. rMPI and MR-MPI [46] orchestrate non-MPI deterministic operations between process replicas to ensure equivalence of internal state. While rMPI and MR-MPI focus on process replication to address fail-stop errors, RedMPI [53] leverages process replication for detection of silent data corruptions (SDCs). The principle of RedMPI is to compare on the receiver side messages sent by replicated senders. If the message contents differ, a silent data corruption is suspected. RedMPI offers an optimization to avoid sending all messages needed in a pure replication scheme and to avoid comparing the full content of long messages. For each MPI message, replicated senders compute locally a hash of the message content, and only one of the replicated senders actually sends the message and the hash code to replicated receivers. Other replicas of the senders send only the hash code. Receivers then compare locally the hash code received from the replicated senders.

Reducing the resource overhead of process replication is a major challenge. One study [52], however, shows that replication can be more efficient than rollback recovery in some extreme situations where the MTBF of the system is extremely low and the time to checkpoint and restart is high. While recent progress in multilevel checkpointing make these situations unlikely, the results in [52] demonstrate that high rollback recovery overheads can lead to huge resource waste, up to the point where replication becomes a more efficient alternative. MR-MPI explores another way of reducing replication resource overhead by offering partial replication, where only a fraction of the processes are replicated. This approach is relevant when the platform presents some asymmetry in reliability (some resources being more fragile than others) and when this asymmetry can be monitored. Partial replication should be complemented by checkpoint restart to tolerate failures of non replicated processes [42]. Some libraries, for example, the ACR library [79] for MPI and Charm++ programs, cover both hard failures and SDCs from replication.

4.1.4. Failure Prediction

One domain that has made exceptional progress in the past five years is failure prediction. Before 2009, considerable research focused on how to avoid failures and their effects if failures could be predicted. Researchers explored the design and benefits of actions such as proactive migration of checkpointing [47, 74, 95]. The prediction of failures itself, however, was still an open issue. Recent results from the University of Illinois at Urbana-Champaign [54–56] and the Illinois Institute of Technology [101, 102] clearly demonstrate the feasibility of error prediction for different systems: the Blue Waters CRAY system based on AMD processors and NVIDIA GPUs and the Blue Gene system based on IBM proprietary components. Failure prediction techniques have progressed by combining data mining with signal analysis and methods to spot outliers.

[4]Internal state of process replicas do not need to be identical, but external interactions of each process replica should be consistent with a correct execution of the parallel application

Some failures can be predicted with more than 60% accuracy[5] on the Blue Waters system. Further research is needed to extend the results to other subsystems, such as the file system. We are still far from the accuracy needed to switch from pure reactive fault tolerance to truly proactive fault tolerance.

Other advances concern the combination of application-level checkpointing and failure prediction [10]. An important question is how to run the failure predictor on large infrastructures. One approach is to run a failure predictor in each node of a system in order to avoid the scalability limitation of global failure prediction. This approach faces two difficulties: (1) local failure prediction will impose an overhead on the application running on the node, and (2) local failure prediction is less accurate than global failure prediction because the failure predictors have only a local view. These two difficulties are explained in [10]. Another important question is how to compute the optimal interval of preventive checkpoints when a proportion of the failures are predicted [10]. In particular, many formulations of the optimal checkpoint interval problem consider that failures follow an exponential distribution of interarrival times. This approximation may be acceptable if we consider all failures in the system. Is it acceptable for failure that are not predicted correctly and for which preventive checkpointing is needed?

4.1.5. Energy Consumption

A new topic emerged in the community few years ago: energy consumption of fault tolerance mechanisms. The first paper on the topic we are aware of [41] presents a study of the energy footprint of fault tolerance protocols. The important conclusion of the paper is that, at least for clusters, little difference exists between checkpointing protocols. The study also shows that the difference in energy depends mainly on the difference in execution time and only slightly on the difference of power consumption of the operation performed by the different protocols. The reason is that the power consumptions of computing, checkpointing, and message logging are close in clusters.

Some teams [76] developed models for expected run time and energy consumption for global recovery, message logging, and parallel recovery protocols. These models show in an exascale scenario that parallel recovery outperforms coordinated checkpointing protocols since parallel recovery reduces the rework time. Other researchers [2] developed performance and energy models and formulated an optimal checkpointing period considering energy consumption as the objective to optimize.

Another research issue is the energy optimization of checkpoint/restart on local nonvolatile memory [85]. The intersection of resilience and energy consumption is also explored in a recent study [86].

4.1.6. Mitigating Silent Data Corruptions

One of the main challenges that HPC resilience faces at extreme scale and in particular in exascale systems and beyond is the mitigation of silent data corruptions (SDCs). As mentioned in previous sections, the risk of silent data corruptions is increasing. Several studies have explored the impact of SDCs in execution results [16, 43, 73]. These studies show that a majority of SDCs leads to noticeable impacts such as crashes and hangs but that only a small fraction of them

[5] Accuracy here is defined as the prediction recall: the number of correctly predicted failures divided by the number of actual failures

actually corrupt the results. Nevertheless, the likelihood of generating wrong results because of SDC is significant enough to warrant study of mitigation techniques.

An excellent survey of error detection techniques is presented in [71]. A classic way to detect a large proportion of SDCs (but not all) is replicating executions and comparing results. Following this approach, RedMPI [53] offers replication at the MPI process level, and replication at the thread level is studied in [96] by leveraging hardware transactional memory. The first issue with SDC detection by replication is the overhead in resources. A second issue is that, in principle, comparison of results supposes that execution generates identical results, which means obtaining bit-to-bit deterministic results from executions using same input parameters. Applications may not have this property because of nondeterministic operations performed during the execution. In general the trend toward more asynchrony and more load balancing plays against deterministic executions. Then the detection from replication becomes the problem of evaluating the similarity of results generated from replicated executions. Quantification of this similarity is an extremely hard problem because it assumes an understanding of how results diverge as a result of indeterministic operations, which itself depends on the thorough understanding of roundoff error propagation. Consequently, in SDC detection explore solutions that requires less resources and potentially relax the precision of detection.

A recent direction could be called approximate replication. The principle of approximate replication is to run the normal computation along with a an approximate computation that generates an approximate result. The comparison is then performed between the result and the approximate result. The approximate calculation gives upper and lower bounds within which the result of the normal calculation should be. Results outside the bounds are suspect; and corrective actions, such as further verification or re-execution, may be triggered. Approximate replication is a generic approach. It could be performed at the numerical method [5] level. It also could be used at the hardware level by comparing floating-point results of a normal operator with the ones of an approximate operator [37, 40, 75].

Another important issue related to SDC detection is the choice of methodology for evaluating and comparing algorithms. The standard process is to inject faults and obtain statistics on coverage and recovery overheads. Simulating hardware at the physical level is not feasible, and simulating it at a register transfer level is onerous. Most researchers inject faults in higher-level simulators [33]. But it is difficult to validate such fault injectors and know how the fault patterns they exhibit are related to faults exhibited by real hardware. A recent study compares different injection methods and their accuracy for the SPECint benchmark [27]. However, the accuracy of injection in HPC applications is still an open problem.

4.1.7. Integrative Approaches

The community has expressed in several reports the need for integrative approaches considering all layers from the hardware to the application. At least five projects explore this notion in different ways. One recent study demonstrates the benefit of cross-layer management of faults [64]. The containment domains approach [28] proposes a model of fault management based on a hierarchical design. Domains at a given level are supposed to detect and contain faults using techniques available at that level. If faults cannot be handled at that level, then the fault management becomes the responsibility of the next level in the hierarchy. Containment approaches operate between domains of the same level and between levels. Such approaches are applicable, in principle, at the hardware level and at all other software levels.

A different approach is proposed by the BEACON pub/sub mechanism in the Argo project [48], the GIB infrastructure in the Hobbes project [49], and the open resilience framework of the GVR project [100]. These mechanisms extend in different ways the concept of communication between software layers, originally proposed in the CIFTS project [62]. BEACON, GIB, and CIFTS/FTB are backplanes implementing exchanges of notifications and commands about errors and failures between software running on a system. Response managers should complement backplane infrastructures by lessening error and failure events and implementing mitigation plans.

4.2. Algorithmic Approaches

In our paper of five years ago, we discussed approaches for either recovering from or successfully ignoring faults, including early efforts in algorithm-based fault tolerance and in fault-oblivious iterative methods. Since that time, significant progress has been made in algorithmic approaches to detecting and recovering from faults. For example, the 2014 SIAM conference on parallel processing featured four sessions of seventeen talks covering many aspects of resilient algorithms. Here, we describe some of the progress in this area. The work presented is just a sampling of recent results in algorithm-based fault tolerance and is meant to give the reader a starting point for further exploration of this area.

Perhaps the most important change has been a clearer separation of the faults into two categories: fail-stop (a process fails and stops, causing a loss of all state in the process) and fail-continue (a process fails but continues, often due to a transient error). The latter has two important subcases based on whether the fault is detected or not. An example of the former is a failure of an interconnect cable, allowing the messaging layer to signal a failure but permitting the process to continue. An example of the latter is a undetected, uncorrectable memory error. Considerable progress has been made in the area of fail-continue faults, spurred by the recognition that transient faults (sometimes called soft faults), rather than fail-stop faults, are likely to be the most important type of faults in exascale systems.

Dense Linear Algebra. Fault-tolerant algorithms for dense linear algebra have a long history; at the time of our original paper, several techniques for algorithms such as matrix-matrix multiplication that made use of clever checksum techniques were already known [24, 65]. Progress in this area includes the development of ABFT for dense matrix multiplication [25] and factorizations [36] that addresses fail-stop faults. These provide the necessary building blocks to address complete applications; for example, a version of the HPLinpack benchmark that handles fail-stop faults with low overhead has been demonstrated [31]. Recent work has extended to the handling of soft faults or the fail-continue case for dense matrix operations [30].

Sparse Matrices and Iterative Methods. Algorithms involving sparse matrices and using iterative methods are likely to be important for exascale systems because many of the science applications that are expected to run on these systems solve large sparse linear and nonlinear systems. Work here has looked at both fail-stop and fail-continue faults. For example, [88, 89] evaluates a number of algorithms for both stability and accuracy in the presence of faults and describes an approach for transforming applications to be more resilient. A related approach can detect soft errors in Krylov methods by using properties of the algorithm so that the computation can be restarted or corrected [23]. Recent work has also looked at working with the system to provide more information and control for the library or application in handling likely errors, such as different types of DRAM failures [14].

Designing for Selective Reliability. One of the limitations of ABFT is that it can address errors only in certain parts of the application, for example, in the application's data but not in the program's instructions. This situation can be addressed in part by using judicious replication of pointers and other data structures [21], though this still leaves other parts of the code unprotected. An alternative is to consider variable levels of reliability in the hardware—using more reliable hardware for hard-to-repair problems and less reliable hardware where the algorithm can more easily recover and structure the ABFT to take advantage of the different levels of reliability [13, 72].

Efficient Checkpoint Algorithms. For many applications, a checkpoint approach is the only practical one, as discussed in Section 4.1. In-memory checkpoints can provide lower overheads than I/O systems but in their simplest form consume too much memory. Thus, approaches that use different algorithms for error-correcting code have been developed. An early example that exploited MPI I/O semantics to provide efficient blocking and resilience for file I/O operations is [59]. A similar approach has been used in SCR [78]. A more sophisticated approach, building on the coding and checksum approach for dense linear algebra, is given in [26].

5. Future Research Areas

The community has identified several research areas that are particularly important to the development of resilient applications on exascale systems:

- Characterization of hardware faults
- Development of a standardized fault-handling model
- Improved fault prediction, containment, detection, notification, and recovery
- Programming abstractions for resilience
- Standardized evaluation of fault-tolerance approaches

Characterization of hardware faults is essential for making informed choice about research needs for exascale resilience. For example, if silent hardware faults are exceedingly rare, then the hard problem of detecting such errors in software or tolerating their impact can be ignored. If errors in storage are exceedingly rare, while errors in compute logic are frequent, then research on mechanisms for hardening data structures and detecting memory corruptions in software is superfluous.

Suppliers and operators of supercomputers are often cagey about statistics on the frequency of errors in their systems. A first step would be to systematically and continually gather consistent statistics about current systems. Experiments could also be run in order to detect and measure the frequency of SDCs, using spare cycles on large supercomputers.

As work progresses on the next generation of integrated circuits, experiments will be needed to better characterize their sensitivity to various causes of faults.

Development of a standardized fault-handling model is key to providing guidance to application and system software developers about how they will be notified about a fault, what types of faults they may be notified about, and what mechanisms the system provides to assist recovery from the fault. Applications running on today's petascale systems are not even notified of faults or given options as to how to handle faults. If the application happens to detect an error, the computer may also eventually detect the error and kill the application automatically, making application recovery problematic. Therefore, today's petascale applications all rely on checkpoint-restart rather than on resilient algorithms.

Development of a standardized fault-handling model implies that computer suppliers can agree on a list of common types of fault that they are able to notify applications about. Recovery from a node failure differs significantly from recovery from an uncorrectable data value corruption. Resilient exascale applications may incorporate several recovery techniques targeted to particular types of faults. Even so, these resilient exascale applications are expected to be able to recover from only a subset of the common types of faults. Also, the ability to recover depends on the time from fault occurrence to fault notification. At the least, "fence" mechanisms are needed in order to ensure that software waits until all pending notifications that could affect previous execution are delivered.

A standardized fault-handling model needs to have a notification API that is common across different exascale system providers. The exact mechanism for notifying applications, tools, and system software components is not as critical as the standardization of the API. A standardized notification API will allow developers to develop portable, resilient codes and tools.

A fault-handling model also needs to define the recovery services that are available to applications. For example, if notified of a failed node, is there a service to add a spare node or to migrate tasks to other nodes? If the application is designed to survive data corruption, is there a way to specify reliable memory regions? Can one specify reliable computational regions in a code? In today's petascale systems, if such services exist, they are available only to the RAS system and are not exposed to the users. A useful fault model would have a standard set of recovery services that all computer suppliers would provide to the software developers to develop resilient exascale applications.

The fault-handling model should support hierarchical fault handling, with faults handled in the smallest context capable of doing so. A fault that affects a node should be signaled to that node, if it is capable of initiating a recovery action. For example, memory corruption that affects only application data should be handled first by the node kernel. The model should provide mechanisms for escalating errors. If a node is not responsive or cannot handle an error affecting it, then the fault should be escalated to an error handler for the parallel application using that node. If this error handler is not responsive or cannot handle the error, then the error should be escalated to a system error handler.

Improved fault prediction, containment, detection, notification, and recovery research requires major efforts in one area: the *detection* of silent (undetected) errors. Indeed, exascale is highly unlikely to be viable without error detection.

Significant uncertainty remains about the likely frequency of SDC in future systems; one can hope that work on the characterization of hardware faults will reduce this uncertainty. Much uncertainty also remains about the impact of SDCs on the executing software. We do not have, at this point, technologies that can cope with frequent SDCs, other than the brute force solutions of duplication or triplication.

Arguably, significant research has been done on algorithmic methods for handling SDCs. Current research, however, focuses on methods that apply to specific kernels that are part of specific libraries. We do not have general solutions, and we do not know whether current approaches could cover a significant fraction of computation cycles on future supercomputers. It is imperative to develop general detection/recovery techniques or show how to combine algorithmic methods with other methods.

Research on algorithmic methods considers only certain types of errors, for example, the corruption of data values, but not the corruption of the executing code, or hardware errors that affect the interrupt logic. It is important to understand whether such omission is justified by the relative frequency of the different types of errors or, if not, what mechanisms can be used to cope with errors that are not within the scope of algorithmic methods.

Moreover, fault detection mechanisms that are not algorithmic specific but use compiler and runtime techniques may be harder, but they would have a higher return because they would apply to all application codes.

Fault *containment* tries to keep the damage caused by the fault from propagating to other cores, to other nodes, and potentially to the corruption of checkpointed data, rendering recovery impossible. Successful containment requires early detection of faults before many computations are done and before that data is transmitted to other parts of the system. Once the fault has propagated, local recovery is no longer viable; and if detection does not occur before an application checkpoint, even global recovery may be impossible. Successful fault containment is key to low-cost recovery.

Fault *prediction* does not replace fault detection and correction, but it can significantly increase the effective system and application MTBF, thus significantly decreasing time wasted to checkpoint and recovery. Furthermore, techniques developed for fault prediction help root cause analysis and thus reduce maintenance time.

Fault *notification* is an essential component of the previously described fault-handling model. The provision of robust, accurate, and scalable notification mechanisms for the HPC environment will require new techniques.

Current overheads for *recovery* from system failures are significant; a system may take hours to return to service. Clearly, the time for system recovery must be reduced.

Programming abstractions for resilience will be able to grow out of a standardized fault handling model. Many programming abstractions have been explored, and several examples were described in previous sections. These research explorations have not shown a single programming abstraction for resilience that works for all cases. In fact, they show that several programming abstractions will need to be developed and supported in order to develop resilient exascale applications.

The development of fault-tolerant algorithms requires various resilience services: For example, some parts of the computation must execute correctly, while other parts can tolerate errors; some data structures must be persistent; others may be lost or corrupted, provided that errors are detected before corrupted values are used; and still other data structures can tolerate limited amounts of corruptions.

More fundamentally, one needs semantics for programs that enable us to express their reliability properties. The development of efficient fault-tolerant algorithms requires the programmer to think of computations as stochastic processes, with performance and outcome dependent on the distribution of faults. The validation or testing of such stochastic programs requires new methods as well.

Resilience services and appropriate semantics would also facilitate the development of system code. Many system failures, in particular failures in parallel file systems, are due to various forms of resource exhaustion, as servers become overloaded or short in memory and fail to respond in a timely manner. The proper configuration of such systems is a trial-and-error process. One would prefer systems that are resilient by design.

Standardized evaluation of fault tolerance approaches will provide a way to measure the efficiency of a new approach compared with other known approaches. It will also provide a way to measure the effectiveness of an approach on different architectures and at different scales. The latter will be important to determine whether the approach can scale to exascale. Even fault-tolerant Monte Carlo algorithms have been shown to have problems scaling to millions of processors [45]. Standardized evaluation will involve development of a portable, scalable test suite that simulates all the errors from the fault model and measures the recovery time, services required, and the resources used for a given resilient exascale application.

Acknowledgments: We thank Estaban Meneses for his help in the software section. This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, and under Award DESC0004131. This work was also supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-FG02-13ER26138/DE-SC0010049. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number ACI 1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

References

1. The Blue Waters super system for super science. *Contemporary High Performance Computing From Petascale toward Exascale, Jeffrey S. Vetter, editor, Chapman and Hall/CRC, pages 339–366, ISBN: 978-1-4665-6834-1, 2013.*

2. Guillaume Aupy, Anne Benoit, Thomas Hérault, Yves Robert, and Jack Dongarra. Optimal checkpointing period: Time vs. energy. *CoRR*, abs/1310.8456, 2013.

3. A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

4. L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In *Proc. 2011 Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC11)*. ACM, 2011.

5. Austin R Benson, Sven Schmit, and Robert Schreiber. Silent error detection in numerical time-stepping schemes. *International Journal of High Performance Computing Applications*, April, 2014.

6. Susmit Biswas, Bronis R. de Supinski, Martin Schulz, Diana Franklin, Timothy Sherwood, and Frederic T. Chong. Exploiting data similarity to reduce memory footprints. In *Proceedings of IEEE IPDPS*, pages 152–163, 2011.

7. W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Extending the scope of the checkpoint-on-failure protocol for forward recovery in standard MPI, concurrency and computation: Practice and experience, special issue: Euro-par 2012. July 2013.

8. Wesley Bland. User level failure mitigation in MPI. In *Euro-Par 2012: Parallel Processing Workshops*, pages 499–504. Springer, 2013.

9. G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale, concurrency and computation: Practice and experience. November 2013.

10. Mohamed-Slim Bouguerra, Ana Gainaru, Leonardo Arturo Bautista-Gomez, Franck Cappello, Satoshi Matsuoka, and Naoya Maruyama. Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing. In *Proceedings of IEEE IPDPS*, pages 501–512, 2013.

11. A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Correlated set coordination in fault tolerant message logging protocols, concurrency and computation: Practice and experience. Vol. 25, No. 4:pp. 572–585, 2013.

12. Aurelien Bouteiller, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Hrault, and Yves Robert. Multi-criteria checkpointing strategies: Response-time versus resource utilization. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 420–431. Springer Berlin Heidelberg, 2013.

13. P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. Fault-tolerant linear solvers via selective reliability. *ArXiv e-prints*, June 2012.

14. PatrickG. Bridges, Mark Hoemmen, KurtB. Ferreira, MichaelA. Heroux, Philip Soltero, and Ron Brightwell. Cooperative application/OS DRAM fault recovery. In Michael Alexander, Pasqua DAmbra, Adam Belloum, George Bosilca, Mario Cannataro, Marco Danelutto, Beniamino Martino, Michael Gerndt, Emmanuel Jeannot, Raymond Namyst, Jean Roman, StephenL. Scott, JesperLarsson Traff, Geoffroy Valle, and Josef Weidendorfer, editors, *Euro-Par 2011: Parallel Processing Workshops*, volume 7156 of *Lecture Notes in Computer Science*, pages 241–250. Springer Berlin Heidelberg, 2012.

15. Allan G Bromley. Charles Babbage's analytical engine, 1838. *Annals of the History of Computing*, 4(3):196–217, 1982.

16. Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 155–164. ACM, 2008.

17. Greg Bronevetsky, Daniel J. Marques, Keshav K. Pingali, Radu Rugina, and Sally A. McKee. Compiler-enhanced incremental checkpointing for openmp applications. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 275–276, New York, NY, USA, 2008. ACM.

18. Franck Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

19. Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

20. Franck Cappello, Amina Guermouche, and Marc Snir. On communication determinism in parallel hpc applications. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–8. IEEE, 2010.

21. Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 91–100, New York, NY, USA, 2012. ACM.

22. Sayantan Chakravorty and L. V. Kale. A fault tolerance protocol with fast fault recovery. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium.* IEEE Press, 2007.

23. Zizhong Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 167–176, New York, NY, USA, 2013. ACM.

24. Zizhong Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Parallel and Distributed Processing Symposium, International*, page 76, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

25. Zizhong Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *Parallel and Distributed Systems, IEEE Transactions on*, 19(12):1628–1641, Dec 2008.

26. Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 213–223, New York, NY, USA, 2005. ACM.

27. Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10. IEEE, 2013.

28. Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *the Proceedings of SC12*, November 2012.

29. John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

30. Teresa Davies and Zizhong Chen. Correcting soft errors online in lu factorization. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 167–178, New York, NY, USA, 2013. ACM.

31. Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 162–171, New York, NY, USA, 2011. ACM.

32. N. DeBardeleben, J. Laros, J. Daly, S. Scott, C. Engelmann, and W. Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. Technical Report LA-UR-10-00030, DARPA, January 2010.

33. Nathan DeBardeleben, Sean Blanchard, Qiang Guan, Ziming Zhang, and Song Fu. Experimental framework for injecting logic errors in a virtual machine to profile applications for soft error resilience. In *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 282–291, Berlin, Heidelberg, 2012. Springer-Verlag.

34. Catello Di Martino, F Baccanico, W Kramer, J Fullop, Z Kalbarczyk, and R Iyer. Lessons learned from the analysis of system failures at petascale: The case of Blue Waters. In *The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, 2014.

35. R.W. Downing, J.S. Nowak, and L.S. Tuomenoksa. No. 1 ESS maintenance plan. *Bell System Technical Journal*, 43:5:1961–2019, 1964.

36. Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 225–234, New York, NY, USA, 2012. ACM.

37. Peter D. Dben, Jaume Joven, Avinash Lingamneni, Hugh McNamara, Giovanni De Michel, Krishna V. Palem, and T. N. Palmer. Low-cost concurrent error detection for floating-point unit (fpu) controllers. *Philosophical Transactions of the Royal Society A, 20130276*, 372(2018), 2014.

38. John Daly (editor), Bob Adolf, Shekhar Borkar, Nathan DeBardeleben, Mootaz Elnozahy, Mike Heroux, David Rogers, Rob Ross, Vivek Sarkar, Martin Schulz, Marc Snir, and Paul Woodward. Inter Agency Workshop on HPC Resilience at Extreme Scale. `http://institute.lanl.gov/resilience/docs/Inter-AgencyResilienceReport.pdf`, February 2012.

39. Mootaz Elnozahy (editor), Ricardo Bianchini, Tarek El-Ghazawi, Armando Fox, Forest Godfrey, Adolfy Hoisie, Kathryn McKinley, Rami Melhem, James Plank, Partha Ranganathan, and Josh Simons. System resilience at extreme scale. Technical report, Defense Advanced Research Project Agency (DARPA), 2009.

40. Patrick J Eibl, Andrew D Cook, and Daniel J Sorin. Reduced precision checking for a floating point adder. In *Defect and Fault Tolerance in VLSI Systems, 2009. DFT'09. 24th IEEE International Symposium on*, pages 145–152. IEEE, 2009.

41. Mohammed el Mehdi Diouri, Olivier Glück, Laurent Lefèvre, and Franck Cappello. Energy considerations in checkpointing and fault tolerance protocols. In *Proceedings of FTXS workshop, IEEE/IFIP DSN'12*, pages 1–6, 2012.

42. J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for hpc. In *Proceedings of the IEEE 32nd International Conference on Distributed Computing Systems ICDCS*, pages 615–626, June 2012.

43. James Elliott, Mark Hoemme, and Frank Mueller. Evaluating the impact of SDC on the GMRES iterative solver4. In *Proceedings of International Parallel and Distributed Processing Symposium, IPDPS'14*, 2014.

44. Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

45. Christian Engelmann. Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale. *Future Generation Computer Systems (FGCS)*, 30(0):59–65, January 2014.

46. Christian Engelmann and Swen Böhm. Redundant execution of HPC applications with MR-MPI. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011*, pages 31–38, 2011.

47. Christian Engelmann, Geoffroy R. Vallée, Thomas Naughton, and Stephen L. Scott. Proactive fault tolerance using preemptive migration. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and network-based Processing (PDP) 2009*, pages 252–257, February 18-20, 2009.

48. Pete Beckman et al. Argo: An exascale operating system. In *http://www.mcs.anl.gov/project/argo-exascale-operating-system.*

49. Ron Brightwell et al. Hobbes - an operating system for extreme-scale systems. In *http://xstack.sandia.gov/hobbes/.*

50. Graham E. Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, UK, 2000. Springer-Verlag.

51. Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, and Ron Brightwell. rMPI: increasing fault resiliency in a message-passing environment. Technical Report SAND2011-2488, Sandia National Laboratories, Albuquerque, NM, 2011.

52. Kurt B Ferreira, Rolf Riesen, Patrick Bridges, Dorian Arnold, Jon Stearley, H Laros III James, Ron Oldfield, Kevin Pedretti, and Ron Brightwell. Evaluating the viability of process replication reliability for exascale systems. In *ACM/IEEE Conference on Supercomputing (SC11)*, Nov 2011.

53. David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

54. Ana Gainaru, Franck Cappello, and William Kramer. Taming of the shrew: modeling the normal and faulty behaviour of large-scale hpc systems. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1168–1179. IEEE, 2012.

55. Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Fault prediction under the microscope: A closer look into HPC systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE/ACM SC'12*, page 77. IEEE Computer Society Press, 2012.

56. Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Failure prediction for HPC systems and applications current situation and open issues. *International Journal of High Performance Computing Applications*, 27(3):273–282, 2013.

57. Al Geist. Private communication, 2012.

58. Al Geist, Bob Lucas, Marc Snir, Shekhar Borkar, Eric Roman, Mootaz Elnozahy, Bert Still, Andrew Chien, Robert Clay, John Wu, Christian Engelmann, Nathan DeBardeleben, Rob Ross Larry Kaplan Martin Schulz, Mike Heroux, Sriram Krishnamoorthy, Lucy Nowell, Abhinav Vishnu, and Lee-Ann Talley. U.S. Department of Energy fault management workshop. Technical report, DOE, 2012.

59. William Gropp, Robert Ross, and Neill Miller. Providing efficient I/O redundancy in MPI environments. In Dieter Kranzlmüller, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number LNCS3241 in Lecture Notes in Computer Science, pages 77–86. Springer Verlag, 2004. 11th European PVM/MPI User's Group Meeting, Budapest, Hungary.

60. William D. Gropp and Ewing Lusk. Fault tolerance in MPI programs. *International Journal of High Performance Computer Applications*, 18(3):363–372, 2004.

61. Amina Guermouche, Thomas Ropars, Marc Snir, and Franck Cappello. Hydee: Failure containment without event logging for large scale send-deterministic MPI applications. In *Proceedings of IEEE IPDPS*, pages 1216–1227, 2012.

62. Rinku Gupta, Pete Beckman, B-H Park, Ewing Lusk, Paul Hargrove, Al Geist, Dhabaleswar K Panda, Andrew Lumsdaine, and Jack Dongarra. CIFTS: A coordinated infrastructure for fault-tolerant systems. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 237–245. IEEE, 2009.

63. Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for Linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

64. Chen-Han Ho, Marc de Kruijf, Karthikeyan Sankaralingam, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Mechanisms and evaluation of cross-layer fault-tolerance for supercomputing. In *Proceedings of ICPP*, pages 510–519, 2012.

65. Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

66. Joshua Hursey, Richard L. Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G. Solt. Run-through stabilization: An MPI proposal for process fault tolerance. In *Proceedings of EuroMPI*, pages 329–332, 2011.

67. Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R. de Supinski, and Rudolf Eigenmann. Mcrengine: A scalable checkpointing system using data-aware aggregation and compression. *Scientific Programming*, 21(3-4):149–163, 2013.

68. William M. Jones, John T. Daly, and Nathan DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 276–279, New York, NY, USA, 2010. ACM.

69. D. S. Katz, J. Daly, N. DeBardeleben, M. Elnozahy, B. Kramer, L. Lathrop, N. Nystrom, K. Milfeld, S. Sanielevici, S. Cott, , and L. Votta. 2009 fault tolerance for extreme-scale computing workshop, Albuquerque, NM - March 19-20, 2009. Technical Report Technical Memorandum ANL/MCS-TM-312, MCS, ANL, December 2009.

70. D.S. Katz and R.R. Some. NASA advanced robotic space exploration. *Computer*, 36(1):52–61, 2003.

71. Ikhwan Lee, Michael Sullivan, Evgeni Krimer, Dong Wan Kim, Mehmet Basoglu, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Survey of error and fault detection mechanisms v2. Technical Report TR-LPH-2012-001, LPH Group, Department of Electrical and Computer Engineering, The University of Texas at Austin, December 2012.

72. Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S Vetter. Rethinking Algorithm-Based Fault Tolerance with a Cooperative Software-Hardware Approach. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

73. Dong Li, Jeffrey S Vetter, and Weikuan Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *SC12: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, Salt Lake City, 11/2012 2012.

74. Antonina Litvinova, Christian Engelmann, and Stephen L. Scott. A proactive fault tolerance framework for high-performance computing. In *Proceedings of the 9th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2010*, 2010.

75. M. Maniatakos, P. Kudva, B.M. Fleischer, and Y. Makris. Low-cost concurrent error detection for floating-point unit (FPU) controllers. *Computers, IEEE Transactions on*, 62(7):1376–1388, July 2013.

76. E. Meneses, O. Sarood, and L.V. Kalé. Energy profile of rollback-recovery strategies in high performance computing. *Parallel Computing*, 2014.

77. Esteban Meneses, Laxmikant V. Kalé, and Greg Bronevetsky. Dynamic load balance for optimized message logging in fault tolerant HPC applications. In *Proceedings of IEEE Cluster*, pages 281–289, 2011.

78. A. Moody, G. Bronevetsky, K. Mohror, and B.R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 International Conference on High Performance Computing, Networking, Storage and Analysis (SC10)*, pages 1–11, 2010.

79. Xiang Ni, Esteban Meneses, Nikhil Jain, and Laxmikant V. Kale. ACR: Automatic checkpoint/restart for soft and hard error protection. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13. IEEE Computer Society, November 2013.

80. Xiang Ni, Esteban Meneses, and Laxmikant V. Kalé. Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm. In *Proceedings of IEEE Cluster'12*, Beijing, China, September 2012.

81. Akira Nukada, Hiroyuki Takizawa, and Satoshi Matsuoka. Nvcr: A transparent checkpoint-restart library for nvidia cuda. In *IPDPS Workshops*, pages 104–113, 2011.

82. Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications. Optimization of multi-level checkpoint model for large scale HPC applications. In *Proceedings of IEEE IPDPS 2014*, 2014.

83. A. Rezaei, G. Coviello, CH. Li, S. Chakradhar, and F Mueller. Snapify: Capturing snapshots of offload applications on Xeon Phi manycore processors. In *Proceedings of High-Performance Parallel and Distributed Computing, HPDC'14*, 2014.

84. Thomas Ropars, Tatiana V. Martsinkevich, Amina Guermouche, Andre Schiper, and Franck Cappello. Spbc: leveraging the characteristics of MPI HPC applications for scalable checkpointing. In *Proceedings of IEEE/ACM SC*, page 8, 2013.

85. Takafumi Saito, Kento Sato, Hitoshi Sato, and Satoshi Matsuoka. Energy-aware I/O optimization for checkpoint and restart on a NAND flash memory system. In *In Proceedings of the Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, pages 41–48, 2013.

86. Osman Sarood, Esteban Meneses, and L. V. Kale. A cool way of improving the reliability of HPC machines. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE/ACM SC'13*, Denver, CO, USA, November 2013.

87. Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proceedings of IEEE/ACM SC'12*, page 19, 2012.

88. Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. *42rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

89. Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

90. Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2):129–173, May 2014.

91. L. Spainhower and T.A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5.6):863–873, 1999.

92. Omer Subasi, Javier Arias, Jesus Labarta, Osman Unsal, Adrian Cristal, and Barcelona Supercomputing Center. Leveraging a task-based asynchronous dataflow substrate for efficient and scalable resiliency, research report of polythecnic university of catalonia - computer architecture department. num: Upc-dac-rr-cap-2013-12. 2014.

93. Alexander Randall V. The Eckert tapes: Computer pioneer says ENIAC team couldn't afford to fail - and didn't. *Computerworld*, 40(8), February 2006.

94. Chao Wang, F. Mueller, C. Engelmann, and S.L. Scott. Hybrid checkpointing for MPI jobs in HPC environments. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 524–533, Dec 2010.

95. Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration and back migration in HPC environments. *J. Parallel Distrib. Comput.*, 72(2):254–267, February 2012.

96. Gulay Yalcin, Osman Sabri Unsal, and Adrian Cristal. Fault tolerance for multi-threaded applications by leveraging hardware transactional memory. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 4. ACM, 2013.

97. John W Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

98. Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *SIGOPS Oper. Syst. Rev.*, 40(2):90–99, April 2006.

99. Gengbin Zheng, Xiang Ni, and L. V. Kale. A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale, in Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS). Boston, USA, June 2012.

100. Ziming Zheng, Andrew A. Chien, and Keita Teranishi. Fault tolerance in an inner-outer solver: a GVR-enabled case study. In *Proceedings of VECPAR 2014, Lecture Notes in Computer Science*, 2014.

101. Ziming Zheng, Zhiling Lan, Rinku Gupta, Susan Coghlan, and Peter Beckman. A practical failure prediction with location and lead time for Blue Gene/P. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pages 15–22. IEEE, 2010.

102. Ziming Zheng, Li Yu, Wei Tang, Zhiling Lan, Rinku Gupta, Narayan Desai, Susan Coghlan, and Daniel Buettner. Co-analysis of RAS log and job log on Blue Gene/P. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 840–851. IEEE, 2011.

DOI: 10.14529/jsfi140102

Runtime-Aware Architectures: A First Approach

Mateo Valero [1,2], *Miquel Moreto* [1], *Marc Casas* [1], *Eduard Ayguade* [1,2], *Jesus Labarta* [1,2]

In the last few years, the traditional ways to keep the increase of hardware performance at the rate predicted by Moore's Law have vanished. When uni-cores were the norm, hardware design was decoupled from the software stack thanks to a well defined Instruction Set Architecture (ISA). This simple interface allowed developing applications without worrying too much about the underlying hardware, while hardware designers were able to aggressively exploit instruction-level parallelism (ILP) in superscalar processors. With the irruption of multi-cores and parallel applications, this simple interface started to leak. As a consequence, the role of decoupling again applications from the hardware was moved to the runtime system. Efficiently using the underlying hardware from this runtime without exposing its complexities to the application has been the target of very active and prolific research in the last years.

Current multi-cores are designed as simple symmetric multiprocessors (SMP) on a chip. However, we believe that this is not enough to overcome all the problems that multi-cores already have to face. It is our position that the runtime has to drive the design of future multi-cores to overcome the restrictions in terms of power, memory, programmability and resilience that multi-cores have. In this paper, we introduce a first approach towards a Runtime-Aware Architecture (RAA), a massively parallel architecture designed from the runtime's perspective.

Keywords: Parallel architectures, runtime system, hardware-software co-design.

Introduction

When uniprocessors were the norm, Instruction Level Parallelism (ILP) and Data Level Parallelism (DLP) were widely exploited to increase the number of instructions executed per cycle. The main hardware designs that were used to exploit ILP were superscalar and Very Long Instruction Word (VLIW) processors. The VLIW approach requires to statically determine dependencies between instructions and schedule them. However, since it is not possible in general to obtain optimal schedulings at compile time, VLIW does not fully exploit the potential ILP that many workloads have. Superscalar designs try to overcome the increasing memory latencies, the so called *Memory Wall* [42], by using Out of Order (OoO) and speculative executions [18]. Additionally, techniques such as prefetching, to start fetching data from the memory ahead of time, deep memory hierarchies, to exploit the locality that many programs have, and large reorder buffers, to increase the number of speculative instructions exposed to the hardware, have been also used to enhance superscalar processors performance. DLP is typically expressed explicitly at the software layer and it consisted in a parallel operation on multiple data performed by multiple independent instructions, or by multiple independent threads. In uniprocessors, the Instruction Set Architecture (ISA) was in charge of decoupling the application, written in a high-level programming language, and the hardware, as we can see in the left hand side of Figure 1. In this context, the architecture improvements were applied at the pipeline level without changing the ISA.

Some years ago, the traditional ways to keep increasing hardware performance at the rate predicted by Moore's Law vanished, additionally to the memory wall. The processor clock frequency stagnated because, when it increased beyond a threshold, the power per unit of area (power density) could not be dissipated. That problem was called the *Power Wall* [27]. A study

[1]Barcelona Supercomputing Center, Carrer Jordi Girona 29, 08034 Barcelona, Spain
[2]Universitat Politecnica de Catalunya, Carrer Jordi Girona 1-3, 08034 Barcelona, Spain

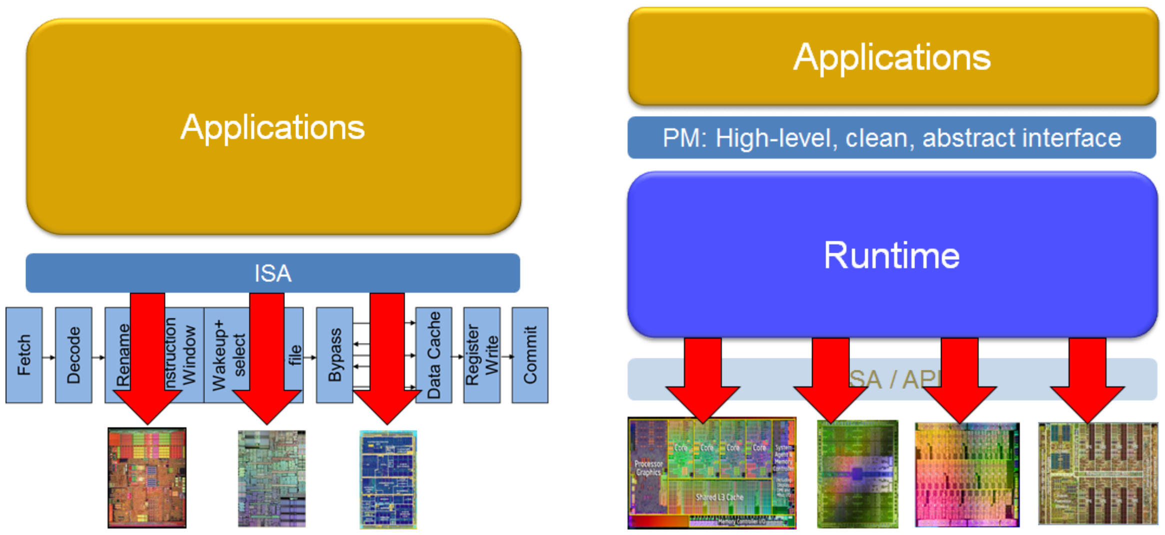


Figure 1. Left: Decoupling the hardware and the software layers in uniprocessors. Right: The runtime drives the hardware design in multiprocessors. We call this approach a Runtime-Aware Architecture (RAA) design.

made by the International Technology Roadmap for Semiconductors predicts an annual frequency increase of 5% for the next 15 years [19]. That means that we are left with parallelism alone in order to further increase performance.

To overcome the stagnation of the processor clock frequency, vendors started to release multi-core devices over a decade ago. They can potentially provide the desired performance gains by exploiting Task Level Parallelism (TLP). However, multi-core designs, rather than fixing the problems associated with the memory and power walls, exacerbate them. The ratio cache storage / operation stagnates or decreases in multi-core designs as well as the memory bandwidth per operation does, making it very hard to fully exploit the throughput that multi-core designs have. Another major concern is energy consumption, since if it keeps growing with the same rate as today, some major technological challenges like designing exascale supercomputers or developing petaflop mobiles will become chimeras. This set of challenges related to power consumption issues constitutes a new power wall.

Additionally, multi-core systems might have a heterogeneous set of processors with a different ISA, connected through several layers of shared resources with variable access latencies and distributed memory regions. To manage data motion among this deep and heterogeneous memory hierarchy while properly handling Non-Uniform Memory Access (NUMA) effects and respecting stringent power budget in data movements is going to be a major challenge in future multi-core machines. All these problems regarding programmability and data management across the memory hierarchy are commonly referred as the *Programmability Wall* [9].

Multi-core architectures can theoretically achieve significant performance with low voltages and frequencies. However, as the voltage supply scales relative to the transistor threshold voltage, the sensitivity of circuit delays to transistor parameter variations increases remarkably, which implies that processor faults will become more frequent in future designs. Additionally, the fact that the total number of cores in future designs will increase in several orders of magnitude only makes the fault prevalence problem more dramatic. In addition to the current challenges in parallelism, memory and power management, we are moving towards a *Reliability Wall* [43].

With the irruption of multi-cores and parallel applications, the simple interface between the hardware and the application started to leak. As a consequence, the role of decoupling again applications from the hardware was moved to the runtime system. This runtime layer is also in charge of efficiently using the underlying hardware without exposing its complexities to the application. In fact, the collaboration between the heterogeneous parallel hardware and the runtime layer becomes the only way to keep the programmability hardship that we are anticipating within acceptable levels while dealing with the memory, power and resilience walls.

Current multi-cores are designed as simple symmetric multiprocessors (SMP) on a chip. However, we believe that this is not enough to overcome all the problems that multi-cores already have to face. To properly take advantage of their potential, an enhanced hardware-software collaboration is required. It is our position that the runtime has to drive the design of future multi-cores to overcome the challenges of the above mentioned *walls*. We envision a Runtime-Aware Architecture (RAA), a holistic approach where the parallel architecture is partially implemented as a software runtime management layer, and the remainder in hardware. In this architecture, TLP and DLP are managed by the runtime and are transparent to the programmer. The idea is to have a task-based representation of parallel programs and handle the tasks in the same way as superscalar processors manage ILP, since tasks have data dependencies between them and a data dependency graph can be built at runtime or statically. As such, the runtime would drive the design of new architecture components to support its activity. In the right hand side of Figure 1 we can see a representation of this idea, where the application is implemented by using a high-level programming model that decouples it from the runtime and the hardware. The runtime not only uses the hardware efficiently, but also drives its design. As such, specific hardware components that support the runtime activities are a key point of the RAA approach.

Under the experience of the current ending age and equipped with a mature vision of what a productive future can be, many good ideas that disappeared during the RISC clock frequency boom of the 80's and 90's can be reshaped and applied with unforeseen scales or scope, resulting in an innovative vision of how to address the current embroilment where hardware technology has taken us.

Our approach towards parallel architectures offers a single solution that could alleviate most of the problems we encounter in the current approaches: handling parallelism, the memory wall, the power wall, the programmability wall, and the upcoming reliability wall in a wide range of application domains from mobile up to supercomputers. Altogether, this novel approach towards future parallel architectures is the way to ensure continued performance improvements, getting us out of the technological hardship that computers have turned into, once more riding on Moore's Law.

In Section 1 we describe more in detail how a task-based runtime manages the workload and how some ideas that are exploited by superscalar processors may be exploited by the runtime. In Section 2 we explain how the superscalar runtime has been used to efficiently exploit multi-cores. In Section 3 the concept of runtime-aware architectures is explained in detail. Section 4 talks about some related work. Finally, in Section 5 we comment the conclusions of this work.

1. Bringing the Superscalar Vision to the Runtime Level

We plan to use a runtime system that uses a task-based abstraction in which the programmer specifies which are the input and output arguments of the different tasks, which are going to

Table 1. Comparing superscalar and runtime visions

Superscalar	**Task-based Runtime**
Instructions	Tasks
Functional Units	Cores
Fetch and Decode Units	Cores
Registers (name space)	Main Memory
Registers (storage)	Local Memory
Out-of-Order Execution	
Pipelined Execution	
Speculative Execution	

be managed in the same way as superscalar processors manage instructions. This dataflow information allows the runtime to dynamically build and maintain a Task-Dependency Graph (TDG), which constitutes the foundation of a tight collaboration with the hardware to drive scheduling decisions and to discover opportunities to manage data movement in the architecture.

More precisely, as we can see in Table 1, the runtime layer conceives the different tasks of a parallel application as if they were instructions in a superscalar processor. Similarly, the fetch, decode and functional units in a pipeline can be seen as the cores of the heterogeneous many-core hardware, while registers can be foreseen as the local and main memories of parallel architectures. As such, concepts like Out-of-Order, pipelined, and speculative execution appear naturally at the runtime level in terms of task-based parallel applications. The task-dependency graph allows the runtime to execute independent tasks Out-of-Order, as instructions in traditional superscalar architectures. Deep pipelines, which are typically a component of superscalar processors, also appear at the many-core level in terms of sequences of tasks that can be overlapped to achieve more performance. In future many-core systems with hundreds of cores, idle cores can be used to speculatively execute tasks to accelerate application progress and prefetch data into the chip.

Since our approach consists in processing tasks in the same way as superscalar architectures handle dynamic instructions, many approaches that have been typically applied at the instruction pipeline level can inspire runtime optimizations and new hardware designs. Even more, as the task-dependency graph is much more complex and has much more parallelism to exploit than instruction dependencies, significant optimizations in terms of performance, power or resilience can be achieved by exposing the task-dependency graph to the available hardware and then balance the workload accordingly. Also, new architectural components to support the runtime activity are expected to play a key role.

In particular, we plan to implement our approach in the top of the OpenMP Superscalar (OmpSs) [11] runtime layer, which represents the state of the art in task-based runtime layers. OmpSs is an embedding of StarSs [29] in OpenMP.

2. Efficiently Exploiting Parallel Architectures from the Runtime

The task abstraction and the management of parallelism from the runtime system represented a major breakthrough in parallel programming. Exposing the dataflow across tasks, enables the runtime system to efficiently operate the parallel hardware in the same way the superscalar processors manage the functional units. This opens the door to a vast amount of complex optimizations that the runtime system and the underlying architecture can perform in a transparent way, providing sustained performance improvements across new hardware generations. For the rest of the section, we comment several techniques that the runtime can apply to overcome each one of the above mentioned walls.

Memory Wall The runtime scheduler can detect and exploit temporal **data reuse** by reordering tasks that reuse the same input data, or that use the outputs of the previous tasks. Also, it can make decisions about data distribution, allocating data close to where the task will be executed, prefetching data ahead of time, and creating explicit copies for increased locality if the same data is required in multiple locations at the same time. Bellens et al. [3] show the potential utility of these techniques in the Cell Broadband Engine microarchitecture, which has eight accelerators, each with a 256KB local memory, and a PowerPC processor. The runtime takes care of the task scheduling and data handling between the different processors of this heterogeneous architecture by using a locality-aware mechanism to reduce the overhead of data transfers from the PowerPC to the accelerators.

Minimizing **data movement** in the memory hierarchy is indeed a key technique to deal with the memory wall, as it reduces the number of accesses to the main memory and exploits synergies between the different components of the memory system. Some initial results have been already obtained in CPU+GPU systems [7]. The runtime system moves the data as needed between the different nodes and GPUs minimizing the impact of communication by using affinity scheduling, caching, and by overlapping communication with the computational task. When a GPU kernel is launched, the GPU threads request a new task to the scheduler. Then, the transfer of any data that might be needed by the prefetched task is initiated. In this way, by the time this task can be executed the data will already be available. This prefetch is more effective when combined with overlapped computation and data transfers.

To allow processors exploiting more ILP, register **renaming** has been exploited in superscalar architectures. Such approaches allow having more physical registers than logical registers, avoiding serialization penalties due to registers reuse. They require keeping track of dependencies between instructions' operands to determine if a new renaming register can be assigned to an architecture register. Renaming can help removing anti-dependencies between instructions, which can also be applied to tasks [4]. Renaming can be applied at task instantiation time, or delayed until just before task execution, similarly to virtual registers [15], which can delay the allocation of physical registers until a late stage in the pipeline, instead of doing it in the decode stage. Since the task-dependency graph can be generated ahead of time, it is possible to delay memory allocation to tasks until they start executing. This **virtual resource allocation** allows other tasks in the critical path to take advantage of this extra memory.

Power Wall The potential of exploiting the **critical path** has been already evaluated in MPI programs [6]. By using the knowledge of the critical path and combining it with straightforward profile-based techniques, a set of compact performance indicators that describe important performance-related questions, such as load imbalance, resource consumption or dynamic workload, an efficient scheduling can be derived. These performance indicators can be used by the runtime system to efficiently manage the load, trade performance for power or vice versa, and overall, optimize and adapt runtime decisions to the users' needs. Improvements up to 18% in execution time and 21% in energy efficiency have been achieved by using dynamic performance metrics to adjust runtime decisions [37].

Exploiting **task specialization** can give significant improvements since different tasks can be scheduled and mapped to different hardware components to deliver the maximum performance while spending low power. For example, by using memory-specific performance information (i. e. cache capacity and memory bandwidth required), the different tasks can be mapped to hardware components with the required memory resources or, alternatively, if a task was already running, it would be possible to switch-off non-required pieces of the memory. That would provide very important improvements in terms of power consumption.

Also, the programmer or the compiler can provide, for the same task, different versions of code targeting several accelerators and, according to the hardware state, the runtime can choose which version of the code must be used [30]. As such, if the machine has some cores with support for vector instructions, the runtime can reduce power consumption be scheduling the appropriate tasks to them to exploit their reduced fetch and decode power consumption.

Reliability Wall A wide range of techniques can be applied at the runtime level to increase applications resilience and deal with the reliability wall. If we assume that the tasks are idempotent, we can take check-points of the tasks' inputs and re-execute them if some fault takes place [38]. To reduce the memory overhead, we can apply a smart copy mechanism that takes just one checkpoint per input, even if it is used by multiple tasks.

Alternatively, it is also possible to extend the programming model to enable the user to specify pieces of code that are particularly sensitive from the resilience perspective by using special pragma annotations. That would give to the runtime the information on what tasks should be mapped into more resilient hardware components.

Another possible improvement that task-based runtimes allow is to tolerate the latency that recomputation techniques induce by overlapping computation with recovery techniques and let the execution progress if the faulty task is not in the critical path. That is an excellent framework to deploy algorithmic recoveries as its cost can be tolerated by overlapping them with standard runtime tasks.

Programmability Wall The runtime can handle **data dependencies** allowing the programmer to deliver straightforward code where just the input and output parameters of the different tasks are specified, which notably simplifies the work of efficiently programming heterogeneous many-core architectures. As such, the task-dependency graph is used by the runtime to expose the parallel workload to the available hardware in a transparent way from the programmer point of view, in the sense that the application source code does not contain information on how to handle the workload besides specifying the input/output parameters.

As such, the programming hardships observed in heterogeneous many-core architectures such as the Cell processor [41] can be avoided by considering the runtime management as a part of the hardware that efficiently manages the load without the need of explicitly exposing the problem to the programmers. As a consequence, the runtime takes care of balancing the load among different cores and is able to assign more tasks to faster cores. The same runtime layer can easily adapt to programs with multiple implementations of the same task type (i.e. tuned for different accelerators or cores), and decide in which execution unit the task has to be run.

A task-based approach can potentially reduce the **synchronization costs** that shared memory approaches typically have and thus achieve significant performance improvements when they manage highly parallel workloads on many-cores chips. As they allow the programmer to specify program parts called tasks, which can be executed concurrently, the mapping of tasks to threads is done dynamically by a runtime environment without any specific programmer responsibility in the way that synchronization costs are reduced.

3. Runtime-Aware Architectures

Our envisioned Runtime-Aware Architecture (RAA) constitutes a new paradigm of parallel computing systems in which the runtime management layer drives the hardware design, and they both collaborate to leverage an unprecedented degree of parallelism on-chip and exploit information on data dependencies. In this section, we describe in detail the different aspects of such hardware-software collaboration and we discuss some new opportunities brought by it. We expose the several aspects of the RAA approach by describing several techniques to deal with the memory, power, resilience and programmability walls and explaining how the hardware-software collaboration can exploit them.

Memory Wall Increasing **data reuse** in superscalar architectures was crucial to reduce the impact of the memory wall. In RAA's, there is a great margin for performance and energy efficiency improvements in the development of scheduling strategies that exploit this knowledge about the future, encoded in the task-dependency graph. RAA's offer the possibility to extend these ideas with specific hardware support that automatically handle these data transfers with the help of the runtime system by relying on software coherency or on specific architectural support to propagate updates across the multiple copies of data.

Prefetching had an extremely important role in fighting the memory wall in superscalar architectures. In RAA's, we can combine the capabilities of the runtime scheduler to exploit information about future data transfers, with the right kind of hardware support to optimize such transfers. The required hardware support is asynchronous data movement operations, that enable the scheduler to overlap data transfers required for future tasks with the execution of the current task, which reduces the bandwidth requirements and tolerates memory latencies as the transfer is out of the critical path [25]. The design requirements for this new memory system differ significantly from today's designs where memory transfers are always in the critical path.

Since the runtime system can decide in advance which tasks will execute in a given core after the current executing task finalizes, it can also determine the data required by the upcoming task and prefetch this data to the desired level of the cache hierarchy using locking and flushing mechanisms developed with this objective [14, 28]. This technique has a lot of potential, but if applied too aggressively or too early, can evict active data of the current executing task, negatively impacting the performance of the application. As an alternative, special purpose buffers can be added to the architecture to store the inputs of future tasks without affecting the contents of the cache hierarchy. Also, producer-consumer data can be efficiently forwarded with the adequate hardware support [23].

Minimizing on-chip and off-chip **data movements** is very important in terms of final performance and power consumption. Exploiting locality via reuse or prefetching is not going to be enough in future many-core systems. Simplified coherence protocols guided by the runtime system can be used to reduce coherence traffic [24]. An interesting complement consists in upgraded movement primitives that perform user defined transformations on the data as it is being transferred. In-memory functional units for integer, floating point or vector operations have been proposed in the past, but these transformations can also be done in the network routers. Also, more advanced Network-on-Chip (NoC) topologies that can dynamically adapt data movements depending on the network contention can help in reducing NoC's contention and communication latencies. In general, criticality-aware communications will be essential in our envisioned RAA.

We can also envision strategies that assign more or faster resources to tasks that are in the **critical path** (or likely to be) of the parallel application. In a many-core system, scheduling critical tasks to faster cores while minimizing resource contention and data motion is of paramount importance and requires a tight collaboration between the architecture and the runtime system. Since hybrid NoC's and memories will be a reality in such systems, developing the adequate prioritization mechanisms for communications and memory accesses of critical tasks is a key point to reduce overall execution time of the parallel application.

Power Wall The critical path is one of the most important dynamic properties of a parallel program. As it is the global execution path of tasks that forces wait operations on other tasks without itself being stalled, the events that are not included on it are, up to certain extend, **latency tolerant**, as they can take more time to complete without hurting the performance of the whole application. That opens a wide range of potential improvements in terms of power consumption or performance: If a given memory access is not in the critical path, it can be performed in a long-latency and thus low power region of the memory. Similarly, tasks that are not in the critical path can be mapped to slower and thus low power hardware components of the many-core system. Alternatively, if a particular task is in the critical path, it can be mapped to faster hardware components as its early completion may significantly reduce the waiting time of other tasks and thus reduce overall energy consumption.

In superscalar processors, specialized functional units can improve performance and achieve low power consumption rates for different phases of programs' execution. **Heterogeneous designs** can accelerate many applications that combine compute-intensive and control-intensive phases of computation, which should ideally be handled by different processing elements. These codes include large-scale scientific computations, complex simulations of physical phenomena, complex visualizations or financial markets' predictions. While systems with heterogeneous functional units can significantly improve performance and power consumption, they require to properly partition the application's code across all the available functional units or to efficiently manage data motion between functional units. Thus, the runtime system can become an excellent management layer to efficiently use hardware with heterogeneous cores and accelerators without making the programmability harder. As the task-dependency graph can be generated ahead of tasks' execution, the management of heterogeneous functional units can be properly planned. Additionally, some historical information on the tasks that have been already executed in terms of performance or resource usage can be kept by the runtime. By combining this historical data with the task-dependency graph, efficient scheduling decisions can be made by the runtime.

Reliability Wall Reducing **error propagation** is crucial to increase applications' resilience and deal with the reliability wall. Since soft faults that take place during a task execution will impact its outputs and all the tasks that use these as input parameters, RAA's offer important opportunities as the information contained in the task-dependency graph can be used as a proxy for tasks' sensitivity in terms of error propagation. As such, we can determine the most sensitive parts of the task-dependency graph and perform resilience enhancements on them. These enhancements can vary from straightforward approaches such as task replication to sophisticated algorithmic checkers. In any case, the resilience techniques should be able to check for errors and correct as many as possible. The hardware can support these resilience enhancements either by enabling fast and efficient recomputations, by using the data that is already in the cache, by supporting fast data movements, or by having a few dedicated cores exclusively focused on running algorithmic checkers to detect and correct data corruptions.

Error Correcting Codes (ECC) [35] are a well known technique based on encoding some data in a redundant way specially conceived to detect any corruption in the original data. They are typically implemented in memory systems to detect and correct data corruptions. ECC can even be used to measure the fault rate that a particular hardware component experiments, which allows to deliver machines with a certain resilience warranty. The RAA approach can also benefit from these approaches by performing ECC checks over data while they are being transferred. For example, if a given packet is waiting in the queue of a switch, some ECC-based checks can be performed on it to detect corruptions and restore data integrity if possible. Such ECC checkers would not impact on performance, as they are performed during transmisions. As such, memory, on- and off-chip networks should have support for ECC.

Programmability Wall For current runtime systems, the granularity of tasks has to be coarse enough to neglect the runtime overhead. As a consequence, the minimum duration of the tasks has to be in the order of tens-hundreds of milliseconds. For some algorithms, finer granularities of tasks are required to better express the inherent parallelism of the algorithm. For that reason, architectural support for task-based execution paradigms has been proposed to accelerate the runtime system and tolerate much finer task granularities.

Managing thousands of tasks at runtime requires hardware mechanisms to accelerate the construction of the TDG [12, 44], and scheduling decisions [22, 33]. In order to balance the load of the application, task stealing techniques can be implemented in hardware [22]. In heterogeneous systems, load balancing and scheduling tasks is very complex from the programmer's perspective. Being able to manage a massive amount of fine-grain tasks together with hardware support for load balancing and scheduling will significantly facilitate writing parallel applications for many-cores.

Our Envisioned RAA Architecture Figure 2 and 3 depict the different parts of our envisioned runtime-aware architecture. We are designing a massively parallel system with multiple nodes where each node has multiple sockets, and each socket multiple clusters of cores. It is our position that future exascale systems will be heterogeneous at multiple levels, with different types of execution units (big and little cores) and accelerators (GPUs, vector processors, network, etc.), interconnection networks (electric, optic, and wireless), and memories (DRAM, non-volatile memory, etc.). Also, managing deep memory hierarchies and multi-level interconnection networks will be critical to obtain peak performance.

Considering hardware heterogeneity, data locality and reuse, power consumption, and reliability at runtime is crucial to overcome the walls that threaten Moore's Law. Enriched information at hardware level is required to allow the runtime system to optimize these objectives. In order to minimize the overhead of this runtime system, specific hardware support is mandatory [12, 22, 33]. With the help of these structures, key aspects of RAA's such as building the TDG, task scheduling, load balancing, and data placement can be made without affecting the performance of the application.

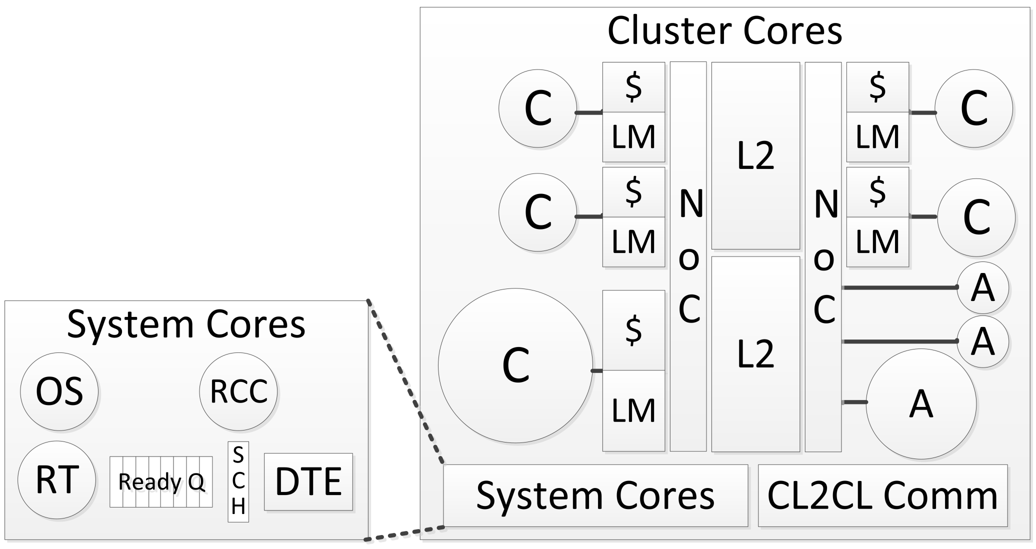


Figure 2. Runtime-Aware Architecture (RAA) system cores and cluster.

These structures would be the main part of the system cores block diagram in Figure 2. Similarly to the IBM Blue Gene/Q processor [17], a specific core for operating system activities (OS in Figure 2) is included in each cluster of cores, while the runtime front-end activities happen in a specific hardware (RT). It is still an open question if these two cores can be combined in a single structure. Runnable tasks are inserted into a queuing system similar to Carbon [22], which supports task stealing, and scheduled to the available cores. In order to characterize the hardware resources that each task type requires, a specific structure is conceived, denoted Resource Contention Core (RCC). Finally, a Data Transfer Engine (DTE) is in charge of data movements in the cores cluster. The DTE exploits data reuse, locality, and prefetches tasks inputs ahead of time.

The generic cluster cores in Figure 2 includes the execution cores, accelerators, on-chip network, and cache hierarchy. Effectively, processor cores thus serve as functional units. In RAA's, we propose to have a **hybrid memory hierarchy** with an L1 data cache and a local memory (or scratchpad) per core, as shown in Figure 2. Managing such a memory hierarchy is very difficult, but if done adequately, we can reduce significantly the coherence traffic and obtain a more energy efficient system. For example, we have proved that with the appropriate compiler support, such a hybrid hierarchy can be exploited for OpenMP codes [1]. In that approach, strided accesses are served by the local memory, while irregular accesses are served by the L1 data cache. A minimal hardware support is required for coherence and consistency purposes, but significant energy savings are obtained as a result. In the context of RAA's, task-based programming models such as OmpSs can be very helpful. For instance, task inputs and outputs can be automatically mapped to the local memories, while other accesses would be served by the L1 cache. The DTE in the system cores can manage DMA transfers to prefetch data in time. Finally, with this approach data movements due to coherence protocols would be significantly reduced.

The current **Network-on-Chip** (NoC) designs such as the ring- and bus-based topologies [39] provide an energy and throughput efficient solution for communication within small multi-cores. Since these traditional approaches for NoC's do not scale for many-cores, it is required to explore novel and scalable NoC approaches such as wireless interconnects, which can be used in conjunction with traditional wired and optical interconnects in novel RAA NoC topolo-

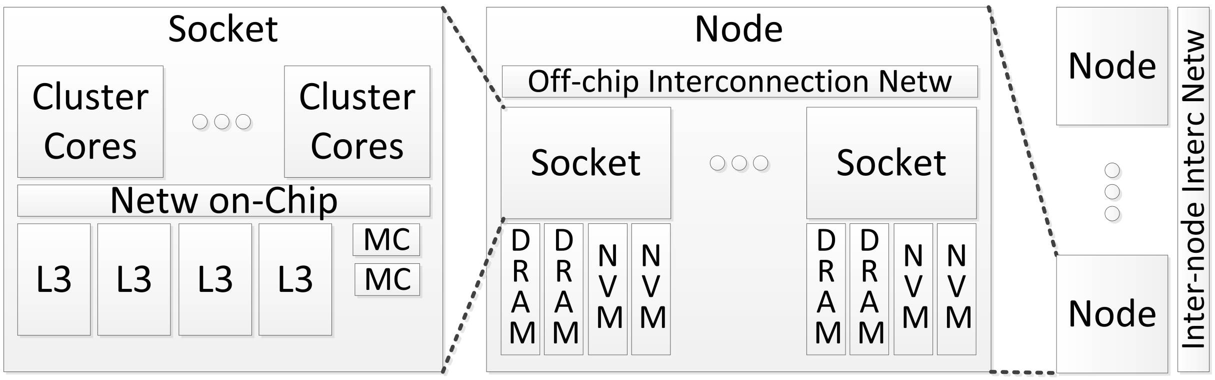


Figure 3. Runtime-Aware Architecture (RAA) socket, node and full system architecture.

gies for both 2D and 3D integrated circuits [45]. Finally, heterogeneous networks on chip are devised to serve the different characteristics of these structures in the hybrid memory hierarchy.

Cluster cores share a local L2 cache, while a large last-level on-chip cache is shared among different clusters in the socket. Local memories in different clusters will be able to communicate via a specific cluster to cluster communication engine (denoted *CL2CL Comm* in Figure 2). This engine could deal also with communications between different sockets in a node.

Different sockets in a node are connected with a high speed interconnection network, while different kinds of memories are available to the socket. Deciding whether to allocate memory in a regular DRAM or in a non-volatile memory is going to be a key aspect that is going to be easier to decide thanks to the enriched information provided by the dataflow representation of the application. For example, we can imagine that final outputs of an application should be mapped to persistent memories.

Several research groups have proposed system architectures with similarities to our envisioned RAA. For example, the SARC architecture [31] proposes having separate clusters of master and worker cores with local memories and coherent L2 and L3 caches. The Rigel architecture [21] proposes having clusters of cores with L1 instruction caches and incoherent L2 caches (per cluster), together with a global shared L3 cache. Finally, the Runnemede architecture [8] also relies on a dataflow execution model to execute in a near-threshold computing environment, with multiple clusters of homogeneous cores and a hierarchy of local memories. In this architecture, coherence between clusters is fully managed in software.

4. Related Work

4.1. Shared and Distributed Memory Programming Models

The most wide spread distributed memory API is MPI [16], which basically consists on a set of macros to explicitly indicate data exchanges between different tasks. Communications can be point to point or collective and use synchronous or asynchronous protocols, the first being more robust in the sense that data cannot be lost but paying the typical synchronization burden. OpenMP is an implementation of the classical shared memory multithreaded fork-join programming model [2]. By default, each thread runs its own parallelized section of code independently. Task- and data-level parallelism can be achieved trough work-sharing constructs that are used to divide the computational load among the threads. Threads are allocated to processors based on environment variables or in code using functions.

In the context of high performance computing, OpenMP is typically used to handle on-chip parallelism, since sharing data threads can use the memory resources more efficiently. However, OpenMP does not scale up to several tens of threads due to synchronization costs. Consequently, MPI must be used to achieved acceptable scalability in moderate and large scale runs. Typically, MPI is used to manage off-chip communications. It is interesting to state that hybrid MPI+OpenMP implementations somehow reflect the way the memory is organized in parallel clusters. As such, hybrid MPI+OpenMP have become the norm in the field of scientific computing.

4.2. Task-based Programming Models

Several task-based programming models have been developed in the past years: Cilk [5] is a runtime system for multithreaded codes. To fully use Cilk potential, the codes must be structured to expose parallelism and exploit locality, leaving Cilk's runtime with the responsibility of scheduling the computation to optimally run on a given platform. As such, the Cilk runtime system manages things like load balancing, synchronization, and communication protocols. Intel Threading Building Blocks (TBB) [32] is a C++ template library for task-based parallelism. It is supposed to simplify the parallel programming burden by asking the programmers to specify logical parallelism instead of threads and by letting the runtime library map logical parallelism onto threads that efficiently use the available hardware. CUDA (Compute Unified Device Architecture) is a parallel computing hardware and programming model developed by NVIDIA specifically designed for graphics processing units (GPUs). It allows the developers to access the virtual instruction set and memory of the parallel GPUs. OpenCL is a standard for cross-platform parallel programming. It allows programmers to implement and run parallel codes in heterogeneous platforms that may include CPU, GPU's, Digital Signal Processors (DSP) or other kinds of processors.

4.3. Dataflow Programming Models

Despite the fact that many of the programming models and computing paradigms mentioned above have been successful on achieving significant throughputs from current high performance computing infrastructures, the exhaustion of traditional performance enhancements techniques, like ILP or OoO, implies that more asynchronous and flexible programming models and runtime systems are going to be used to expose huge amounts of parallelism to the hardware. To reduce synchronization costs, optimize data motion across deep memory hierarchies, and handle critical path based optimizations, we need dataflow execution scenarios, which are much more flexible and have far more potential than traditional fork-join approaches.

Some dataflow programming models are being designed to overcome such issues. The Habanero [34] project is aimed to develop a programming model, a compiler and a runtime system to handle task-based asynchronous parallelism while taking advantage of locality among tasks and data distributed across the cores.

Charm++ [20] is a C++ based asynchronous message driven programming model. Its fundamental working units are message driven objects called *chares*. When the program triggers a message, an object is created and its associated work is carried out. Once the work is finished, the chare is destroyed and its output is sent to the next burst of chares that use these data as input. The chares are dynamically mapped to physical processors by the runtime system. Such mapping is transparent to the program and, therefore, it is done with the aim to increase load balancing and fault tolerance.

Interestingly, domain specific programming languages, like Sequoia [13], have been developed to specifically express hierarchical memory by using some programming model primitives and thus allowing the programmer to describe data motion vertically through the machine and to localize computation to particular memory locations within it.

4.4. Data-Graph Execution ISA's

Some ISA's have specifically designed to execute the instructions by using a data-graph approach. The LAU multiprocessor [10] was proposed aiming to take advantage of parallelism over three levels: Between jobs, between tasks within a job and between instructions within a task. To enable scalable and distributed processor cores, tiled architectures have been proposed [26, 36, 40]. They consist of multiple simple processing elements connected by an on-chip interconnect. Scheduling instructions in tiled architectures is crucial to obtain good performance [26]. Explicit Data Graph Execution (EDGE) architectures are examples of tiled architectures [36]. Unlike traditional processor architectures that operate at the granularity of a single instruction, EDGE ISA's support large graphs of computation mapped to a flexible hardware substrate, with instructions in each graph communicating directly with other instructions, rather than going through a shared register file. This capability not only reduces design complexity, but amortizes execution overheads over a large graph of instructions.

5. Conclusions

In this paper, we introduce a first approach towards a *Runtime-Aware Architecture* (RAA), a massively parallel architecture designed from the runtime's perspective. This approach offers a unified and general solution that can potentially solve most of the problems we encounter in the current approaches: handling parallelism, the memory wall, the power wall, the programmability wall, and the upcoming reliability wall in a wide range of application domains from mobile up to supercomputers. Altogether, this novel approach toward future parallel architectures is the way to ensure continued performance improvements, getting us out of the technological hardship that computers have turned into, once more riding on Moore's Law.

Acknowledgments*. This work has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557, the HiPEAC Network of Excellence, and by the European Research Council under the European Union's 7th FP, ERC Grant Agreement number 321253. We would like to thank Alex Ramirez, Osman Unsal, Adrian Cristal, Mario Nemirovsky, Ramon Beivide, Alejandro Rico and all the RoMoL team for the prolific discussions and all the feedback that we received while preparing this manuscript.*

References

1. L. Alvarez, L. Vilanova, M. González, X. Martorell, N. Navarro, and E. Ayguadé. Hardware-software coherence protocol for the coexistence of caches and local memories. In *SC*, 2012.

2. E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*,

20(3):404–418, Mar. 2009.

3. P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *SC*, 2006.
4. P. Bellens, J. M. Pérez, F. Cabarcas, A. Ramírez, R. M. Badia, and J. Labarta. CellSs: Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2):77–95, 2009.
5. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216, 1995.
6. D. Bohme, F. Wolf, B. R. de Supinski, M. Schulz, and M. Geimer. Scalable critical-path based performance analysis. In *IPDPS*, pages 1330–1340, 2012.
7. J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Implementing OmpSs support for regions of data in architectures with multiple address spaces. In *ICS*, pages 359–368, 2013.
8. N. P. Carter et al. Runnemede: An architecture for ubiquitous high-performance computing. In *HPCA*, pages 198–209, 2013.
9. B. Chapman. *The Multicore Programming Challenge*, volume 4847 of *Lecture Notes in Computer Science*, pages 3–3. Springer Berlin Heidelberg, 2007.
10. D. Comte, N. Hifdi, and J.-C. Syre. The data driven lau multiprocessor system: Results and perspectives. In *IFIP*, pages 175–180, 1980.
11. A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.
12. Y. Etsion, F. Cabarcas, A. Rico, A. Ramírez, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. Task superscalar: An out-of-order task pipeline. In *MICRO*, pages 89–100, 2010.
13. K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC*, 2006.
14. V. Garcia, A. Rico, C. Villavieja, P. Carpenter, N. Navarro, and A. Ramirez. Adaptive runtime-assisted block prefetching on chip-multiprocessors. Technical Report UPC-DAC-RR-2014-8, Department of Computer Architecture, UPC, May 2014.
15. A. González, J. González, and M. Valero. Virtual-physical registers. In *HPCA*, pages 175–184, 1998.
16. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.
17. R. Haring et al. The IBM Blue Gene/Q compute chip. *IEEE Micro*, 32(2):48–60, Mar. 2012.
18. J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.).* Morgan Kaufmann, 2012.

19. International technology roadmap for semiconductors (ITRS), system drivers. In *ITRS*, 2011.

20. L. V. Kalé and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *OOPSLA*, pages 91–108, 1993.

21. J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *ISCA*, pages 140–151, 2009.

22. S. Kumar, C. J. Hughes, and A. D. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA*, pages 162–173, 2007.

23. M. Manivannan, A. Negi, and P. Stenström. Efficient forwarding of producer-consumer data in task-based programs. In *ICPP*, pages 517–522, 2013.

24. M. Manivannan and P. Stenström. Runtime-guided cache coherence optimizations in multi-core architectures. In *IPDPS*, 2014.

25. V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *ICS*, pages 5–16, 2010.

26. M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Instruction scheduling for a tiled dataflow architecture. In *ASPLOS*, pages 141–150, 2006.

27. T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, Apr. 2001.

28. V. Papaefstathiou, M. Katevenis, D. S. Nikolopoulos, and D. N. Pnevmatikatos. Prefetching and cache management using task lifetimes. In *ICS*, pages 325–334, 2013.

29. J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *Int. J. High Perform. Comput. Appl.*, 23(3):284–299, Aug. 2009.

30. J. Planas, R. M. Badia, E. Ayguade, and J. Labarta. Self-adaptive OmpSs tasks in heterogeneous environments. In *IPDPS*, pages 138–149, 2013.

31. A. Ramirez et al. The SARC architecture. *Micro, IEEE*, 30(5):16–29, Sept 2010.

32. J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.

33. D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *ASPLOS*, pages 311–322, 2010.

34. J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS*, pages 181–192, 2009.

35. J. Sim, G. H. Loh, V. Sridharan, and M. O'Connor. Resilient die-stacked DRAM caches. In *ISCA*, pages 416–427, 2013.

36. A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *MICRO*, pages 89–102, 2006.

37. S. Sridharan, G. Gupta, and G. S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *ICS*, pages 337–348, 2013.

38. O. Subasi, F. J. Arias, O. Unsal, J. Labarta, and A. Cristal. Leveraging a task-based asynchronous dataflow substrate for efficient and scalable resiliency. Technical Report UPC-DAC-RR-CAP-2013-12, Department of Computer Architecture, UPC, May 2013.

39. B. Vermeulen, J. Dielissen, K. Goossens, and C. Ciordas. Bringing communication networks on a chip: test and verification implications. *IEEE Communications Magazine*, 41(9):74–81, 2003.

40. E. Waingold, M. B. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. P. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

41. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *CF*, pages 9–20, 2006.

42. W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

43. X. Yang, Z. Wang, J. Xue, and Y. Zhou. The reliability wall for exascale supercomputing. *IEEE Trans. Comput.*, 61(6):767–779, June 2012.

44. F. Yazdanpanah, D. Jiménez-González, C. Alvarez-Martinez, Y. Etsion, and R. M. Badia. Analysis of the task superscalar architecture hardware design. In *ICCS*, pages 339–348, 2013.

45. D. Zhao and Y. Wang. SD-MAC: Design and synthesis of a hardware-efficient collision-free QoS-aware MAC protocol for wireless network-on-chip. *IEEE Trans. Comput.*, 57(9):1230–1245, 2008.

Received June 8, 2014.

DOI: 10.14529/jsfi140103

Towards a performance portable, architecture agnostic implementation strategy for weather and climate models

Oliver Fuhrer[1], *Carlos Osuna*[2], *Xavier Lapillonne*[2], *Tobias Gysi*[3,4], *Ben Cumming*[5], *Mauro Bianco*[5], *Andrea Arteaga*[2], *Thomas C. Schulthess*[5,6,7]

We propose a software implementation strategy for complex weather and climate models that produces performance portable, architecture agnostic codes. It relies on domain and data structure specific tools that are usable within common model development frameworks – Fortran today and possibly high-level programming environments like Python in the future. We present the strategy in terms of a refactoring project of the atmospheric model COSMO, where we have rewritten the dynamical core and refactored the remaining Fortran code. The dynamical core is built on top of the domain specific "Stencil Loop Language" for stencil computations on structured grids, a generic framework for halo exchange and boundary conditions, as well as a generic communication library that handles data exchange on a distributed memory system. All these tools are implemented in C++ making extensive use of generic programming and template metaprogramming. The refactored code is shown to outperform the current production code and is performance portable to various hybrid CPU-GPU node architectures.

Keywords: numerical weather prediction, climate modeling, hybrid computing, programming models.

1. Introduction

The socio-economic value of numerical weather prediction and climate modelling is driving the continuous development and improvement of atmospheric models on current and emerging supercomputing architectures [22, 33]. However, attaining high scalability and efficiency with atmospheric models is a challenging task for several reasons. The typical codebase of an atmospheric model has several hundred thousand to million lines of Fortran code. Atmospheric models are community codes with large developer communities, and even larger user communities. As a consequence, strong commitments towards a single source code, performance portability across different high performance computing architectures, and a high usability of the source code by non-expert programmers are required. The life cycle of an atmospheric model consists of a development phase of approximately 10 years and a deployment phase with continuous development of 20 years or more. Due to the non-linear nature of the Earth's atmosphere, the large range of spatial and temporal scales involved and the complexity of the underlying physical processes, assessing the correctness and predictive capabilities of an atmospheric model is in itself a scientifically and computationally arduous task. Furthermore, many atmospheric models are characterized by a low arithmetic density and limited scalability. The low arithmetic density results from the algorithmic motifs in the most time-consuming parts of the codes, typically finite difference, finite element or finite volume discretizations on a structured or unstructured grid. As a consequence of these motifs, scalability is limited by the total number of horizontal

[1]Federal Office of Meteorology and Climatology MeteoSwiss, Switzerland, oliver.fuhrer@meteoswiss.ch
[2]Center for Climate Systems Modeling C2SM, ETH Zurich, Switzerland
[3]Department of Computer Science, ETH Zurich, Switzerland
[4]Supercomputing Systems AG, Zurich, Switzerland
[5]Swiss National Supercomputing Centre, ETH Zurich, Switzerland
[6]Institute for Theoretical Physics, ETH Zurich, Switzerland
[7]Computer Science and Mathematics Division, Oak Ridge National Laboratory, USA

grid points employed in a specific simulation, which can be low for multi-decadal global climate simulations.

Given these constraints, weather and climate codes are struggling to adapt to current and emerging hardware architectures. Adaptation of productive models to traditional multi-core systems typically use flat MPI parallelization, i.e. where each individual core is considered an independent compute node with no utilization of shared memory on a physical node of the computer system at the algorithmic level[8]. Attempts to migrate codes to a hybrid programming model such as MPI+OpenMP typically result in disappointing performance gains [6, 11, 26] or require substantial software refactoring that are not simpler than efforts to migrate models to hybrid architectures with GPU accelerators [19].

An extensive development effort based purely on Fortran plus compiler directives (OpenMP, F2C-ACC [13], and OpenACC) is currently ongoing for the global Non-hydrostatic Icosahedral Model (NIM) being developed at the National Oceanic and Atmospheric Administration (NOAA) [15, 17]. For inter-node communication NIM employs the Scalable Modeling System [14]. Using this approach, the dynamical core of NIM obtains good performance on multi-core CPUs, NVIDIA GPUs and Intel Xeon Phi accelerators. The team is currently working on porting the physics using the same approach.

A different approach was taken by Norman et al. [24] when porting the Fortran-based CAM-SE model to hybrid CPU-GPU architectures. CUDA Fortran is used for the parts of the model that run on GPUs, and the resulting port scaled and performed well on ORNL's Titan supercomputer (a Cray XK7 with Hybrid CPU-GPU nodes that are based on the NVIDIA Kepler architecture). The question of how such a port will migrate onto hybrid nodes with different accelerators that do not support the CUDA programming model is still to be addressed.

The Japanese Meteorological Agency is developing ASUCA, a new local-area numerical weather prediction model. In parallel, a group at Tokyo Institute of Technology developed another version of this Fortran/OpenMP code using C++/CUDA, which allowed them to run scale ASUCA up to 4000 GPUs on the TSUBAME 2.0 supercomputer [30, 31]. However, it is unclear how sustainable an approach that relies on different code bases for different architectures will be, and whether it will be supported by the climate and numerical weather prediction community.

Here we propose a different strategy: An existing monolithic Fortran code is partially rewritten based upon re-usable, domain or data structure specific tools that aim to separate concerns of domain decomposition for distributed memory and multi-threading on novel, heterogeneous node architectures. Our approach was applied to the regional weather and climate model COSMO, and the implementation is being fully integrated into the production trunk of the code. Our target is to run COSMO with a single code base at scale and in a performance portable manner both on CPU-based systems as well as different designs of hybrid CPU-GPU systems.

The problem of maintaining complex models on modern computing architectures is particularly acute in the context of the challenges posed by future exa-scale computing systems. With this paper we hope to make a positive contribution to this more general discussion over programming models. Using COSMO as a demonstration vehicle, we show how multiple architectures with their corresponding programming models can be supported with an implementation approach that shields the developers of the atmospheric model from the architectural complexities. The discussion is based on full production problems, rather than simplified benchmarks, in order

[8]The backend of an MPI implementation may still take advantage of a specific node architecture

to highlight all challenges of the refactoring effort. We demonstrate benchmarking results for various architectures that outperform the existing implementation of the model.

The paper is organized as follows. In Section 2 the COSMO model is briefly introduced. Section 3 will present the software strategy that underpins the implementation for different programming models with a single source code. In Section 4 performance results that affirm the different design choices, along with benchmarks of typical use-cases, are presented. Finally, we conclude with a summary and a discussion of future work in Section 5.

2. COSMO

The Consortium for Small-Scale Modeling (COSMO) is a consortium of seven European national weather services which aims to develop, improve and maintain a non-hydrostatic local area atmospheric model [9, 12, 32]. The COSMO model is used for both operational [4, 35] and research [20, 29] applications by the members of the consortium and many universities worldwide. Separate to the consortium, the CLM-Community [8] is an open international network of scientists are applying and developing the COSMO model for regional climate simulations. The current trunk version of the COSMO model is a Fortran code that relies on external libraries only for message passing (MPI) and I/O (grib, NetCDF). Being a community code used both for numerical weather prediction and climate simulation, COSMO is deployed on leadership class computing facilities at the largest compute centers, on medium sized research clusters at universities as well as on individual laptops for development.

The code base of COSMO can be divided into different components. The *dynamics* solves the governing fluid and thermodynamic equations on resolved scales. These partial differential equations (PDEs) are discretized on a three dimensional structured grid using finite difference methods. The so-called *physics* are modules representing sub-gridscale processes that cannot be resolved by the dynamics, such as cloud microphysics, turbulence and radiative transfer. For numerical weather prediction the incorporation of observational data – for example from ground measurement stations or weather radars – is handled by the *assimilation* code. Finally, the *diagnostics* compute derived quantities from the main prognostic variables and the *I/O* code handles input/output operations to files. A Runge-Kutta scheme is used to integrate the state variables forward in time [3], and Fig. 6 illustrates the order of operations on a time step. The COSMO model is a local area model, meaning that simulations are driven by external boundary conditions from a global model. The horizontal grid is a regular grid in a rotated latitude/longitude coordinate system. In this paper `i` and `j` refer to the indices enumerating horizontal longitude and latitude cells, and `k` refers to the vertical levels.

2.1. Benchmarks

Two model setups are used to generate the performance results presented in this paper. First, COSMO supports an initialization mode without requiring reading of external data from file by internally generating a model state and external boundary conditions from an analytical description of the atmosphere with random, three-dimensional perturbations. This configuration was used for the weak and strong scaling results presented in Section 4. This has the advantage that we can eliminate strong regional contrasts typically present in real data, which would inhibit a comparison between different domain sizes. Considerable care has been taken to ensure that the performance results with this model setup are representative for simulations with real data.

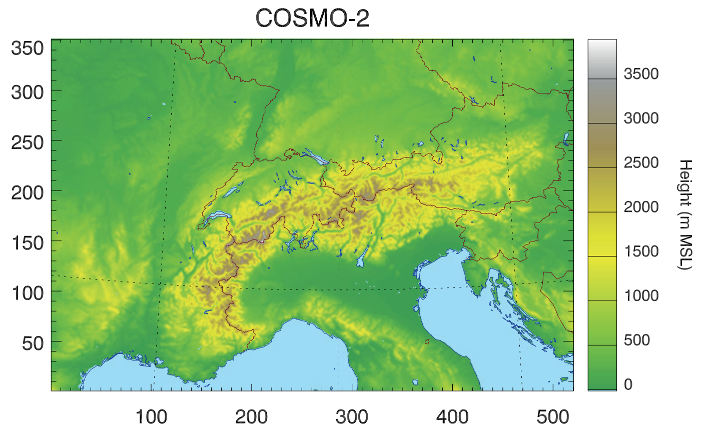


Figure 1. Computational domain of COSMO-2 benchmark. Contour shading shows model topography in meters above mean sea level (MSL)

Secondly, we use the 2.2 km operational setup of COSMO currently deployed at the Swiss national weather service MeteoSwiss. This setup, named COSMO-2, has a computational domain covering the greater Alpine region (see Fig. 1), with a total number of 520×350 horizontal gridpoints and 60 vertical levels. The COSMO-2 benchmark exercises the model in a representative way including all key physics modules. The COSMO-2 benchmark has to run with a time-compression factor of approximately 70 to meet the operational requirements of MeteoSwiss, meaning that the time-to-solution of a 24 h and 33 h simulation should correspond to 21 minutes and 29 minutes of wall-clock time, respectively. Such strict time-to-solution constraints are often the main driver choosing the size of computer systems used for operational weather forecasting.

3. Software Strategy

Like most weather and climate models, COSMO is implemented as a monolithic Fortran code that is highly optimized for the computer architectures in use at the weather services. At the time of writing these computer architectures are vector processors and distributed multi-core systems. The model is under continuous development, which means that there are many developers contributing code and who have a basic knowledge of the software architecture. Migrating such an implementation to new computer architectures that require different programming models is extraordinarily laborious. This was true in the past, when massively parallel computing systems appeared that have distributed memory with a message passing model. It continues to be a challenge today with multi-core processors that call for a threading model, as well as emerging architectures with hybrid CPU-GPU nodes.

In addition to software refactoring costs, the migration of a large weather and climate model like COSMO to new programming models requires a significant investment in training the developer community that has to adopt these models. Since the majority of this community are not computer scientists with neither extensive experience in software engineering or interests in new programming models, but domain scientists and applied mathematicians that have to be familiar with the underlying equations and their numerical discretizations, the adoption of new architectures can face significant resistance.

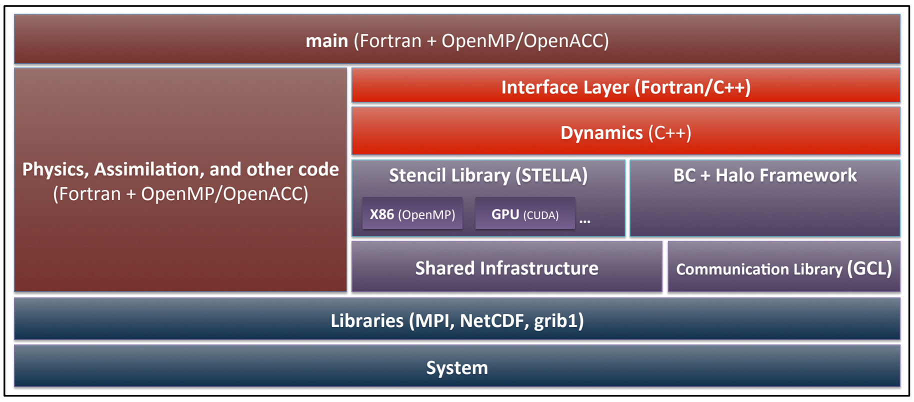

main (Fortran + OpenMP/OpenACC)
Physics, Assimilation, and other code (Fortran + OpenMP/OpenACC)
Interface Layer (Fortran/C++)
Dynamics (C++)
Stencil Library (STELLA)
X86 (OpenMP)
GPU (CUDA)
...
BC + Halo Framework
Shared Infrastructure
Communication Library (GCL)
Libraries (MPI, NetCDF, grib1)
System

Figure 2. Software architecture of the new COSMO code. Red colors correspond to user code, purple colors are for re-usable middleware tools which have been developed, blue is for I/O and system libraries.

There is thus a natural restoring force in favor incremental over disruptive or revolutionary changes to computer architectures and programming models. This situation is common for high-performance computing in general, and has to be addressed, because rapid gains in performance and improvements in production costs can only be achieved if new architectures are adopted in a timely manner. For COSMO, the root cause of this restoring force is the present software model: the monolithic Fortran code that prohibits a separation of concerns between the code that implements the atmospheric model the computer architecture-specific implementation.

A central emphasis of our refactoring effort of COSMO was to introduce new, domain or data structure specific tools that allow hiding architectural details and the corresponding programming models from the user code that implements the model. In order to minimize the cost of adoption, these new tools had to be integrated with the original Fortran code base.

Figure 2 illustrates the software architecture of the refactored COSMO code. Only the dynamics, which comprises about 20% of the 250'000 line trunk of COSMO but requires approximately 60% of the runtime, was completely rewritten [16]. To this end two domain-specific tools were developed and used.

The first of these, named STELLA (Stencil Loop Language), is a domain-specific embedded languages (DSELs) that leverage C++ and CUDA. STELLA was designed to implement the different stencil motifs for structured grids that are used in COSMO. The user uses a simple abstraction to specify stencil operations, without having to be concerned by the programming model and hardware specific loop and implementation details, which are generated by the DSEL using template metaprogramming. A short description of STELLA is given in Section 3.1.

The second library provides abstractions for operations at domain and sub-domain boundaries, specifically boundary conditions and the exchange of halo grid points. This framework leverages the Generic Communication Library (GCL [5]), which is a generic C++ library that implements a flexible and performance portable interface to specify halo-exchange patterns for regular grids. A short description of the communication framework and GCL is given in Section 3.3.

For all remaining parts of the COSMO code (physics, assimilation, etc.) we have used a less disruptive porting strategy based on compiler directives [21]. In Section 3.2, we will briefly discuss the implementation of the physics code that is based on OpenACC directives allowing the code to run on GPUs. A similar extension in terms of OpenMP directives that supports threading on a multi-core processors has been developed and will be provided in a future release of the code.

The choice to leave the physics, assimilation, and other parts of the code in their original Fortran form was a pragmatic one to minimize the impact of changes on the existing developer community. However, the elegant separation of architectural concerns from the atmospheric model specification is not possible with this approach, as compiler directives necessarily introduce details of the underlying programming model. The new architecture of the dynamics has the following benefits over this directives-based approach:

- Application developers (domain scientists) use an abstract hardware model (instead of a specific hardware architecture)
- Fine-grained parallelization at the node level is separated from course-grained data distribution across nodes.
- Separation of concerns, whereby we avoid exposing hardware specific constructs/optimizations to the user code.
- Single source code which can be maintained and developed by domain scientists.
- Re-usable software components (tools) which can be shared with other codes from the same domain, or other domains that use similar algorithmic motifs and data structures.
- Use of rigorous unit testing and regression testing that allows scalable development with large teams.

3.1. Dynamics rewrite (STELLA)

STELLA [16] is a DSEL for stencil codes on structured grids, that uses template metaprogramming [2] [1] to embed the domain-specific language within the C++ host language. The DSEL abstracts the stencil formulation from its architecture-specific implementation, so that users of the library write a single, performance-portable code. At compile-time the DSEL is translated into an executable with performance comparable to hand-written code optimized for the target architecture.

A typical stencil operator in COSMO concatenates multiple smaller stencil operators (so-called stencil stages). A stencil is made up of two logical components: 1) the loop-logic that iterates over the physical domain of the model represented by a structured grid; and 2) the loop body implementing the actual stencil update function. In STELLA an update function is implemented with a function object (functor), while the architecture specific loop-logic is defined using a DSEL that targets an abstract hardware model. Data locality is leveraged by fusing multiple stencil stages into a single kernel. Figure 3 depicts an example implemented using STELLA.

Table 1. Data layout of STELLA fields for the CPU and GPU backends

Backend	CPU	GPU
programming model	OpenMP	CUDA
storage order (by stride)	$j > i > k$	$k > j > i$

STELLA provides multiple backends for specific hardware architectures that utilize the corresponding programming model. Each backend employs its own data layout (see Table 1) and parallelization strategy that best maps onto the targeted architecture. At the time of writing,

```
// declarations
IJKRealField data;
Stencil horizontalDiffusion;

// declare stencil stage
template<typename TEnv>
struct Laplace {
  STENCIL_STAGE(TEnv)

  STAGE_PARAMETER(FullDomain, phi)
  STAGE_PARAMETER(FullDomain, lap)

  static void Do(Context ctx, FullDomain) {
  ctx[lap::Center()] = −4.0∗ctx[phi::Center()] +
      ctx[phi::At(iplus1)] + ctx[phi::At(iminus1)] +
      ctx[phi::At(jplus1)] + ctx[phi::At(jminus1)] ;
  }
};
```

```
//define and initialize the stencil
StencilCompiler::Build(
  horizontalDiffusion,
  // define the input/output parameters,
  pack_parameters(
    Param<res, cInOut>(dataOut), Param<phi, cIn)(data)
  ),
  define_temporaries(
    StencilBuffer<lap, double, KRange<FullDomain,0,0> >(),
  ),
  define_loops(
    define_sweep<cKIncrement>(
      define_stages(
        StencilStage<Laplace, IJRange<cIndented,−1,1,−1,1>,
          KRange<FullDomain,0,0> >(),
        StencilStage<Divergence, IJRange<cIndented,0,0,0,0>,
          KRange<FullDomain,0,0> >(),
      )
    )
  )
);

// execute the stencil instance
horizontalDiffusion.Apply();
```

Figure 3. Example of a STELLA stencil that implements a diffusion operator based on two stages: first a Laplace operator followed by a divergence operator. The left panel is a stencil stage for the Laplacian (note that the Divergence stage is not shown). The right panel shows the definition of the full stencil. It specifies the field parameters and temporary fields and assembles all the loops, defining its properties (ranges, ordering, etc.), by concatenating the two stages.

STELLA provides two backends, one based on OpenMP [34] for multi-core CPUs, and one based on CUDA [25] for GPUs (see Tab. 1).

The dynamics of COSMO was rewritten in C++ using the STELLA DSEL in order to provide a single source code that performs well both on CPU and GPU architectures. The dynamics is composed of ∼50 STELLA stencils of different size and shape. Additional C++ code organizes data fields in repositories and orchestrates the stencil calls along with calls to the halo exchange and boundary condition framework (see Section 3.3).

3.2. Fortran code refactoring (OpenACC)

The parts of the code that have been left in their original Fortran form were ported to GPUs with OpenACC compiler directives [27]. The OpenACC Application Programming Interface allows the programmer to specify, mostly at the loop level, which regions of a program should be run in parallel on the accelerator. The OpenACC programming model also provides control over data allocation and data movement between the host CPU and the accelerator. OpenACC is an open standard which is currently supported by three compiler vendors: Cray, PGI, and CAPS [7, 10, 28].

Figure 4 shows a code example which adds two two-dimensional arrays in Fortran. The compiler generates accelerator code for the region enclosed by the `parallel` and `end parallel` directives. The `data create` and `update host/device` directives control memory allocation on the device and data transfer between the CPU and the accelerator.

Atmospheric models such as COSMO have a relatively low arithmetic intensity, and the gain in performance obtained by running the computation on a GPU can be lost if data has to be moved back and forth between host and GPU memory. To avoid such data transfers wherever

```
!$acc update device(b,c)
!$acc parallel
do j = 1, NI
  do i = 1, NJ
    a(i,j) = b(i,j) + c(i,j)
  enddo
enddo
!$acc end parallel
!$acc update host(a)
!$acc end data
```

```
!$acc host_data use_device(a)
call c_wrapper(a)
!$acc end host_data
!$acc end data
```

Figure 4. Left: example of a two dimensional arrays addition in Fortran using OpenACC. Loops between the `parallel` and `end parallel` will be executed on parallel on the accelerator. Right: example using `host_data` directive to interface OpenACC and C/C++ code. The corresponding GPU address of variable `a` is passed to the routine c_wrapper.

possible all computations that require two-dimensional or three-dimensional fields are executed on the GPU.

The OpenACC approach is particularly convenient, as it allows porting large parts of the model in a relatively short time by simply adding the directives to the existing code. This naïve porting approach was used for large parts of the code which are not performance critical. For the performance critical parts, mostly in the physics, code refactoring was required to get acceptable performance with OpenACC. The main optimizations consist of restructuring loops in order to better expose parallelism and using larger kernels. A detailed description of the OpenACC porting and optimizations of the main components of the physics can be found in [21]. Some parts of the code which are not executed at every time step, such as I/O related computations, are still partially run on the CPU without degrading overall performance (see Figure 6).

3.3. Inter-node parallelization (GCL)

COSMO uses a two-dimensional domain decomposition in the horizontal plane to distribute data and work across nodes. The size and shape of overlap regions between domains (called halos) vary from field to field, as they are determined by the size of the stencils that require data from neighboring nodes. The current production version of COSMO uses MPI to exchange the data in the halo regions between nodes. Further methods are also needed in order to impose boundary conditions on the global boundary of the simulation domain.

The *halo update and boundary condition framework* combines these two operations (boundary conditions and halo exchanges) into a single framework. This allows the user to provide a unique description of these operations by defining the halo regions, fields and boundary conditions with one user interface. Just like the STELLA library, the framework abstracts implementation details of internode communication and application of boundary conditions. It supports the various memory layouts used in the backends for STELLA. For optimal performance, memory layout and boundary condition type are generated at compile time for a particular architecture. Furthermore, the framework provides Fortran bindings that allow it to be used from the Fortran side of the code as well. Figure 5 shows an example of the definition of a halo update and boundary condition object that can be executed any time a halo update is required by a stencil.

The data exchange in the halo updates is handled using the Generic Communication Library (GCL [5]). GCL is a C++ library with which users specify at compile time the data layout of the fields, the mapping of processes and data coordinates, data field value types, and methods for packing/unpacking halos. The library supports both host and accelerator memory spaces. If the

```
// define the ghost points that require updating
IJBoundary boundaryRegion;
boundaryRegion.Init(−2,2,−2,2);

// define a BC and halo−update job
HaloUpdateManager haloUpdate;
haloUpdate.AddJob(zeroGradientBCType, phi, boundaryRegion);

// apply a stencil to the data field phi
verticalDiffusion.Apply();

// start BC and halo−update job
haloUpdate.Start();

// do additional computation that does not modify data field "phi"

// complete BC and halo−update job
haloUpdate.Wait();

// apply stencil to the data field phi (consuming the halo points)
horizontalDiffusion.Apply();
```

Figure 5. Example of a configuration of a boundary condition and halo-update required for updating the halo points required by the diffusion operator introduced in Figure 3

underlying communication libraries are GPU-aware, the data exchange between accelerators will take advantage of direct GPU to GPU communication, bypassing the host memory. Presently GCL uses MPI, but other communication libraries for distributed memory architectures could be used.

3.4. Integration

As discussed above, a complete rewrite of the COSMO model was not an option. Thus the inter-operability of rewritten code parts or newly developed tools and the existing frameworks is of paramount importance. The integration of different programming models (OpenACC and STELLA) as well as flexible tools (STELLA and GCL libraries) with the legacy Fortran code framework of COSMO called for particular attention. The main challenge was that while the memory layout is fixed in the Fortran code, the memory layouts of STELLA fields are configurable and depend on the backend. Figure 6 shows a diagram of the COSMO time loop together with the porting approach for each part of the model.

In order to avoid expensive data transfers between the GPU and the CPU within each step of the time loop, all required data fields are allocated on the GPU at startup in the Fortran code using the OpenACC `data create` constructs during startup, where they remain for the entire run. The allocated fields are then reused in different parts of the Fortran code by using the `present` directive. Inter-operability with C++/CUDA can be achieved via the `host_data` OpenACC directive (see right panel of Figure 4) which allows access to the data field GPU pointers that can then be passed to a library function call.

An interface layer between the C++ and Fortran code was developed in C++ to provide Fortran bindings to the dynamics and boundary/halo framework. At every time step the C++ dynamics is invoked via the interface layer. The memory layout in the Fortran code is `ijk`, *i.e.* `i` is the stride-one dimension. The dynamics uses an `kij` storage order on the CPU for performance reasons (see Table 1). For the GPU backend the storage order is `ijk`, like the Fortran part, however the STELLA library can use aligned memory fields that offer better performance on the GPU. The interface layer supports two different modes: 1) shared fields, *i.e.* memory layouts are equivalent, the pointer allocated in the Fortran parts is directly reused in the dynamics, and

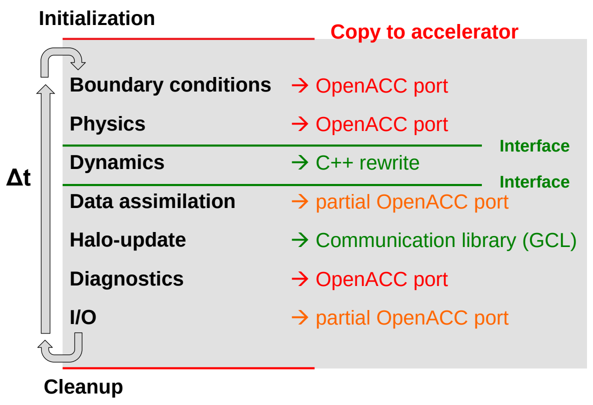

Initialization
Copy to accelerator
Δt
Boundary conditions → OpenACC port
Physics → OpenACC port
Interface
Dynamics → C++ rewrite
Interface
Data assimilation → partial OpenACC port
Halo-update → Communication library (GCL)
Diagnostics → OpenACC port
I/O → partial OpenACC port
Cleanup

Figure 6. Workflow of the COSMO model on GPU. After the initialization phase on CPU all required data is transferred to the GPU

no data transformation or copies are required; 2) copy fields, *i.e.* memory layouts of fields are different (this includes different alignments), the call to the dynamics will trigger a copy (and transformation) of fields upon entry and exit.

Table 2 summarizes the effect of copying data between the Fortran and C++ sections in the GPU backend. In this example, using an optimally aligned data structure in the dynamics (copy fields mode) is on par with sharing fields with the Fortran code (shared fields mode). Previous generation GPUs had a more significant penalty for un-aligned memory accesses and using copy mode was typically advantageous. In any case, the specific choice of data storage layout can be chosen freely in the C++ part of the code and the interface layer automatically recognizes if copies are required or pointers can be shared.

The Fortran code uses a two time-level scheme whereby each variable has two fields, and at every time step the pointers for the fields are swapped in order to reuse the allocated memory. Similarly, STELLA uses double buffering to swap the pointers between time steps. An important function of the interface layer is to ensure consistency of all the field pointers between the Fortran parts and the dynamics, which is non trivial.

The interface layer also provides Fortran bindings to the boundary condition and halo exchanges framework (Section 3.3). This allows the Fortran parts of COSMO to apply halo exchanges and the boundary conditions strategies using a unique implementation which can deal with any underlying memory layout and architecture backend.

Finally, considering the important refactoring effort for COSMO, systematic unit testing and regression testing was introduced and has become an integral part of the software strategy. All tests are executed on a daily basis using a continuous integration tool [18].

4. Performance Results

In this section we compare the performance of the refactored code (referred to as *new*) with with the current COSMO production code (referred to as *current*). We will also present detailed experiments which demonstrate the performance impact of specific design choices outlined in the previous section.

Table 2. Performance comparison of the copy-field and the shared-fields modes for passing fields between Fortran and C++ via the interface layer on a K20X GPU. All measurements are time per time step. Shown is stencil computation time, time for copying fields in the interface layer and the total dynamics time.

	stencil time [ms]	copy time [ms]	total [ms]
shared-field mode	148	0.200	204
copy-field mode	142	6.38	206

Unless explicitly stated, performance results have been measured on a hybrid Cray XC30 named Piz Daint[9] at the Swiss National Supercomputing Centre (CSCS) in Lugano. This system has a hybrid node architecture with a Intel Xeon E5-2670 CPU with 32 GB host memory and a NVIDIA Tesla K20X GPU with 6 GB GDDR5 memory. The compute nodes are interconnected with Cray's Aries network that has a three-level dragonfly topology.

4.1. Weak and strong scaling

The left panel in Fig. 7 shows weak scaling results of the current and new code on Piz Daint. Two configurations with 128x128x60 and 64x64x60 gridpoints are shown by the solid and dashed lines (as well as the solid and empty markers, respectively), respectively. The colors correspond to the current code running purely on the CPUs (blue squares), the new code running purely on the CPUs (red circles) and the new code running with use of the GPU (orange triangles). For all cases, the 3D domain is decomposed only over the horizontal i, j-dimensions. For the CPU cases, the per-node sub-domain is further decomposed onto 8 MPI ranks that are placed on the individual cores (we don't use OpenMP threading or hyper-threading). When running in hybrid mode, only a single MPI rank is placed on a node (one per GPU). The runtime shown on the y-axis corresponds to a forward integration of 100 timesteps with I/O disabled. All curves are almost flat, which corresponds to an almost perfect linear scaling up to 1000 nodes for both per-node domain sizes. Focusing on the 128x128x60 case using 9 nodes, see also table 4, we observe a 1.5x speedup when running the new code (red solid circles) on the CPU compared to the current code (blue solid squares) for the 1.5x problem size. Moving the new code from the CPU (red solid circles) to the hybrid mode with GPUs (orange solid triangles) leads to an additional speedup of 2.9x.

Strong scaling performance with the total problem size fixed at 512x512x60 (solid line) and 256x256x60 (dashed lines) for an integration of 100 time steps is shown in the right panel of Fig. 7. Since we decompose the computational domain only along the horizontal i, j-dimensions, the number of horizontal grid points decreases with increasing node count, while the number of vertical levels remains constant at 60 levels. Analogous to the weak scaling experiment presented above, the pure CPU runs are done with 8 MPI ranks per socket, leading to a finer decomposition than for the GPU runs. While the CPU results for both the current and new code (blue squares and red circles) show very good strong scaling up to more than 100 nodes, the runs in hybrid mode (orange triangles) saturate at much lower nodes counts. The reason for the rapid saturation of the hybrid architecture is that a minimal number of horizontal grid points are required to efficiently use of the GPU that requires minimal number of concurrent threads to saturate the

[9] `http://user.cscs.ch/computing_resources/piz_daint/index.html`

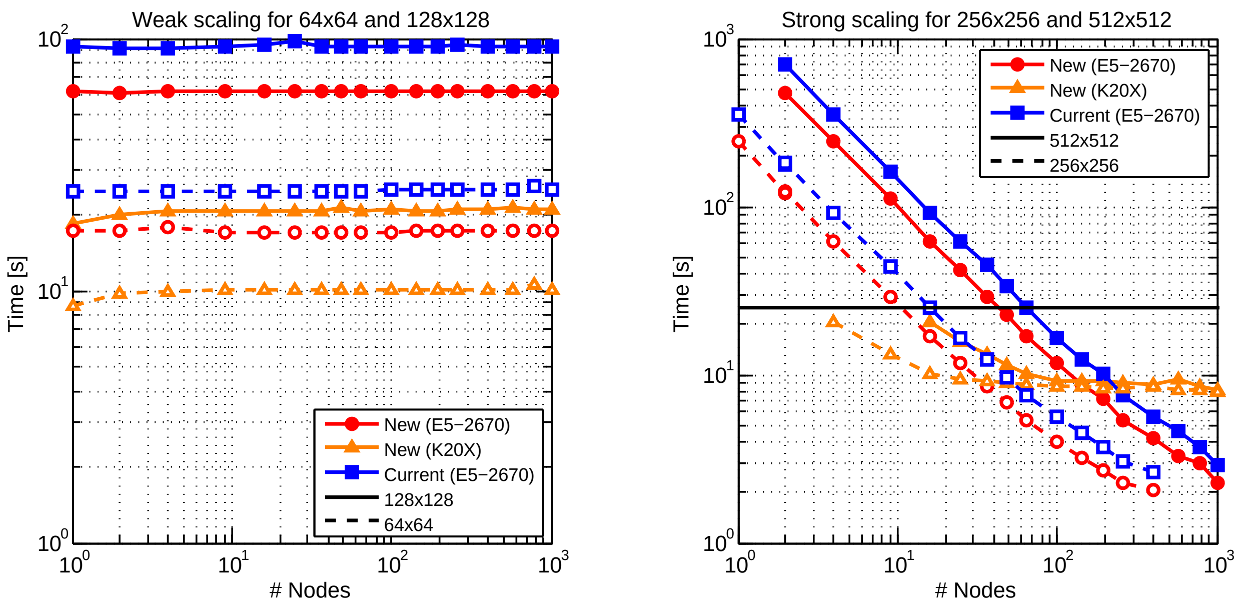


Figure 7. Left: weak scaling for two different per-node grid sizes (128x128 solid line, 64x64 dashed line). Right: strong scaling for two different total grid sizes (512x512 solid line, 256x256 dashed line). Both show the current trunk version of the code (red) and the refactored code running on CPU (blue) or GPU (green). Runtime is given in seconds and corresponds to a total of 100 timesteps. All simulations have been done with disabled I/O.

available memory bandwidth. The upper limit of the strong scaling curves is given by a minimum of 3x3 horizontal grid points per node which are required to run the COSMO model. The lower limit of the strong scaling curves is given by the available memory.

The horizontal solid black line delimits the time-to-solution requirement for the 2 km model of numerical weather prediction of MeteoSwiss. With this fixed time-to-solution requirement, one can map a given simulation to significantly less computational resources on the hybrid CPU-GPU architecture. For the 512x512x60 problem, the current code requires 64 CPU nodes (with one Intel Xeon E5-2670), while the new code requires 48 CPU nodes and only 14 hybrid nodes (with one Intel Xeon E5-2670 and a K20X GPU).

Table 3. Runtime with 128x128x60 grid points per node using 9 nodes, comparing the current code to the new code on CPU and GPU.

Code	Architecture	Time [s]	Speedup
Current	E5-2670	92.3	REF
New	E5-2670	61.6	1.5x
New	K20X	20.8	4.4x

4.2. Different node architectures

Simply put, the superior efficiency of the hybrid CPU-GPU nodes is due to the higher memory bandwidth of the GPU. This suggests that nodes with a different CPU to GPU ratios might be more attractive. In this section, we investigate the performance of the COSMO-2 benchmark on two different node architectures. We compare runs on Piz Daint that use 8 nodes, each with one Intel Xeon E5-2670 and one NVIDIA K20X socket, to runs on a single "fat node" system named *Opcode* that has a 4:1 GPU to CPU ratio. Specifically, Opcode is a Tyan FT77A server with two Intel Xeon E5-2670 sockets, each connected via the PCIe Gen3 bus to two PLX8747 PCIe switches which each are connected to two NVIDIA Tesla K20 GPUs, i.e. 4 GPUs per Intel Xeon socket or a total of 8 Tesla K20 GPUs per node. Figure 8 gives an overview of the different node architectures.

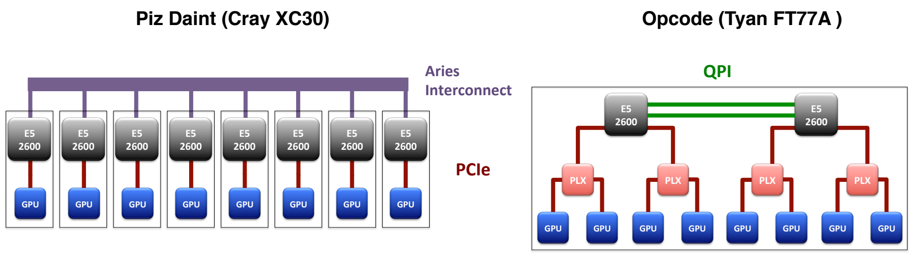


Figure 8. Comparison of the Piz Daint and Opcode systems used for the COSMO-2 benchmarks. Piz Daint nodes each have a CPU and GPU and are inter-connected via the Aries network. Opcode is a single node system with 2 CPUs and 8 GPUs, where 4 GPUs are connected to each CPU via PCIe.

Table 4 shows the runtime of the Fortran, C++ and communication components individually as well as the total runtime of the COSMO-2 benchmark. The simulation has been integrated up to +33 simulation hours. Note that the benchmark has a required time-to-solution of 29 minutes (1740 s), which on Piz Daint is almost achieved for this run using only 8 nodes.

For the dynamics, we would not expect large performance differences since STELLA uses the same backend and compiler for both architectures. The results indicate that the dynamics runs 10% slower on the Opcode system as compared to Piz Daint. This can be explained by the slightly lower peak memory bandwidth and floating-point performance of the K20 cards as compared to the K20X on Piz Daint.

On the other hand, the Fortran code which has been ported using OpenACC directives is 53% slower on the Opcode system. The rather large difference is due to the different Fortran compilers (Cray's in the case of Piz Daint and PGI on Opcode). As OpenACC compilers mature, it is to be expected that these large performance differences will decrease.

Table 4. COSMO-2 benchmark on different hybrid node architectures. The simulation time in [s] is given for a 33h forecast. The COSMO-2 computational domain (520x350) is decomposed to 2x4 GPUs on both platforms. On Piz Daint all the 8 GPUs are on different nodes, while on Opcode all the GPUs are on the same node

	Computation [s]		Communication [s]	Total [s]
	Dynamics (Stella)	Fortran (OpenACC)		
Piz Daint (K20X)	890	574	426	1889
Opcode (K20)	975	878	431	2283

In spite of very different interconnect hardware between the GPUs, the inter-GPU communication times are almost on par. On the Opcode system, communication is over the PCIe bus leveraging GPUDirect [23] or - in case two communicating GPUs are attached to different sockets - over the Intel QuickPath interconnect (QPI). Despite performing inter-GPU communication over the non-dedicated PCIe bus or the QPI, results for the Opcode system show a total communication time comparable to Piz Daint, where GPUs communicate with the network interface controller using GPUDirect over the PCIe, and different nodes are interconnected with the Cray's Aries network.

Finally, it should be noted that for running the refactored version of COSMO a system with fat-nodes is considerably more cost-effective and energy efficient. Furthermore, the refactored code is found to be performance portable to different node architectures.

4.3. GPU-to-GPU vs. communication over host

As has been mentioned in the previous section, GPUDirect allows communication between GPUs either on the same node or on different nodes without requiring copies to the CPU. Several MPI libraries leverage GPUDirect to enable faster GPU to GPU communication (G2G). G2G-enabled MPI libraries allow directly passing in GPU memory addresses to a restricted set of the library functions (e.g. MPI_Send/MPI_Recv). In this section, we compare the time of individual components of the halo-update on Piz Daint for different setups of the COSMO-2 benchmark.

Table 5. Shown are networking plus synchronization time, time for packing the data into messages, and time for copying message buffers to/from the CPU for different configurations of the dynamics of the COSMO-2 benchmark. The first two data columns show results for runs using the hybrid version of the code, with and without G2G (see text). The last column are measurements gathered for a CPU version of the code with exactly the same domain decomposition and setup. All values are in seconds.

Part	**GPU (with G2G)**	**GPU (without G2G)**	**CPU**
Network + Sync [s]	280.5	245.3	195.6
Packing [s]	33.2	33.5	52.9
Copy [s]	–	89.6	–
Total [s]	315.5	368.4	248.5

Table 5 shows time spent in the different parts of the halo-updates contained in the dynamics for a 33 h integration of the COSMO-2 benchmark. Over the whole simulation, a total of 7.19 GB of data is exchanged by each of the 8 MPI ranks used for the benchmark. For every halo-update, first the halos of the data fields have to be packed into linear buffers (Packing) the halos of the fields are packed into linear buffers. For runs using the hybrid code with G2G disabled, these linear buffers have to be copied to the CPU (Copy). Finally, the communication can be started by passing the data to the MPI library and waiting for the communication to finish (Network + Sync). There is a considerable load imbalance due to different computational loads for the MPI ranks which are situated at the corners of the computational domain, and the synchronization time is not negligible.

First, one can see that the packing time is lower on the GPU as compared to packing on the CPU, which can be explained by the higher memory bandwidth of the GPU. The large copy time required when disabling G2G indicates that it is worth avoiding these copies if a MPI library with G2G support is available. Finally, one can observe that the network plus synchronization time is considerably smaller for the CPU runs. Further investigations indicate that this is mostly due to a larger sensitivity of the GPU communication to load imbalance.

5. Summary and future work

We have discussed a reimplementation of the COSMO numerical weather prediction and regional climate model that is performance portable across multiple hardware architectures such as distributed multi-core and hybrid CPU-GPU systems. The dynamics has been rewritten entirely in C++. It is implemented using a domain-specific embedded language named STELLA [16] which is targeted for stencil computations on structured grids. STELLA is implemented using template metaprogramming and supports a multi-threaded model with fine grained parallelism on a variety of node architectures supported through architecture specific backends to STELLA The dynamics is also built on top of a generic halo exchange and boundary condition framework that provides the necessary functionality for domain decomposition on distributed memory systems. The data exchange between nodes is implemented in terms of a Generic Communication Library (GCL [16]). With this strategy, the domain scientists develop methods against an abstract machine model, rather than architecture specific programming models, allowing full flexibility in the choice of underlying architectures and programming models. For practical reasons concerning the user community, the physics and other parts of the COSMO code were left in Fortran and ported to hybrid CPU-GPU platforms with OpenACC directives. The performance results we presented show the refactored code to outperform the current production code by a factor 1.5 for full production problems. Scalability and performance of the new code on various architectures corresponds to expectations based on algorithmic motifs of the code and hardware characteristics. This demonstrates that an innovative, tools-based approach to software can exploit the full performance of a system just as well as a code that has been implemented in Fortran or C.

The reimplementation of COSMO started in Autumn 2010 after a brief feasibility study and negotiations with the community. Design and implementation of STELLA as well as the rewrite of the dynamics for a single node took about two years with two staff. Refactoring the physics parts took about three man-years, and the communication and boundary condition framework as well as the GCL took another two man- years. The overall integration and testing with the legacy Fortran code started immediately after completion of the first version of the rewritten dynamics and communication framework. It took two years and a total of three man-years. Intensive co-development of the G2G communication libraries at Cray and the GCL at CSCS during the last 9 months of the project led to the successful deployment of Piz Daint in Fall 2013 with the refactored COSMO code being one of the first applications to run at scale on the system after acceptance in January 2014.

Thus, the three-year reimplementation project of COSMO took about as long as the planing and procurement of a new supercomputer. It seems important that the innovation of the software is kept in sync with the hardware cycle. More important for future development, however, is the strategy we employed: breaking up a monolithic Fortran code into tools that are reusable in other domains. Libraries like STELLA, GCL, as well as the halo update and boundary condition framework can be readily adapted to new emerging hardware architectures. Furthermore, with these tools it will be easier to implement new dynamical cores, allowing faster innovation on the numerical methods and the models as well.

A new project is under way to extend STELLA to types of block-structured grids used in global models, such as icosahedral and cubed-sphere meshes. The goal of this project is to provide performance portable tools to implement global weather, climate and seismic models. Furthermore, the tools discussed here will be extended with Python bindings, in order to support future model development directly within a high-level and object-oriented language environment. The aim of such projects will be to cut down on the integration cost, which in the case of the COSMO reimplementation project were considerable. We propose that future implementations of complex models such as COSMO should be based on performance portable tools and descriptive high-productivity languages like Python that allow for more dynamic development, rather than monolithic codes implemented in prescriptive languages such as Fortran or C/C++. This will make model development much more efficient and facilitate the co-design of tools and hardware architectures that are much more performant and efficient than today's code. Furthermore, it will simplify the introduction of new programming models and languages, as their use can be limited to the appropriate part of the toolchain and there will be no immediate need to change the entire code base of a model.

Several people contributed to the new version COSMO and we would like to acknowledge them here: Tiziano Diamanti, Florian Dörfler, David Leutwyler, Peter Messmer, Katharina Riedinger, Anne Roches, Stefan Rüdisühli, and Sander Schaffner. Also, the authors would like to acknowledge the Center for Climate Systems Modeling (C2SM) and specifically Isabelle Bey for administrative support. We thank the Swiss National Supercomputing Centre (CSCS) for providing access to the compute resources used to carry out the benchmarks. Finally, we would like to acknowledge the High Performance and High Productivity (HP2C) Initiative for funding this research.

References

1. I. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, And Techniques From Boost And Beyond.* The C++ in-Depth Series. Addison Wesley Professional, 2005.

2. A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

3. M. Baldauf. Stability analysis for linear discretisations of the advection equation with runge–kutta time integration. *Journal of Computational Physics*, 227(13):6638 – 6659, 2008.

4. M. Baldauf, A. Seifert, J. Förstner, D. Majewski, and M. Raschendorfer. Operational convective-scale numerical weather prediction with the cosmo model: Description and sensitivities. *Monthly Weather Review*, 139:3387–3905, 2011.

5. M. Bianco. An interface for halo exchange pattern, 2012.

6. F. Cappelo and D. Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *Proceedings of 2000 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'00. ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), 2000.

7. CAPS. *CAPS the many core company*, 2014. `http://www.caps-entreprise.com/`.

8. Climate Limited-area Modelling Community. `http://www.clm-community.eu/`.

9. Consortium for Small-Scale Modeling. `http://www.cosmo-model.org/`.

10. Cray Inc. *Cray Fortran Reference Manual*, 2014. `http://docs.cray.com/books/S-3901-82/S-3901-82.pdf`.

11. M. J. Djomehri and H. H. Jin. Hybrid MPI+OpenMP programming of an overset CFD solver and performance investigations. NASA/NAS Technical Report NAS-02-002, NASA Ames Research Center, 2002.

12. G. Doms and U. Schättler. The nonhydrostatic limited-area model LM (lokal-model) of the DWD. Part I: Scientific documentation. Technical report, German Weather Service (DWD), Offenbach, Germany, 1999.

13. M. Govett. *F2C-ACC Users Guide, Version 4.2*, 2012. `http://www.esrl.noaa.gov/gsd/ab/ac/Accelerators.html`.

14. M. Govett, L. Hart, T. Henderson, J. Middlecoff, and D. Schaffer. The scalable modeling system: directive-based code parallelization for distributed and shared memory computers. *Parallel Computing*, 29(8):995–1020, 2003.

15. M. Govett, J. Middlecoff, and T. Henderson. Running the NIM next-generation weather model on gpus. In *Proceedings 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 792–796, 2010.

16. T. Gysi, O. Fuhrer, C. Osuna, M. Bianco, and T. Schulthess. Stella: A domain-specific language and tool for structured grid methods. *submitted*.

17. T. Henderson, J. Middlecoff, J. Rosinski, M. Govett, and P. Madden. Experience applying fortran GPU compilers to numerical weather prediction. In *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC 2011)*, pages 34–41, 2011.

18. Jenkins CI. *Jenkins the continuous integration tool*, 2014. `http://jenkins-ci.org/`.

19. H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, 1999.

20. W. Langhans, J. Schmidli, O. Fuhrer, S. Bieri, and C. Schär. Long-term simulations of thermally driven flows and orographic convection at convection-parameterizing and cloud-resolving resolutions. *Journal of Applied Meteorology and Climatology*, 52:1490–1510, 2013.

21. X. Lapillonne and O. Fuhrer. Using compiler directives to port large scientific applications to GPUs: An example from atmospheric science. *Parallel Processing Letters*, 24(1):1450003, 2014.

22. J. K. Lazo, J. S. Rice, and M. L. Hagenstad. Benefits of investing in a supercomputer to support weather forecasting research: An example of benefit cost analysis. *Yuejiang Academic Journal*, 1:1–22, 2010.

23. Mellanox. Nvidia gpudirect technology—accelerating gpu-based systems. `http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf`.

24. M. Norman, J. Larkin, R. Archibald, V. Anantharaj, I. Carpenter, P. Micikevicius, and K. Evans. Porting the spectral element community atmosphere model (CAM-SE) to hybrid gpu platforms. In *2012 SC COMPANION: HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS (SCC)*, pages 1788–1798, Salt Lake City, UT, 2012.

25. NVIDIA. *CUDA Parallel Computing Platform.* `https://developer.nvidia.com/cuda`.

26. L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review*, 44(3):373–393, 2002.

27. OpenACC Corporation. *The OpenACC Application Programing Interface*, 2011. `http://www.openacc.org/`.

28. Portland Group Inc. *PGI Compiler Reference Manual*, 2014. `http://www.pgroup.com/doc/pgiref.pdf`.

29. A. Possner, E. Zubler, O. Fuhrer, U. Lohmann, and C. Schär. A case study in modeling low-lying inversions and stratocumulus cloud cover in the bay of biscay. *Weather and Forecasting*, 29(2):289–304, 2014/06/01 2014.

30. T. Shimokawabe, T. Aoki, J. Ishida, K. Kawano, and C. Mtiroi. 145 tflops performance on 3990 gpus of tsubame 2.0 supercomputer for an operational weather prediction. In *Proc. Int. Conf. Comp. Sci.*, volume 4 of *Procedia Computer Science*, pages 1535–1544, 2011.

31. T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-fold speedup, 15.0 tflops full gpu acceleration of non-hydrostatic weather model asuca production code. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, Nov 2010.

32. J. Steppeler, G. Doms, U. Schättler, H. Bitzer, A. Gassmann, U. Damrath, and G. Gregoric. Meso gamma scale forecasts using the nonhydrostatic model LM. *Meteor. Atmos. Phys.*, 82, 2002.

33. N. Stern. Review on the economics of climate change. Technical report, HM Treasury, London, UK, 2006. `http://www.hm-treasury.gov.uk/stern_review_report.htm`.

34. The OpenMP ARB. *The OpenMP API Specification for Parallel Programming*, 2013. `http://www.openmp.org`.

35. T. Weusthoff, F. Ament, M. Arpagaus, and M. W. Rotach. Assessing the benefits of convection-permitting models by neighborhood verification: Examples from map d-phase. *Monthly Weather Review*, 138:3418–3433, 2010.

Received June 6, 2014.

DOI: 10.14529/jsfi140104

Communication Complexity of the Fast Multipole Method and its Algebraic Variants

Rio Yokota[1]***, George Turkiyyah***[1]***, David Keyes***[1]

A combination of hierarchical tree-like data structures and data access patterns from fast multipole methods and hierarchical low-rank approximation of linear operators from $\mathcal{H}$-matrix methods appears to form an algorithmic path forward for efficient implementation of many linear algebraic operations of scientific computing at the exascale. The combination provides asymptotically optimal computational and communication complexity and applicability to large classes of operators that commonly arise in scientific computing applications. A convergence of the mathematical theories of the fast multipole and $\mathcal{H}$-matrix methods has been underway for over a decade. We recap this mathematical unification and describe implementation aspects of a hybrid of these two compelling hierarchical algorithms on hierarchical distributed-shared memory architectures, which are likely to be the first to reach the exascale. We present a new communication complexity estimate for fast multipole methods on such architectures. We also show how the data structures and access patterns of $\mathcal{H}$-matrices for low-rank operators map onto those of fast multipole, leading to an algebraically generalized form of fast multipole that compromises none of its architecturally ideal properties.

Keywords: communication complexity, hierarchical low-rank approximation, fast multipole methods, H-matrices, sparse solvers.

1. Introduction

1.1. Exascale features of the fast multipole method

Wherever it can be applied, the fast multipole method (FMM) [14] has many appealing properties for extreme computing, beginning with its optimal computational complexity of $\mathcal{O}(N)$ for resolving the interdependence of N degrees of freedom within a specifiable error tolerance. Its computationally expensive phases have extremely high flop/s to byte/s ratios, a data locality property known as "computational intensity" [38], up to two orders of magnitude better than the conventional sparse-matrix vector multiply kernel of Krylov and other purely algebraic solvers that do not take into account the low-rank mathematical structure of the operators being approximated. Meanwhile, the dominant kernels of FMM share with the matrix-vector multiply a level of computational concurrency that scales with problem size N. Furthermore, no all-to-all communication is required in an optimal implementation of FMM in a distributed memory environment of P processes, and the $\mathcal{O}(\log P)$ messages exchanged are permitted, among themselves, a high degree of concurrency and asynchronicity. In short, fast multipole appears to be an ideal algorithm for massively distributed memory architectures with many cores sharing the local memory of a node – the dominant design for contemporary extreme scientific computing.

The low complexity of FMM is accomplished by hierarchically clustering successively distant interactions to reduce the intrinsic $\mathcal{O}(N^2)$ arithmetic and $\mathcal{O}(P^2)$ communication complexity of an explicit interaction loop in a weak scaling implementation with N/P degrees of freedom per process. A number of open-source libraries for fast multipole methods for distributed memory have been released [2] [3] [4]. Hierarchical N-body methods have been at the core of many Gordon

[1]King Abdullah University of Science and Technology (KAUST), Thuwal, 23955-6900, Saudi Arabia, {rio.yokota,george.turkiyyah,david.keyes}@kaust.edu.sa

[2]http://www.scafacos.de

[3]http://www.mrl.nyu.edu/~harper/kifmm3d/documentation/download.html

[4]https://bitbucket.org/rioyokota/exafmm-dev

Bell Prizes [19, 20, 22, 24, 29, 35–37]. Looking inward in strong scaling within a fixed memory rather than outward in weak scaling with expanding memory, they have also been effectively implemented on GPGPUs [4, 6, 10, 15, 23, 26, 27, 30, 40, 41]. However, the mathematical theory of FMM is based upon the structure of the underlying operators, be they Laplace, Helmholtz, Stokes, elasticity, etc., and is based on forming and translating expansions of the Green's function, or resolvent operator. One means to exploit the power of FMM for operators for which we do not possess Green's functions, but which are "nearby" in a spectrally equivalent sense, is to employ FMM (complemented with boundary elements as necessary to enforce finite domain boundary conditions) as a preconditioner inside a Krylov framework [42].

$\mathcal{H}$-matrix [16] theory is an alternative pathway for exploiting the increasingly low-rank structure of successively distant interactions in the context of the commonly arising constant-coefficient operators of scientific computing, and also for many others for which an explicit resolvent operator cannot be written down. There exist some quality open-source libraries for $\mathcal{H}$-matrices [5] [6] [7], but they are essentially toolboxes for algorithmic experimentation. None yet approach or are essentially in their construction motivated by the extreme architectural requirements of the exascale.

Many adaptations of current workhorse algorithms, or fresh innovations, are required if currently anticipated exascale hardware is to be used near its potential in scientific computing, since our existing code base has been engineered primarily to squeeze out as many floating point operations as possible. Instead, algorithms must now focus on squeezing out synchronizations, memory storage, and memory transfers, while extra flops on locally cacheable data represent small costs in time and energy.

Today's scalable solvers, in particular, exploit convenient global synchronizations, for which top systems provide special hardware. The all-to-all exchange of scalar data on each node that all processes wait to complete, such as an inner product of globally distributed vectors, is an endangered idiom in architectures consisting of a billion threads. Even if the work imbalance between such synchronizing steps can be bounded, which is an incompletely solved challenge in distributed adaptive computations, the processors of the future cannot be expected to be performance-reliable. For example, dynamic clocking to maintain safe levels of heat generation or dynamic correction of errors due to low signal-to-noise ratios in energy efficient hardware, will cause completion times for equal work to vary in unpredictable ways among identical processors. After decades of algorithm refinement during a period of programming model stability with bulk synchronous processing (BSP) [33], new programming models and new algorithmic capabilities (to take full advantage of the potential of exascale simulation in such areas as data assimilation, inverse problems, and uncertainty quantification) must be co-designed with the hardware.

There are numerous constraints on exascale algorithms, chief among which are minimizing memory storage and minimizing frequency of deep memory access. Roadmaps for exascale (e.g., [25]) show that energy requirements of memory accesses at increasing distance or depth will grow by orders of magnitude relative to the cost of performing a floating point operation between the operands. Further, storage will dominate the cost of acquisition of exascale hardware. Hence, algorithmic solutions are directly constrained by operating and acquisition costs to be parsimonious in memory use.

[5]http://bebendorf.ins.uni-bonn.de/AHMED.html
[6]http://www.hlibpro.com
[7]https://bitbucket.org/poulson/dmhm

For the Laplace and Laplace-like operators that commonly arise in elliptic, time-implicit parabolic, and frequency-domain hyperbolic PDEs, algebraic multigrid (AMG) scales efficiently in proportion to available memory on hundreds of thousands of rigidly schedulable, tightly coupled cores for systems with billions of unknowns [2]. However, for constant coefficient problems the fast multipole method (FMM) is asymptotically superior in complexity and tolerates less synchronization. Algebraic fast multipole (AFM) based on hierarchical matrix decompositions may extend the performance robustness of FMM to the coefficient variability of AMG and improve upon memory requirements, and therefore exemplifies the adaptations that many algorithms must make.

In Section 1.2, we recall a hierarchical feature of contemporary architecture and accompanying constraints that motivate wider exploitation of FMM. In section 1.3, we recall the advantages of several hierarchical algorithms and we motivate interest in an algebraic variant of fast multipole. Section 2 presents a new result on FMM communication complexity with respect to three types of distributions that are illustrative of applications in 3D. Section 3 describes the $\mathcal{H}^2$ formulation of $\mathcal{H}$-matrices and highlights how its data structures and communication patterns map onto those of the FMM and how the matrix decomposition is formed. In Section 4, we conclude this work in progress with a consideration of open questions.

1.2. Architectural implications for exascale algorithms

Due to the leveling of clock speeds of CMOS processors primarily for reasons of energy consumption and concomitant heat dissipation, the first CMOS-based exascale machines will consist of approximately one billion threads executing at about 1 GigaHertz. They will likely be configured approximately as one million nodes of one thousand cores each: a GigaHertz-MegaNode-KiloCore machine. Perhaps the number of cores per fixed memory node will not exceed one hundred, in which case ten million nodes will be required. Weak scaling has proved to be an easier paradigm than strong scaling to meet efficiently because extra memory and extra memory bandwidth come proportionally with the increase in cores. The high cost of this memory is one of the main pressures in favor of manycore and GPGPU architectures. Forecasts of the architecture of exascale may be found in [25] and [9]. Under any form of scaling, algorithms are favored that reduce the per flop requirements of storage and memory bandwidth, and increase the uniformity and predictability of the flops that are ultimately executed.

Bad and good news simultaneously lurk behind every aspect of the evolution of exascale architecture. Programmers will have to explicitly control more of the data motion, since it carries the highest energy cost in the computational environment. However, they will be given tools that some have craved to better control the vertical (replication-oriented) data motion. The horizontal (interprocessor) data motion has been successfully under programmer control in the form of message passing for over 25 years. Vertical replication has generally been handled beneath the level of programmer concern. However, recently with the rolling of data in and out of accelerators, programmers have embraced this control, even if they bemoan the slow transmission speed of today's channels. Authors of performance-seeking tree-based codes are already in the habit of managing their data in great detail, to the extent that dynamic runtime systems may have little to improve upon [1].

Today's optimal algorithms have evolved, as mentioned above, to squeeze out flops, whereas the premium at the exascale is to squeeze out memory storage and accesses. However, whereas storage-optimal methods may not be computationally optimal, computationally optimal methods tend to be storage-optimal because the amount of relevant data cannot be greater than that touched in executing the flops. Therefore, computationally optimal algorithms are the first places to look for memory parsimonious exascale kernels.

Other drivers of exascale algorithmic innovation, generally, include exploitation of adaptive precision, as a means of reducing storage, and algorithmic-based fault tolerance, including detecting and handling errors within the user application, rather than making fault tolerance a hardware responsibility. In some sense, optimal fast multipole and $\mathcal{H}$-matrix formulations control accuracy in a way that reduces the opportunity and the pressure for adaptive precision – very few bytes are wasted by mathematical overresolution compared to other formulations of the same underlying continuous mathematics. As for algorithmic-based fault tolerance (ABFT), FMM and $\mathcal{H}$-matrices may lack an advantage of multigrid and Krylov solvers, with their fresh periodic computations of residuals to drive corrections in the latter. However, ABFT for FMM and $\mathcal{H}$-matrices is in its infancy. Today's hardware can still be treated as reliable, and among today's optimal hierarchical methods, FMM and $\mathcal{H}$-matrices excel in concurrency, tolerance of asynchronicity, arithmetic intensity, and the ability to tune SIMT data sizes to natural boundaries in the hardware hierarchy.

1.3. Hierarchical algorithms

A diverse collection of algorithms with optimal arithmetic complexity – linear or at most log-linear in the number of degrees of freedom – share the key characteristic of being hierarchical. We list them here in their simplest forms; each one is, of course, the subject of decades of research that fill books.

The fast Fourier transform (FFT) replaces $\mathcal{O}(N^2)$ multiply-add operations with $\mathcal{O}(N \log N)$ with a constant as low as 5 for favorable radix. Geometric multigrid for solving the Poisson Dirichlet on a uniform 3D grid with second-order finite differences replaces $\mathcal{O}(N^{7/3})$ multiply-add operations for a Gaussian band solver on a naturally ordered version of the problem with $\mathcal{O}(N \log N)$ with a constant as low as about 6 for solution to truncation error. Sparse grids represent sufficiently smooth functions on a d-dimensional of resolution N on a side, which would require N^d to store, with $\mathcal{O}(N(\log N)^{d-1})$ storage at a cost in accuracy that is only logarithmically degraded. Like the FFT and multigrid, the "combination form" of sparse grids is able to work with data structures at each level of the hierarchy that have the same simple Cartesian structure of the original. All three algorithms therefore generate a logarithmic number of hierarchically coarsened versions of the original. (Multigrid may require a special solve for the coarsest grid.)

Fast multipole reduces an $\mathcal{O}(N^2)$ summation to an $\mathcal{O}(N \log N)$ or $\mathcal{O}(N)$ complexity through a hierarchy of tree-based operations by distinguishing between near interactions that must be treated directly and recursively coarsened far interactions. Its complexity of implementation, even in serial, is therefore somewhat greater than the hierarchical methods above, but like the others, the principal feature is the recursive generation of a self-similar collection of problems. The $\mathcal{H}$-matrix format is suitable for a variety of operations, including matrix-vector multiplication, matrix inversion, matrix-matrix multiplication, matrix-matrix addition, etc., and has recently been generalized to tensors. In its classical $\mathcal{H}$ and $\mathcal{H}^2$ forms, the asymptotic complexity $\mathcal{O}(N^2)$ of a matrix-vector multiply is reduced to $\mathcal{O}(N \log N)$ or $\mathcal{O}(N)$, respectively. Fast Multipole Methods (FMM) and $\mathcal{H}$-matrices share many common features that arise from their hierarchical nature. The former is geometric while the latter is algebraic, but the two methods have similar work-flow and data structures. Many parallelization techniques that have been developed for the FMM can be applied to its algebraic variant. The matrix-free nature of FMM and the larger prefactor of its computational complexity are the sources of its strong arithmetic intensity. Being matrix-free also reduces the amount of communication.

Other methods deserve to be mentioned in this context even if they are not strictly optimal by the definition above. Nested dissection ordering and its graph partitioning generalization, when applied to algebraic systems generated from local PDE discretizations, create a recursive sequence of separators of decreasing size and leave behind independent blocks. Employing low-rank representations for the separator blocks preserves sparsity and lowers the overall factorization costs both in memory and flops. The dense Schur complements that arise recursively in many initially sparse discretizations are amenable to low-rank decompositions. For elliptic problems, large amounts of compression are possible and the resulting factorization can be used as a nearly optimal-order direct solver [39].

These hierarchical algorithms are vital to the optimal performance of many scientific codes, as kernels in simulations based on formulations of partial differential equations, integral equations, and interacting particles. Successfully migrating these kernels to the computational environment of the exascale will create a path that many full applications can follow, just as a generation earlier, dense and sparse linear algebra libraries led applications into efficient use of distributed memory and message passing.

2. Communication Complexity of Fast Multipole Methods

Communication becomes the bottleneck for any algorithm as it approaches the limit of its parallel scalability. Therefore, communication complexity is what distinguishes algorithms that scale from ones that do not. It is well known that FMM has $\mathcal{O}(N)$ *arithmetic* complexity, but relatively little attention has been given to its *communication* complexity. In this section we provide new upper bounds for the communication complexity of FMM, first for the uniform case. We then extend the complexity analysis to the nonuniform case and prove that the same upper bound holds. We define N as the sum of the number of particles on all processes and P as the number of processes. These are the only two variables in the current analysis of the communication complexity. We do not consider the communication during the initial partitioning phase in the present analysis.

Table 1. Communication complexity of FMM.

Reference	Processes		Data per Process		Communication complexity
Teng [32]	$\mathcal{O}(P)$		$\mathcal{O}\left((N/P)^{2/3}(\log N+\mu)^{1/3}\right)$		$\mathcal{O}\left(P(N/P)^{2/3}(\log N+\mu)^{1/3}\right)$
Lashuk *et al.* [27]	$\mathcal{O}(\sqrt{P})$		$\mathcal{O}\left((N/P)^{2/3}\right)$		$\mathcal{O}\left(\sqrt{P}(N/P)^{2/3}\right)$
Ibeid *et al.* [21]	Global	Local	Global	Local	Global + Local
	$\mathcal{O}(\log P)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}\left((N/P)^{2/3}\right)$	$\mathcal{O}\left(\log P+(N/P)^{2/3}\right)$

2.1. Uniform distribution

Analysis of communication complexity of FMM has been performed previously by Teng [32], Lashuk *et al.* [27], and Ibeid *et al.* [21]. The communication complexity of these different studies is shown in tab. 1. "Processes" is the number of processes involved in the communication. Teng assumes a nonuniform distribution, while the other two focus on uniform distributions, but this is not what causes the difference in the complexity. We will show in section 2.2 that the tightest upper bound of Ibeid *et al.* still holds for the nonuniform case. In this section we will first prove the upper bounds for the uniform case, while describing the reason for the discrepancies between the three. The three papers use different terminology so we provide tab. 2 to map the correspondence.

There are two improvements between the communication complexity of Teng $\mathcal{O}\left(P(N/P)^{2/3}(\log N+\mu)^{1/3}\right)$ and Lashuk *et al.* $\mathcal{O}\left(\sqrt{P}(N/P)^{2/3}\right)$. The first is P changing to $\sqrt{P}$ and the second is the $(\log N+\mu)^{1/3}$ disappearing. The P changing to $\sqrt{P}$ is due to $\sum_i^{\log P-1} 2^i = \mathcal{O}(P)$ on the bottom of page 650 in Teng [32] changing to $\sum_i^{\log P-1} min(2^{\log P-i-1},2^i)=\mathcal{O}(\sqrt{P})$ on the right side of page 7 in Lashuk *et al.* [27]. This improvement is made possible by the hypercube reduce and scatter scheme shown in Algorithm 3 in the same paper, where processes are paired in a $(\log P)$-dimensional hypercube. The $(\log N+\mu)^{1/3}$ factor in Teng stems from the proof of Lemma 4.8 where he assumes that there could be $\mathcal{O}(\log N+\mu)$ neighbors in the near-field graph if a highly refined leaf box existed next to a large leaf box. Such cases do not exist for a uniform distribution so it is easy to prove that this factor disappears in this case. Furthermore, we will show in section 2.2 that it is still possible to bound the neighbors in the near-field list to $\mathcal{O}(1)$ for a nonuniform distribution by using a 2:1 refinement constraint [31] during the tree construction.

We now focus on the two discrepancies between the communication complexity of Lashuk *et al.* $\mathcal{O}\left(\sqrt{P}(N/P)^{2/3}\right)$ and Ibeid *et al.* $\mathcal{O}\left(\log P+(N/P)^{2/3}\right)$. The first difference is $\sqrt{P}$ becoming $\log P$ and the second is the product changing to a sum. The change from a product to a sum comes from the separation of the global tree from the local tree as shown in fig. 1 and tab. 1. The global tree is formed from the hierarchical grouping of processes, which has P leaf nodes and a $\mathcal{O}(\log P)$ depth. The local tree is formed from local particles that belong to the process and has $\mathcal{O}(\log(N/P))$ depth. For a uniform distribution with a full tree and when the number of processes is a power of two, it is very easy to see that the leaf of the global tree is the root of the local tree as shown in fig. 1. We will show in section 2.2 that this separation between the global tree and local tree is still possible for nonuniform distributions as well.

Table 2. Correspondence of terminology.

Teng [32]	Lashuk *et al.* [27]	Ibeid *et al.* [21]
near-field graph	U-list	P2P-list
far-field graph	V-list	M2L-list
box	box	cell

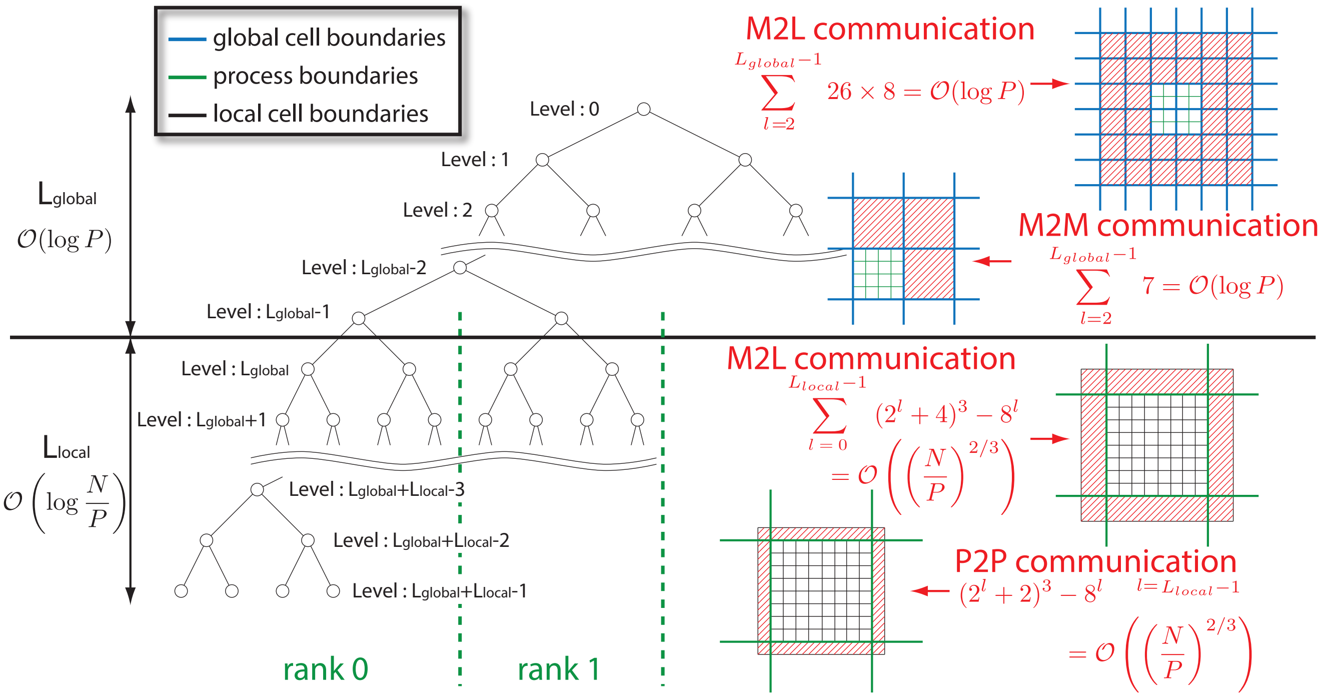

global cell boundaries
process boundaries
local cell boundaries
Level : 0
Level : 1
Level : 2
Level : Lglobal-2
Level : Lglobal-1
Level : Lglobal
Level : Lglobal+1
Level : Lglobal+Llocal-3
Level : Lglobal+Llocal-2
Level : Lglobal+Llocal-1
Lglobal
$\mathcal{O}(\log P)$
Llocal
$\mathcal{O}\left(\log \frac{N}{P}\right)$
rank 0
rank 1
M2L communication
$\sum_{l=2}^{L_{global}-1} 26 \times 8 = \mathcal{O}(\log P)$
M2M communication
$\sum_{l=2}^{L_{global}-1} 7 = \mathcal{O}(\log P)$
M2L communication
$\sum_{l=0}^{L_{local}-1} (2^l+4)^3 - 8^l = \mathcal{O}\left(\left(\frac{N}{P}\right)^{2/3}\right)$
P2P communication
$(2^l+2)^3 - 8^l \quad l = L_{local}-1 = \mathcal{O}\left(\left(\frac{N}{P}\right)^{2/3}\right)$

Figure 1. Communication patterns and complexities of FMM.

We separate the global tree from the local tree because they have different communication complexity as shown in tab. 3 and also fig. 1. "Processes" is the total number of processes to communicate with during a given phase. In the global tree, one octree cell is owned by many processes, whereas in the local tree many octree cells are owned by a single process. Therefore, the redundancy of the information of octree cells in the global tree increases exponentially as the level gets coarser. Exploiting this redundancy and getting rid of the all-to-all type communication pattern is what brings the $\sqrt{P}$ in Lashuk *et al.* [27] down to $\log P$ in Ibeid *et al.* [21]. The ability to remove the all-to-all type communication altogether may seem counterintuitive since information from every process *is* required by every other process for the construction of the local essential tree. The key is to use the reduction in the M2M operation and the redundancy at the coarse levels to limit the number of processes that need to communicate directly.

As an example, let us first consider the communication phase for the M2M operation in the global tree shown in fig. 1. At the leaf nodes of the global tree one local root cell exists on

Table 3. Breakdown of communication in Ibeid *et al.* [21].

	Processes	Cells per level	Cells per Process	Communication
Global M2L	$\sum_i^{\log P} 26$	26×8	8	$\mathcal{O}(\log P)$
Global M2M	$\sum_i^{\log P} 7$	7	1	$\mathcal{O}(\log P)$
Local M2L	26	$(2^i+4)^3 - 8^i$	$\sum_i^{\log_8(N/P)} (2^i+4)^3 - 8^i$	$\mathcal{O}((N/P)^{2/3})$
Local P2P	26	$(2^i+2)^3 - 8^i$	$(2^{\log_8(N/P)}+2)^3 - 8^{\log_8(N/P)}$	$\mathcal{O}((N/P)^{2/3})$

each process, and the M2M operation will take each cell from the neighboring 8 processes and reduce this information to the parent node. This requires communication with the 7 neighboring processes. The tree structure shown in fig. 1 is a binary tree, which would be an octree in 3D. Similarly, the illustrations of the geometric partitions in fig. 1 are in 2-D but we are actually considering a 3-D octree. After the M2M operation at the leaf node (level L_{global}) of the global tree, the neighboring 8 processes have the same information about the node at level $L_{global} - 1$. Therefore, when performing the M2M operation at the next level, one only needs to communicate with 7 processes instead of 7×8 due to the 8-fold redundancy. If we apply this logic recursively, we see that the M2M communication will always require communication with 7 processes. The 7 processes to communicate with will move farther and farther away as we go up the tree, so we must sum up the number of processes at each level to get to total number of processes to communicate with. There are $\mathcal{O}(\log P)$ levels in the global tree and there is always one cell worth of data that is being sent, so the communication complexity of the global M2M phase is $\mathcal{O}(\log P)$ as shown in tab. 3.

Similar logic can be applied to the communication for the global M2L operations. Even though the number of octree cells contained in each process grows exponentially at coarser levels of the global tree, the redundancy of information also increases at the same exponential rate if we use the M2M communication described above. Therefore, the *unique* information that must be communicated during the M2L phase remains constant at each level of the global tree. For the standard definition of neighbors, the M2L phase requires two cells worth of a halo region to be communicated. This results in 26×8 cells to be communicated from 26×8 neighboring processes. However, the 8 child cells that arrived during the M2M communication can be used to reduce the number of processes to communicate with to 26, while each process now sends 8 cell's worth of data as shown in tab. 3. Similar to the M2M phase, there are $\sum_{i}^{\log P} 26$ processes to communicate with in total. Therefore, the communication complexity of the global M2L phase is $\mathcal{O}(\log P)$.

For the local M2L and P2P phases, it is only necessary to communicate with the same 26 neighboring processes regardless of the level, so the number of processes to communicate with is $\mathcal{O}(1)$ instead of $\mathcal{O}(\log P)$. However, unlike the global tree communication, the local tree communication requires more cells to be sent as the level of the local tree gets finer. The M2L phase has two cells worth of a halo region and the P2P phase has one cell worth of a halo region that needs to be communicated, so the width of these halos are constant but their lengths are not. For the M2L halo, having two cells on each side adds four cells per dimension so we have $(2^i + 4)^3 - 8^i$ cells to send at the i-th level. For the P2P halo, we have one cell on each side so the number is $(2^i + 2)^3 - 8^i$. The halo size is basically the surface to volume ratio, which means that the complexity is $\mathcal{O}(N/P)^{2/3}$. It can also be seen from tab. 3 that summing for powers of four up to a base eight logarithm gives $\sum_{i}^{\log_8(N/P)} 4^i = (N/P)^{\log 4/\log 8} = (N/P)^{2/3}$.

Looking back at tab. 1, we now see that it is the separation of the global tree from the local tree that turns the product in $\mathcal{O}(\sqrt{P}(N/P)^{2/3})$ of Lashuk *et al.* to the sum in $\mathcal{O}(\log P + (N/P)^{2/3})$. In other words, there are a $\mathcal{O}(1)$ number of neighboring processes that require $\mathcal{O}(N/P)^{2/3}$ data during the local P2P and M2L communications. For the global communications the amount of data to send per process pair is $\mathcal{O}(1)$ but there are a constant number of processes to communicate with that are different at each level of the global tree, which has a depth of $\mathcal{O}(\log P)$. This is a significant improvement on the upper bound of the communication complexity of FMM with respect to Teng [32] and Lashuk *et al.* [27]. We will extend this analysis for uniform distributions to nonuniform distributions in the following subsection.

2.2. Nonuniform distribution

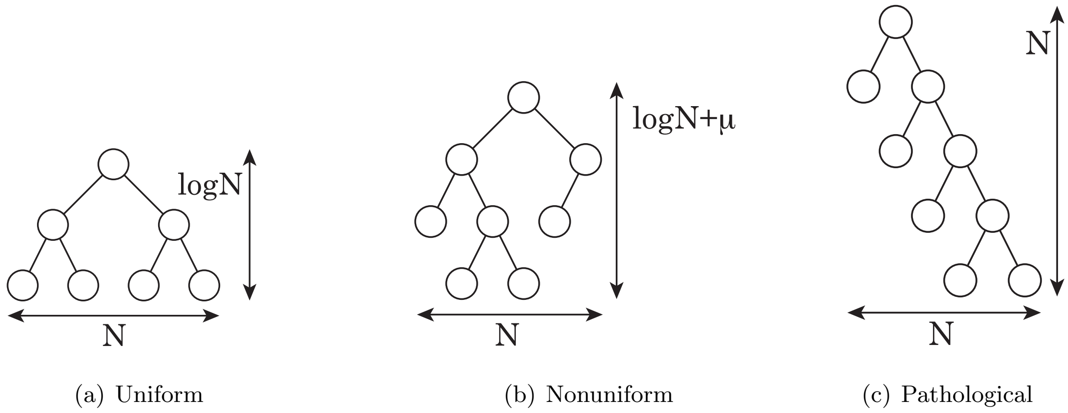


Figure 2. Refinement distributions reflected in tree data structures.

The complexity analysis for the uniform distribution with a full octree can be extended to the nonuniform distribution with an adaptive tree. Before discussing the extension to nonuniform distributions, we first define the type of nonuniform distribution in which we are interested. For this purpose, three different types of tree structures are shown in fig. 2. The full tree that corresponds to uniform distribution as discussed in the previous subsection is shown in fig. 2(a). The adaptive tree that results from a μ-nonuniform distribution [32] is shown in fig. 2(b). We will discuss the μ-nonuniform distribution in this subsection. The μ-nonuniform distribution is in contrast to a pathological nonuniform distribution that results in a $\mathcal{O}(N)$ depth tree shown in fig. 2(c).

When considering nonuniform distributions, it is necessary to exclude pathological cases like the one in fig. 2(c). Fortunately, such a distribution will not occur in practice since it corresponds to exponentially increasing spatial refinement throughout the entire domain. One could accidentally produce such a distribution by using a naive adaptive mesh refinement near a singularity, but this is obviously not a sound technique for the numerical integration of singular functions. Furthermore, for such pathological cases the FMM will reduce to a direct N-body method since the depth of the tree becomes $\mathcal{O}(N)$. At every level of the tree, all cells are direct neighbors of each other so the far-field approximation is not valid between any of them. Therefore, all pairs of particles will be calculated by direct summation in this case. Hence, the hypothesis for the arithmetic complexity of FMM being $\mathcal{O}(N)$ excludes such cases to begin with, so they will not be considered in the current communication complexity analysis.

The communication complexity for the uniform case is extended to the nonuniform case by adopting the definition of μ-nonuniform distributions [32], which is depicted in fig. 2(b). For μ-nonuniform distributions, the depth of the tree is still $\mathcal{O}(\log N)$ but has a constant number of additional levels that come from the nonuniformity. In fig. 3 we show the depth of the tree as a function of the number of particles N for three types of distributions. "random" is a random distribution of particles in a cube, which is representative of the distribution of atoms in a molecular dynamics simulation. "surface" has points only on the surface of a sphere and is

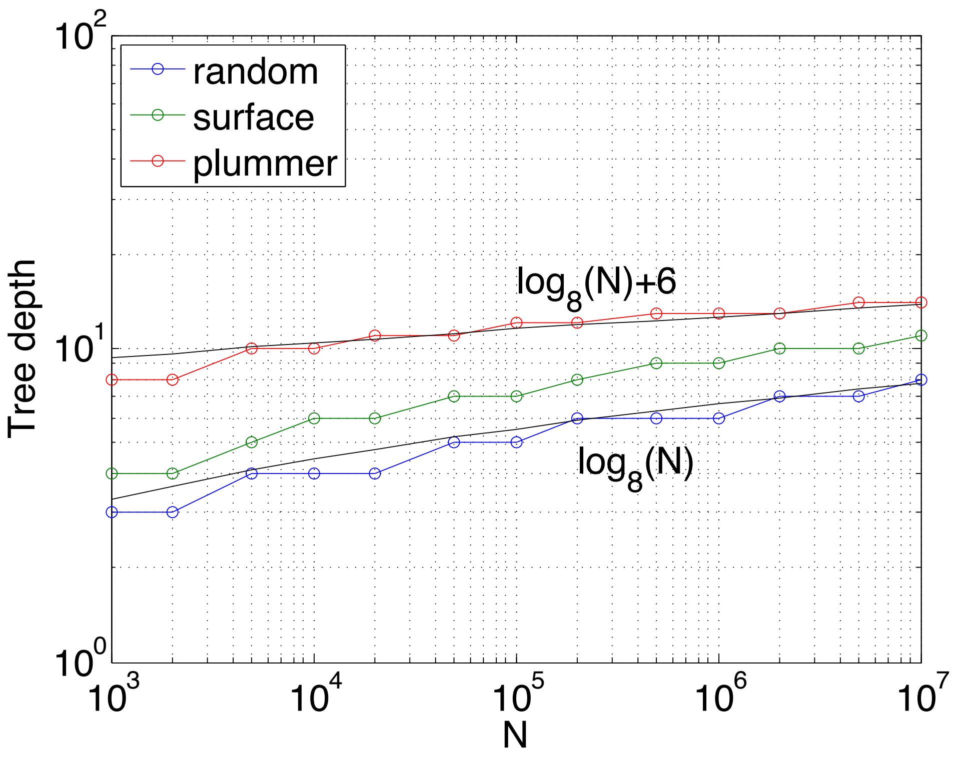


Figure 3. Tree depth as a function of number of particles N for different distributions. "random" is a random distribution of particles in a cube, "surface" has points only on the surface of a sphere, and "Plummer" [28] has high concentration of particles in the center of the domain. They all exhibit $\mathcal{O}(\log N)$ behavior but with a different constant. The maximum number of particles per leaf cell was set to 16 for these plots.

representative of a boundary integral calculation. "Plummer" has high concentration of particles in the center of the domain, which is common in cosmological simulations. The "random" distribution results in an almost full tree and is the most uniform among the three. The "surface" distribution has inter-particle spacing that grows as $\mathcal{O}(N^{\frac{1}{d-1}})$ for a d dimensional simulation. Therefore, it still has $\mathcal{O}(\log N)$ depth but with a different constant. The "Plummer" distribution also has $\mathcal{O}(\log N)$ depth but with an even larger constant. FMM applications can be categorized into either of these three types of distributions, so we will assume the μ-nonuniform distribution for the following analysis of communication complexity of FMM.

The maximum depth of the tree has further constraints that stem from the finite precision in numerical simulations. The first limit comes at 22 levels of an octree, where the 64-bit unsigned integer will overflow $2^{64} < 8^{22}$. It is possible to build deeper trees by using multiple integers to store the Morton/Hilbert keys. The next limit occurs at 53 levels, where the number of significand (mantissa) bits will not be enough to distinguish the two points expressed in double precision floating point numbers. Say for example, we have two particles with coordinates $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ with a distance $|\mathbf{x}_{i+1} - \mathbf{x}_i|/|\mathbf{x}_i| < 1/2^{53}$. The double precision floating point value for the coordinates of these two particles will be identical because the difference will be in the 54th mantissa bit, which does not exist. Therefore, even if we use multiple 64-bit integers to store the Morton/Hilbert key, it would not be possible to build a tree structure with over 53 levels, because particles are indistinguishable past that level and cannot be correctly binned into to smaller sub-cells. Actually, the round-off error in the P2P kernel will become unacceptable far before we reach this level of refinement.

With this definition of μ-nonuniform distributions, we revisit the differences between the communication complexity of Teng [32], Lashuk *et al.* [27], and Ibeid *et al.* [21] shown in tab. 1, and see if they are valid for the μ-nonuniform case. The first difference between Teng and Lashuk *et al.* is the change from P to $\sqrt{P}$. There is no assumption of uniformity in the hypercube reduce and scatter communication that yields the $\sqrt{P}$ factor, so this communication scheme should be directly applicable to Teng's analysis, which will change P to $\sqrt{P}$. We have mentioned in section 2.1 that the $(\log N + \mu)^{1/3}$ factor comes from the assumption that there could be $\mathcal{O}(\log N + \mu)$ neighbors in the near-field graph if a highly refined leaf box existed next to a large leaf box. The far-field graph does not contribute to the $\mathcal{O}(\log N + \mu)$ factor because the proof of Lemma 4.8 in Teng [32] shows that all non-leaf-boxes have a in-degree bounded by a constant. Therefore, all we need to prove is that the near-field graph can be bounded by a constant for any μ-nonuniform distribution in order to extend the communication complexity of Lashuk *et al.* to a μ-nonuniform case. The 2:1 balance refinement of octrees by Sundar *et al.* [31] will yield precisely such a $\mathcal{O}(1)$ bound on the near-field graph. Therefore, the communication complexity $\mathcal{O}\left(\sqrt{P}(N/P)^{2/3}\right)$ of Lashuk *et al.* is valid for μ-nonuniform distributions if the hypercube reduce and scatter communication and 2:1 refinement are used.

The next task is to prove that the change from $\mathcal{O}\left(\sqrt{P}(N/P)^{2/3}\right)$ to $\mathcal{O}(\log P + (N/P)^{2/3})$ is valid for μ-nonuniform distributions. For a μ-nonuniform distribution, most of the constants in tab. 3 will change, but they will still be $\mathcal{O}(1)$ and will not change the overall communication complexity as shown in tab. 4. The global tree and local tree can still be separated during the analysis for the μ-nonuniform case. Teng [32] proved that it is always possible to form a P-way partition of a μ-nonuniform distribution with $\mathcal{O}(N/P)$ particles each. This means the depth of the global tree is always $\mathcal{O}(\log P)$ even for the μ-nonuniform case. From Lemma 4.8 in Teng [32] we know that the non-leaf-boxes have a in-degree bounded by a constant. Therefore, the number of cells per level for the global communication is $\mathcal{O}(1)$, and so are the number of processes to communicate with per level. This means that the communication complexity of both global M2L and global M2M phases is $\mathcal{O}(\log P)$.

In the uniform case, the local communications had a halo width of one and two cells for the P2P and M2L phases, respectively. In the general case, this depends on the multipole acceptance criteria (the definition of well-separatedness), but is bounded by a constant even

Table 4. Breakdown of communication for the μ-nonuniform case.

	Processes	Cells per level	Cells per Process	Communication
Global M2L	$\sum_{i}^{\log P} \mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log P)$
Global M2M	$\sum_{i}^{\log P} \mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log P)$
Local M2L	$\mathcal{O}(1)$	$\mathcal{O}(4^i)$	$\sum_{i}^{\log_8(N/P)} \mathcal{O}(4^i)$	$\mathcal{O}((N/P)^{2/3})$
Local P2P	$\mathcal{O}(1)$	$\mathcal{O}(4^i)$	$\mathcal{O}(4^{\log_8(N/P)})$	$\mathcal{O}((N/P)^{2/3})$

for μ-nonuniform distributions. Again, we may use Lemma 4.8 in Teng [32] as a proof for this bound. Therefore, as long as the halo width is $\mathcal{O}(1)$ we can make the same surface to volume ratio argument to get a $\mathcal{O}(N/P)^{2/3}$ complexity as shown in tab. 4. In conclusion, all the upper bounds on the communication complexity of the global M2L and M2M and the local M2L and P2P phases are valid for the μ-nonuniform case. Therefore, the adaptive FMM with μ-nonuniform distribution has a communication complexity of $\mathcal{O}(\log P + (N/P)^{2/3})$.

3. Hierarchical Matrices as Algebraic Variants of FMM

As mentioned in the introduction, hierarchical matrix representations of different flavors have been proposed in the last decade or so to exploit the underlying low rank hierarchical structure of matrices that appear in broad classes of applications. Hackbusch [16, 18] pioneered the concepts of hierarchical matrices in the form of $\mathcal{H}$ and $\mathcal{H}^2$ matrices and developed a substantial mathematical theory for their ability to approximate integral operators and boundary value problems of elliptic PDEs. There ideas have been developed considerably over the years, for the construction and use of hierarchical matrices in solving discretized integral equations and preconditioning finite element discretizations of PDEs [3, 5, 12, 13, 17]. Hierarchically semi-separable (HSS) and hierarchically block-separable (HBR) are related and well-studied representations that also use low rank blocks of a dense matrix in a hierarchical fashion. The concept of a semi-separable matrix originated from matrices associated with separable kernels allowing their low rank representation as outer products of two thin matrices [34]. Matrices are semi-separable if their upper and lower triangular parts are, each, part of a low rank matrix. HSS matrices refer to matrices whose off-diagonal blocks are numerically of low rank and whose block structure consists of blocks that grow geometrically in size with their distance from the main diagonal. Fast factorization algorithms for HSS matrices have been developed in [7, 39]. HBR matrices have a similar structure but emphasize the telescoping nature of the matrix factorization [11] to use in the construction of direct solvers for integral equations [8].

A particular variant of hierarchical matrices, $\mathcal{H}^2$, has many similarities with Fast Multiple representations and can therefore benefit from the substantial algorithmic developments of FMM and in particular the $\mathcal{O}((N/P)^{2/3})$ communication complexity on distributed memory machines as derived in section 2. In this section, we describe these similarities by showing how the matrix-vector multiplication operation in the hierarchical format maps to the various computational kernels of the FMM.

3.1. Representation

Fig. 4 depicts the elements of an algebraic $\mathcal{H}^2$ representation of a dense matrix A. The representation consists of:

- a tree I that organizes the row indices of the matrix hierarchically. We use a binary tree in the illustration in fig. 4 but a quadtree, or an octree, or another application-specific organization is possible. In particular, an octree is natural when the matrix is a discretization of a volume integral operator. The tree represents row blocks of the matrix. It is used to store column basis vectors in which to express the data of the various matrix blocks. The thin basis matrices U are stored explicitly at its leaves, and small interlevel transfer matrices E are stored at the higher levels and used to generate the level-appropriate bases as described below.

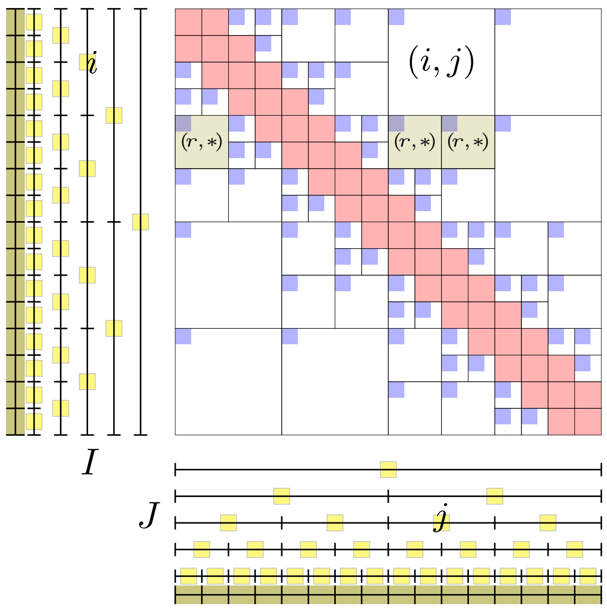


Figure 4. Hierarchical compressed representation of a dense matrix. Matrix blocks A_{ij} are compressed by expressing them in factorized form $U_i S_{ij} V_j^t$.

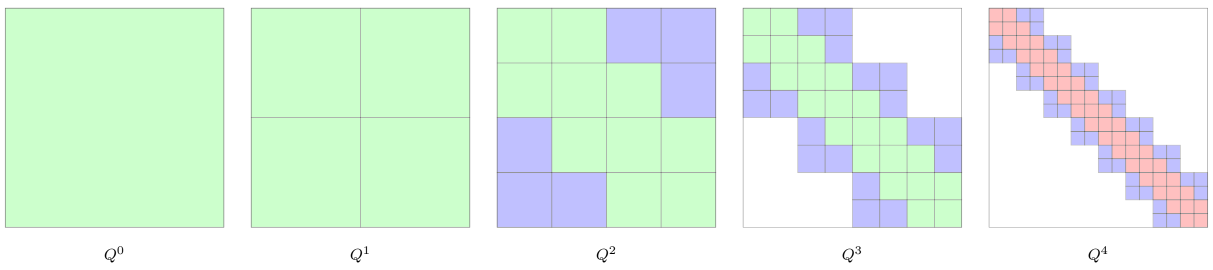


Figure 5. Quadtree representation of matrix structure. Leaves of the quadtree represent disjoint matrix blocks that collectively cover the complete matrix.

- a tree J that organizes the column indices of the matrix hierarchically. As with I, J encodes possible block partitionings of the columns of A. It stores row basis vectors V for the data of the matrix A. The basis matrices are stored at its leaves and interlevel transfer matrices F are stored at the higher levels and used to generate dynamically the level-appropriate row basis.
- a quadtree Q whose nodes are indexed by two indices $i \in I$ and $j \in J$. The (i, j) pairs at the leaves of this incomplete quadtree (fig. 5) form a hierarchical partitioning of the matrix. Collectively, the leaves cover the whole matrix A. Some of the leaves are labelled as low rank leaves and represent the blocks of A that can be approximated to the desired tolerance by a low rank factorization $U_i S_{ij} V_j^t$. These leaves of the quadtree store the small matrices S_{ij}, which may interpreted as the projections of the A_{ij} blocks on the corresponding U_i and V_j bases.
- a set of dense $b \times b$ matrices that are not compressed. The complement of the low rank leaves of Q represents blocks of A that are not economically expressible as low rank factorizations and are more effectively stored in their original format. These leaves appear only at the lowest level of the quadtree Q and represent blocks of sizes $b \times b$, where b is tuned to size of the cache memories of the target hardware. In fig. 4 these dense blocks, for clarity of illustration, are shown along a diagonal band only but these blocks may appear anywhere in the matrix.

This representation of a matrix reduces the needed storage to $\mathcal{O}(N)$. This is due to the fact that all the blocks in a block row share a common column basis, and this basis is itself nested and can be expressed hierarchically. This is illustrated in fig. 4 where block row r (shown as shaded) contains three blocks all sharing the same U_r in their respective individual $U_r S_{rj} V_j^t$ representations. The index r of this row block is not an index of a leaf node in the tree I however; its column basis U is not stored explicitly but is expressible in terms of the column bases of its children. Fig. 6 shows the hierarchical structure of the column space.

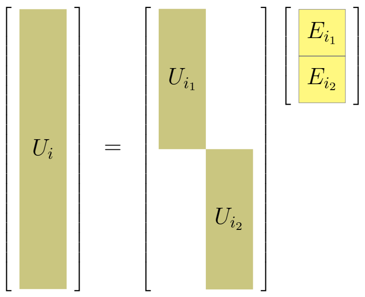


Figure 6. Hierarchical representation of column basis vectors of node i in terms of its children. Low rank matrix blocks are expressed in terms of the level-appropriate basis vectors. A similar hierarchy exists for the row bases.

The growth of the storage requirements of this hierarchical representation can be estimated from the representations needed for storing the row and column bases and those needed for storing the leaves of the quadtree. Let k be the rank used in the approximation of the $A_{ij} = U_i S_{ij} V_j^t$ blocks and assume for simplicity that it is constant across all blocks. The column bases requires $\mathcal{O}(Nk)$ units of storage at the leaves and $\mathcal{O}(Nk^2)$ for the transfer matrices The number of leaves of the quadtree is linear in N assuming that the size of every block row at every level on the quadtree is bounded by some constant. Since every S_{ij} matrix is of size $k \times k$, the storage of the quadtree requires $\mathcal{O}(Nk^2)$ and therefore all the elements of the representation require storage that grows only linearly with matrix size.

3.2. Matrix-Vector Multiplication with Hierarchical Matrices

A matrix-vector operation with a hierarchical matrix can be performed in $O(N)$ arithmetic operations, and can use the same communication infrastructure of FMM. This may be seen by expressing the product as:

$$y = \left(\sum_{(i,j)\in D} A_{ij} \right) x + \left(\sum_{(i,j)\in L} U_i S_{ij} V_j^t \right) x = \underbrace{\sum_{(i,j)\in D} A_{ij} x_j}_{\text{Dense mat-vecs operations}} + \sum_{i\in I} \underbrace{U_i \underbrace{\sum_{(i,j)\in L} S_{ij} \underbrace{V_j^t x}_{\text{Upsweep}}}_{\text{Coupling phase}}}_{\text{Downsweep}}$$

where D is the set of dense leaves of the quadtree and L is the set of its low rank leaves.

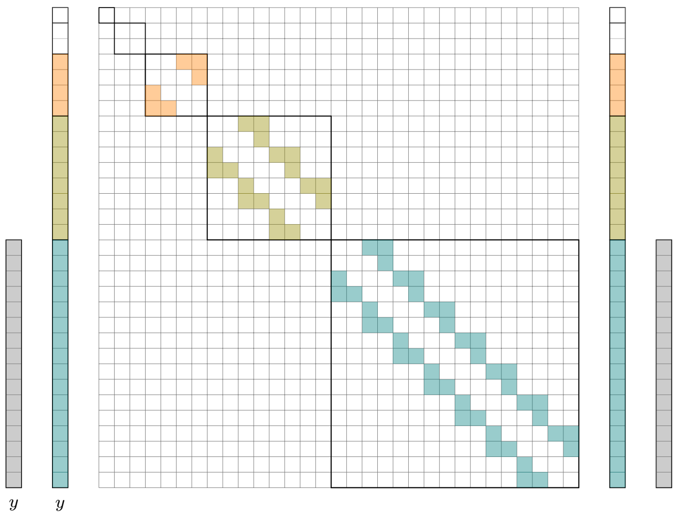


Figure 7. Low rank portion of the matrix-vector operation.

The dense product portion consists of multiplications of $b \times b$ matrices with corresponding vectors, while the low rank portion of the operation can be separated into three computational kernels:

Upsweep. In this phase, the products $V_j^t x$ are computed for all $j \in J$. Since the bases V_j are stored explicitly only at the leaves and the bases at interior nodes are expressible in terms of $k \times k$ transfer operators, this computation may be performed as follows:

- at the leaves: $\hat{x}_j = V_j^t x$
- at the interior nodes: $\hat{x}_j = \sum F_k^t \hat{x}_k$

The result of the upsweep operations is a set of vectors of size $k \times 1$ at each node of the tree J representing the matrix columns. This set of vectors is denoted by $\hat{x}$ and is shown (in a linearized fashion) in fig. 7.

Coupling operations. In this phase, the products $S_{ij}\hat{x}_j$ are performed and accumulated as $\hat{y}_i$ in the corresponding nodes of the I tree. This is illustrated in fig. 7 where the three colors are used to distinguish the S_{ij} corresponding to the three levels Q^2, Q^3, and Q^4 of the matrix in fig. 5. This phase may be viewed as a set of block-sparse matrix-vector multiplications, one per level of the tree. The various levels of may be processed concurrently, and the block sizes in these block-sparse matrices are $k \times k$.

Downsweep. In this final portion of the computation the products $U_i\hat{y}_i$ are performed and assembled in their corresponding positions in the final vector y. Since the bases U_i are available in explicit form only at the leaves, all the products that correspond to interior nodes have to be computed by going down the I tree. Starting the root, contributions to the children of every interior node are accumulated and a direct computation is performed at the leaves. The computational pattern is dual to that of the upsweep phase:

- at interior nodes: $\hat{y}_k += E_k\hat{y}_i$ for every child k of node i
- at leaf nodes: $y = U_i\hat{y}_i$

The analogy to the FMM computations and communication patterns is partially summarized in tab. 5. The upsweep phase in the hierarchical matrix-vector multiplication corresponds to building and propagating multipole expansions (P2M and M2M) up the tree. The block multiplications of the coupling phase correspond to computing M2L interactions. The downsweep phase corresponds to propagating local expansions (L2L and L2P). Finally the dense $b \times b$ multiplications correspond to P2P direct interactions between nearfield particles.

The trees I and J may be assumed to have a constant (2^d) number of children per node resulting in binary trees, quadtrees, or octrees depending on the dimension d. The most natural type of tree to use depends on the provenance of the matrix. For problems involving the discretization of volume integrals, octrees are perhaps the most reasonable ones to use. Schur complement matrices arising from planar interfaces may be more naturally represented using quadtrees even in volumetric problems. Evidently, it is always possible to use binary trees regardless of the origin of the matrix, but this choice may limit attainable performance. The I and J trees may not be uniform but may have an adaptive depth as warranted by the structure of the matrix.

The structure of Q, the matrix quadtree in which the S blocks are stored as illustrated in fig. 5, does not generally have the same regularity as the corresponding set of interaction lists of FMM computations. While FMM methods use a strictly-geometric criterion to determine

Table 5. Communication breakdown of hierarchical matrix-vector multiplication compared to FMM (cf. tab. 3).

H-matrix operation	FMM operation	Processes	Blocks per level	Blocks per Process	Communication
Global $\sum S_{ij}\hat{x}_j$	Global M2L	$\sum_i^{\log P} \mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log P)$
Global $\sum F_k^t\hat{x}_k$	Global M2M	$\sum_i^{\log P} \mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log P)$
Local $\sum S_{ij}\hat{x}_j$	Local M2L	$\mathcal{O}(1)$	$\mathcal{O}(2^{(d-1)i})$	$\sum_i^{\log_{2^d}(N/P)} \mathcal{O}(2^{(d-1)i})$	$\mathcal{O}((N/P)^{\frac{d-1}{d}})$
Local $\sum A_{ij}x_j$	Local P2P	$\mathcal{O}(1)$	$\mathcal{O}(2^{(d-1)i})$	$\mathcal{O}(2^{(d-1)\log_{2^d}(N/P)})$	$\mathcal{O}((N/P)^{\frac{d-1}{d}})$

admissibility, hierarchical matrices may have a general structure for the sparsity of these block rows reflecting a non-geometric admissibility criterion. This is in fact the specific sense in which this hierarchical representation can be viewed as an algebraic variant of FMM. The leaves of the Q tree are generated algebraically so that a factorization $U_i S_{ij} V_j^t$ of rank-k results in an appropriate approximation of the block A_{ij}. FMM computations generally do not store the S_{ij} blocks but rather compute them from their analytical expressions at run-time as needed for the M2L interactions, hence saving $\mathcal{O}(Nk^2)$ storage. Such savings are only possible when there is an underlying analytical kernel. For the more general problems, storing the $k \times k$ S_{ij} matrices at the leaves of the Q quadtree is what allows hierarchical matrix representations to approximate general (dense) operators in a way that is scalable both arithmetically and in terms of communication.

In order to obtain meaningful bounds on the communication complexity of the hierarchical matrix vector multiplication, we assume that the number of blocks in a block row is bounded by a constant independent of N. In FMM computations using a regular Cartesian subdivision, this number corresponds to the number of same-size cells that are interacted with and hence is the same constant for all levels in that case. Having an $\mathcal{O}(1)$ size for the block rows is essential for maintaining linear arithmetic complexity. We also assume that the bandwidth of any level in the quadtree Q is bounded by a constant. If we denote by d the spatial dimension of the problem that generated the matrix, the amount of communication may require a large halo, due to a potentially large bandwidth in some row-block, but of $\mathcal{O}(1)$. The asymptotic complexity therefore still scales as the surface to volume ratio of d-dimensional blocks, using similar arguments to those described in Section 2. The first and third rows of tab. 5 show the asymptotic communication results for the global and local portions of the coupling phase for problems originating from a (small) spatial dimension d. The global and local portions of this computation are distributed as described in fig. 1 in section 2.

The upsweep and downsweep phases of the multiplication do not depend on the particular structure of the Q quadtree but only on the structure of the I and J trees. For practical implementations, these trees have a regular fanout of 2^d, and although they may be nonuniform in their depth they do not depend on the bandwidth of Q. Their asymptotic communication complexity is therefore essentially the same as that of the FMM for both the uniform and non-uniform cases as described in section 2. The second row of tab. 5 displays identical results to tab. 4 for the global portion of the upsweep phase. The asymptotic constants however would depend on d and on the nonuniformity of the tree depth. The rest of the upsweep and downsweep phases have analogous results. Finally, the matrix blocks requiring dense $b \times b$ direct block multiplications may induce more communication than a regular FMM P2P phase due the distribution of the dense blocks at the lowest level, but the bounded bandwidth insures a communication cost similar to that of the coupling phase. In short, this hierarchical representation under fairly reasonable assumption on its structure inherits the strong communication complexity results that have made FMM a powerful kernel for extreme computing.

4. Conclusions

Driven by seemingly inexorable trends in computer architecture at extreme scale, we have identified an algebraic form of the fast multipole method (AFM) as a candidate for migrating basic linear algebraic subroutines to hybrid hierarchical distributed-shared memory machines targeting billion-thread concurrency, where the performance portability of trusted workhorse methods of bulk synchronous structure is questionable, due to the lack of performance guarantees of individual cores. AFM is a hybrid of the fast multipole and $\mathcal{H}$-matrices, with the algebraic generality of the latter captured in the distributed tree-like data structures of the former. As with any method predicated upon compression due to low rank, the mathematical efficiency of AFM depends upon the operator. AFM is of interest for both dense and sparse operators arising from physical models through the discretization of integral and differential equations, where the underlying solution fields are fundamentally continuous and the operators smoothly varying functions of distance, with sufficiently rapid decay. AFM does not presume possession of a Green's function. With respect to massively distributed memory, it benefits from the FMM feature that no all-to-all synchronization is required. With respect to accelerators such as GPGPUs that impose in the hardware their own scales, such as the number of threads in a warp that must execute in a SIMT manner, it benefits from the tradeoff between the order of the expansion and the separation of the interacting degrees of freedom. One may choose an order that fits the architecture and enforce the admissibility condition for low-rank representation of an interaction by the size of the blocks.

The primary claims of this contribution are:

- The communication complexity of FMM for uniform distributions is $\mathcal{O}\left(\left(\frac{n}{p}\right)^{\alpha} + \log P\right)$, where $\alpha = 2/3$ for three dimensions.
- This complexity still holds for nonuniform distributions with adaptive tree structures to which FMM is applicable, and is generalizable to arbitrary dimensions.
- This complexity also holds for algebraic variants of FMM, such as H-matrices, HSS, and RS, wherever an upper bound exists on the number of blocks in a block row in the low-rank representation.

We have not yet built an implementation of the algebraic fast multipole method that illustrates of its potential performance advantages and versatility. There are components that need to be developed at the GPGPU or accelerator scale and the resulting node code needs to be merged with the FMM tree data structure and traversal mechanisms. The memory efficiency and strong absolute and scaling performance of FMM for analytically specified kernels such as the Laplacian is well documented in existing implementations, such as ExaFMM, so the primary criterion affecting the memory consumption and execution time of the AFM as a solver for linear systems or as a component, such as a matrix-vector multiply or a preconditioner in a larger context, is how efficiently general operators of interest can be compressed. We conclude with four ripe open problems:

- For operators that do not satisfy the requirement of bounded block bandwidth, independent of N, in their low-rank representation, how does the communication complexity of AFM generalize beyond the convenient bound inherited from FMM? What is the analog of dimension d in this case?
- What are the sizes of constants in the asymptotic communication complexity of the FMM and AFM method? These will generally depend upon the distributions of interacting degrees of freedom represented in the tree data structures and mathematical ranks of the operators involved. These constants will determine the natural crossover points for the applicability of FMM and AFM relative to other methods that excel in smaller problems but possess inferior asymptotic communication complexity.
- What is the achievable asynchronicity of the communications of FMM and AFM in practice and in what programming models are they best expressed?

- How should the algorithmic hierarchy be adapted to architectural hierarchy in practice? In general, the "natural" granularities (e.g., FMM expansion degree and number of threads in a GPGPU warp, size of AFM bases and capacities of various levels of cache) will not match, and the mismatch has performance implications. This highlights the importance of the tunability of the granularities of hierarchy that are at the disposal of the algorithm designer and scientific user and their exposure in the software.

Acknowledgments

Our inspiration for this paper comes from the community of researchers who joined us for two workshops bearing the title "Scalable Hierarchical Algorithms for eXtreme Computing" (SHAXC), held at the King Abdullah University of Science and Technology (KAUST) during 28-30 April 2012 and 2-4 May 2014. We are especially grateful to Lorena Barba and Matthew Knepley, who joined us in co-organizing SHAXC-1, and to Alex Litvenenko, who joined us in co-organizing SHAXC-2. Sponsorship of SHAXC-1 was from the Strategic Initiative in Extreme Computing and of SHAXC-2 from its successor, the Extreme Computing Research Center (ECRC) at KAUST. Archives of the workshops may be found online. One of ten research centers at KAUST, the ECRC operates on two fronts: (1) performing basic research and developing algorithms and software that will enable today's applications to migrate to the exascale systems that are expected by the end of the decade, and (2) enabling KAUST's scientific and engineering simulation campaigns to exploit today's state-of-the-practice petascale systems.

References

1. M. Abduljabbar, R. Yokota, and D. Keyes. Asynchronous execution of the fast multipole method using charm++. *arXiv:1405.7487*, 2014.

2. A. H. Baker, R. D. Falgout, T. Gamblin, T. V. Kolev, M. Schulz, and U. M. Yang. Scaling algebraic multigrid solvers: On the road to exascale. In C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, editors, *Competence in High Performance Computing*, pages 215–226. Springer, 2012.

3. M. Bebendorf and J. Ostrowski. Parallel hierarchical matrix preconditioners for the curl-curl operator. *Journal of Computational Mathematics*, 27(5):624–641, 2009.

4. J. Bédrof, E. Gaburov, and S. Portegies Zwart. A sparse octree gravitational N-body code that runs entirely on the GPU processor. *Journal of Computational Physics*, 231:2825–2839, 2012.

5. S. Börm. Approximation of integral operators by matrices with adaptive bases. *Computing*, 74(3):249–271, 2005.

6. M. Burtscher and K. Pingali. An efficient CUDA implementation of the tree-based Barnes Hut N-body algorithm. In *GPU Computing Gems*, chapter 6. Elsevier, 2011.

7. S. Chandrasekaran, M. Gu, and T. Pals. A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM J. Matrix Anal. Appl.*, 28(3):603–622, 2006.

8. E. Corona, P.-G. Martinsson, and D. Zorin. An $O(N)$ direct solver for integral equations on the plane. *arXiv:1303.5466*, 2013. submitted to SIAM Journal of Scientific Computing.

9. J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J. Andre, D. Barkai, J. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, Xuebin C., A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, W. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, J. Zhong, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, W. Kramer, J. Labarta, Al. Lichnewsky, T. Lippert, R. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, R. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

10. E. Gaburov, J. Bédrof, and S. Portegies Zwart. Gravitational tree-code on graphics processing units: Implementation in CUDA. *arXiv:1005.5384v1*, 2010.

11. A. Gillman, P. M. Young, and P.-G. Martinsson. A direct solver with $O(N)$ complexity for integral equations on one-dimensional domains. *Frontiers of Mathematics in China*, 7(2):217–247, 2012.

12. L. Grasedyck and W. Hackbusch. Construction and arithmetics of $\mathcal{H}$-matrices. *Computing*, 70(4):295–334, 2003.

13. L. Grasedyck, R. Kriemann, and S. Le Borne. Parallel black box HLU preconditioning for elliptic boundary value problems. *Computing and Visualization in Science*, 11(4-6):273–291, 2008.

14. L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987.

15. N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *Journal of Computational Physics*, 227:8290–8313, 2008.

16. W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. part I: Introduction to $\mathcal{H}$-matrices. *Computing*, 62(2):89–108, 1999.

17. W. Hackbusch and S. Börm. Data-sparse approximation by adaptive H2-matrices. *Computing*, 69(1):1–35, 2002.

18. W. Hackbusch, B. Khoromskij, and S.A. Sauter. On $\mathcal{H}^2$-matrices. In H. Bungartz, R. Hoppe, and Zenger C., editors, *Lectures on Applied Mathematics*, pages 9–29. Springer, 2000.

19. T. Hamada and K. Nitadori. 190 Tflops astrophysical N-body simulation on cluster of GPUs. In *SC '10 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

20. T. Hamada, R. Yokota, K. Nitadori, T. Narumi, K. Yasuoka, M. Taiji, and K. Oguri. 42 Tflops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

21. H. Ibeid, R. Yokota, and D. E. Keyes. A performance model for the communication in fast multipole methods on hpc platforms. *arXiv:1405.6362v1*, 2014.

22. T. Ishiyama, K. Nitadori, and J. Makino. 4.45 Pflops astrophysical N-body simulation on K computer – The gravitational trillion-body problem. In *Proceedings of the 2012 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

23. P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn. Scaling hierarchical N-body simulations on GPU clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

24. A. Kawai, T. Fukushige, and J. Makino. $ 7.0/Mflops astrophysical N-body simulation with treecode on GRAPE-5. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, pages 1–6, 1999.

25. P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, Al. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, DARPA, 2008.

26. H. Langston, M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. Re-introduction of communication-avoiding FMM-accelerated FFTs with GPU acceleration. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2013.

27. I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

28. H. C. Plummer. On the problem of distribution in globular star clusters. *Monthly Notices of the Royal Astronomical Society*, 71:460–470, 1911.

29. A. Rahimian, I. Lashuk, K. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, 2010.

30. M. J. Stock and A. Gharakhani. Toward efficient GPU-accelerated N-body simulations. *AIAA Paper*, 2008-608, 46th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, Jan. 7 - 10:1–13, 2008.

31. H. Sundar, R. S. Sampath, and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.

32. S.-H. Teng. Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation. *SIAM Journal on Scientific Computing*, 19(2):635–656, 1998.

33. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

34. R. Vandebril, M.Van Barel, G. Golub, and N. Mastronardi. A bibliography on semiseparable matrices*. *CALCOLO*, 42(3-4):249–270, 2005.

35. M. S. Warren, T. C. Germann, P. S. Lomdahl, D. M. Beazley, and J. K. Salmon. Avalon: An Alpha/Linux cluster achieves 10 Gflops for $150k. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–10, 1998.

36. M. S. Warren and J. K. Salmon. Astrophysical N-body simulation using hierarchical tree data structures. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 570–576, 1992.

37. M. S. Warren, J. K. Salmon, D. J. Becker, M. P. Goda, and T. Sterling. Pentium pro inside: I. a treecode at 430 Gigaflops on ASCI red, II. price/performance of $ 50/Mflop on Loki and Hyglac. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–16, 1997.

38. S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications in Applied Mathematics and Computational Science*, 52(4):65–76, 2009.

39. J. Xia, S. Chandrasekaran, M. Gu, and X.-S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010.

40. R. Yokota and L. A. Barba. Treecode and fast multipole method for N-body simulation with CUDA. In *GPU Computing Gems*, chapter 9. Morgan Kaufmann, Emerald edition, 2011.

41. R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, and K. Yasuoka. Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence. *Computer Physics Communications*, 180:2066–2078, 2009.

42. R. Yokota, J. Pestana, H. Ibeid, and D. E. Keyes. Fast multipole preconditioners for sparse matrices arising from elliptic equations. *arXiv:1308.3339v2*, 2014.

DOI: 10.14529/jsfi140105

Model-Driven One-Sided Factorizations on Multicore Accelerated Systems

Jack Dongarra[1 2 3]***, Azzam Haidar***[1]***, Jakub Kurzak***[1]***, Piotr Luszczek***[1]***, Stanimire Tomov***[1]***, Asim YarKhan***[1]

Hardware heterogeneity of the HPC platforms is no longer considered unusual but instead have become the most viable way forward towards Exascale. In fact, the multitude of the heterogeneous resources available to modern computers are designed for different workloads and their efficient use is closely aligned with the specialized role envisaged by their design. Commonly in order to efficiently use such GPU resources, the workload in question must have a much greater degree of parallelism than workloads often associated with multicore processors (CPUs). Available GPU variants differ in their internal architecture and, as a result, are capable of handling workloads of varying degrees of complexity and a range of computational patterns. This vast array of applicable workloads will likely lead to an ever accelerated mixing of multicore-CPUs and GPUs in multi-user environments with the ultimate goal of offering adequate computing facilities for a wide range of scientific and technical workloads. In the following paper, we present a research prototype that uses a lightweight runtime environment to manage the resource-specific workloads, and to control the dataflow and parallel execution in hybrid systems. Our lightweight runtime environment uses task superscalar concepts to enable the developer to write serial code while providing parallel execution. This concept is reminiscent of dataflow and systolic architectures in its conceptualization of a workload as a set of side-effect-free tasks that pass data items whenever the associated work assignment have been completed. Additionally, our task abstractions and their parametrization enable uniformity in the algorithmic development across all the heterogeneous resources without sacrificing precious compute cycles. We include performance results for dense linear algebra functions which demonstrate the practicality and effectiveness of our approach that is aptly capable of full utilization of a wide range of accelerator hardware.

Keywords: hardware accelerators, dense linear algebra, task superscalar scheduling.

1. Introduction

With the release of CUDA in 2007, the communities of scientific, HPC, and technical computing started to enjoy the performance benefits of GPUs. The ever expanding capabilities of the hardware accelerators allowed GPUs to deal with more demanding kinds of workloads and there was very little need to mix different GPUs in the same machine. Intel offering in the realm of hardware acceleration came in the form of a coprocessor called MIC (Many Integrated Cores) now known as Xeon Phi and available under Knights Corner moniker. In terms of capabilities, on the one hand, they are similar to those of GPUs but, on the other hand, there are some slight differences in workloads that could be handled by Phi. We do not intend in this paper to compare between GPUs and the recently introduced MIC coprocessor. Instead, we take a different stand because we believe that the users will combine CPUs, GPUs, and coprocessors to leverage strengths of each of them depending on the workload. In a similar fashion, we are showing here how to combine their strengths to arrive at another level of heterogeneity by simultaneously utilizing all three: CPUs, GPUs, and coprocessors. We call this a multi-way heterogeneity for which we present a unified programming model that alleviates the complexity of dealing with multiple software stacks for computing, communication, and software libraries.

[1]University of Tennessee, Knoxville, USA
[2]Oak Ridge National Laboratory, Oak Ridge, USA
[3]University of Manchester, Manchester M13 9PL, UK

2. Background and Related Work

This paper presents research in algorithms and a programing model for high-performance dense linear algebra (DLA) factorizations for heterogeneous multiple multicore CPUs, GPUs, and Intel Xeon Phi coprocessors (MICs). The mix can contain resources with varying capabilities, e.g., CUDA GPUs same architecture but from different device generations. While the main goal is to obtain as high fraction of the peak performance as possible for an entire heterogeneous system, an often competing secondary goal is to propose a programming model that would simplify the development. To this end, we propose and develop a new lightweight runtime environment, and a number of dense linear algebra routines based on it. We demonstrate the new algorithms, their performance, and the programming model design using the Cholesky and QR factorizations.

2.1. High Performance on Heterogeneous Systems

Efficient use of current computer architectures, namely, running close to their peak performance, can only be achieved for algorithms of high computational intensity. In the specific context of DLA, $O(n^2)$ data requires $O(n^3)$ floating point operations (flop's). In contrast, less compute intensive algorithms that use sparse linear algebra can reach only a fraction of the peak performance, e.g., a few Gflop/s for highly optimized sparse matrix-vector multiply (SpMV) [6] versus over 1000 Gflop/s for double-precision matrix-matrix multiply (DGEMM) on a Kepler K20c GPU [11]. This highlights the interest in DLA and the importance of designing computational applications that can make efficient use of the available hardware resources.

Early results for DLA were tied to development of high performance Basic Linear Algebra Subroutines (BLAS). Volkov el al. [30] developed fast single-precision matrix-matrix multiply (SGEMM) for the NVIDIA Tesla GPUs and highly efficient LU, QR, and Cholesky factorizations based on that. The fastest DGEMM and factorizations based on it for the NVIDIA Fermi GPUs were developed by Nath et al. [20]. These early BLAS developments were eventually incorporated into the CUBLAS library [11]. The main DLA algorithms from LAPACK were developed for a single GPU and released through the MAGMA library [19].

More recent work has concentrated on porting additional algorithms to GPUs, and on various optimizations and algorithmic improvements. The central challenge has been to split the computation among the hardware components to efficiently use them, to maintain balanced load, and to reduce idle times. Although there have been developments for a particular accelerator and its multicore host [3, 5, 9, 12, 25, 26], to the best of our knowledge there has been no efforts to create a DLA library on top of a unified framework for multi-way heterogeneous systems.

2.2. Heterogeneous parallel programming models

Both NVIDIA and Intel provide various programming models for their GPUs and coprocessors. In particular, to program and assign work to NVIDIA GPUs and Intel Xeon Phi coprocessors, we use CUDA APIs and heterogeneous offload pragmas/directives, respectively. To unify the parallel scheduling of work between these different architectures we designed a new API, based on a lightweight task superscalar runtime environment, that automates the task scheduling across the devices. In this way, the API can be easily extended to handle other devices that use their own programming models with OpenCL [9] being a natural target for supporting AMD Radeon compute cards.

Task superscalar runtime environments have become a common approach for effective and efficient execution in multicore environments. This task scheduling mechanism has its roots in dataflow execution, which has a long history dating to the nineteen sixties [7, 16, 24]. It has seen a reemergence in popularity with the advent of multicore processors in order to use the available resources dynamically and efficiently. Many other programming methodologies need to be carefully managed in order to avoid fork-join style parallelism, also called Bulk Synchronous Processing [29], which is increasingly wasteful in face of ever larger numbers of cores in a single CPU socket.

In our current research, we build on the QUARK [31, 32] runtime environment to demonstrate the effectiveness of dynamic task superscalar execution in the presence of heterogeneous architectures. QUARK is a superscalar execution environment that has been used with great success for linear algebra software on multicore platforms. The PLASMA linear algebra library has been implemented using QUARK and has demonstrated excellent scalability and high performance [2, 17].

There is a rich area of work on execution environments that begin with serial code and result in parallel execution, often using task superscalar techniques, for example Jade [23], Cilk [8], Sequoia [13], SuperMatrix [10], OmpSS [22], Habanero [5], StarPU [4], or the DepSpawn [14] project.

We chose QUARK as our lightweight runtime environment for this research because it provides flexibility in low level control of task location and binding that would be harder to obtain using other superscalar runtime environments. But, the conceptual work done in this research could be replicated within other projects, so we view QUARK simply as a convenient representative of a lightweight, task-superscalar runtime environment.

Both GPUs and Intel Xeon Phi coprocessors provide high degree of parallelism that can deliver excellent application performance for DLA. However, it is important to understand the distinctions in programming models between GPUs and Intel Xeon Phi coprocessors.

2.2.1. CUDA

CUDA utilizes a large number of concurrent threads of execution to exploit hardware parallelism and achieve high performance. CUDA threads are grouped together into blocks that execute concurrently on the GPU Streaming Multiprocessors (SMs) following the SIMD execution model. NVIDIA Kepler K20 GPU has 2496 CUDA cores and delivers 1.17 Tflop/s in peak double precision floating point performance.

2.2.2. MIC Architecture and Programming Models

The Intel Xeon Phi coprocessor comprises of 61 Intel Architecture (IA) cores. Additionally, there are 8 memory controllers supporting up to 16 GDDR5 channels delivering a theoretical bandwidth of 352 GB/s [1]. Its peak double precision floating point performance is 1200 Gflop/s. Unlike a GPU, the Xeon Phi coprocessor runs a Linux operating system and provides x86 compatibility. It supports popular programming models used on multi-core architectures including MPI, OpenMP and Threading Building Blocks. Xeon Phi also offers the following flexible programming models depending on applications:

- **Native model**: It is used to run applications entirely on a Xeon Phi coprocessor like any other multi-core system.

- **Offload model**: It is used to utilize a Xeon Phi coprocessor as an accelerator to execute computation intensive regions of an application offloaded from host.

Offloading incurs overhead costs for initialization, marshaling and transferring data, and code invocation. Applications must have a high ratio of computation to communication for the offloaded portion. Offload communication overheads can be hidden if code is structured to overlap them with computation, or if data is reused across offload regions [21]. To effectively make use of this model, asynchronous offload features must be used, which include: asynchronous data transfer, asynchronous compute and memory management without data transfer. Signal and wait clauses need to be used in offload pragmas and directives to enable asynchronous data transfer and computation. For example:

offload_transfer pragma/directive called with a signal clause begins an asynchronous data transfer.

offload pragma/directive called with a signal clause begins an asynchronous computation.

offload_wait pragma/directive called with a wait clause blocks execution until an asynchronous data transfer or computation is complete.

At this point, this programming model has been codified by the OpenMP 4.0 standard and may be considered analogous to the OpenACC standard initially implemented on GPUs only.

Beside the above programming models, Intel provides two software libraris – SCIF (Symmetric Communication Interface) and COI (Coprocessor Offload Infrastructure) – to ease the development of software tools and applications that run on the Xeon Phi Coprocessor.

- SCIF (Symmetric Communication Interface) is an API for communication between processes on MIC and the host CPU within the same system. It resembles POSIX socket interface whereby connections are established using *scif_listen*, *scif_connect*, and *scif_accept* routines. For efficient communication between the processes SCIF API provides send-receive semantics where one serves as the source and the other as the destination of the communication channel. Efficient data transfer is ensured by Remote Memory Access (RMA) semantics where a region of memory needs to be registered (using *scif_register* first and *scif_unregister* afterwards) for remote access. Upon registration, further data transfers (using *scif_readfrom* and *scif_writeto* can occur in a one-sided manner with just one process either reading from or writing to a memory window. These reads and writes can make both synchronous and asynchronous progress and the completion of asynchronous RMA operations is checked using *scif_fence_mark* and *scif_fence_wait* routines.
- COI (Coprocessor Offload Infrastructure) model exposes a pipelined programming model to the user. This model allows workloads to run and data to be moved asynchronously, allowing the host processor, device processor, and DMA engines to remain busy simultaneously and independently. This library provides services to create coprocessor-side processes, create FIFO pipelines between the host and coprocessor, move code and data, invoke code (functions) on the coprocessor, manage memory buffers that span the host and coprocessor, enumerate available resources, and so on [21].

3. BLAS

The *Basic Linear Algebra Subroutines* (BLAS) are the main building blocks for dense matrix software packages. Furthermore, the matrix-matrix multiplication routine is the most common

```
for (m = 0; m < M; m++)
  for (n = 0; n < N; n++)
    for (k = 0; k < K; k++)
      C[n][m] += A[k][m]*B[n][k];
```

Figure 1. Canonical form of matrix-matrix multiplication.

```
for (m_ = 0; m_ < M; m_+=tileM)
 for (n_ = 0; n_ < N; n_+=tileN)
  for (k_ = 0; k_ < K; k_+=tileK)
   for (m = 0; m < tileM; m++)
    for (n = 0; n < tileN; n++)
     for (k = 0; k < tileK; k++)
      C[n_+n][m_+n] +=
       A[k_+k][m_+m]*
       B[n_+n][k_+k];
```

Figure 2. Matrix-matrix multiplication with loop tiling.

```
for (m_ = 0; m_ < M; m_+=tileM)
 for (n_ = 0; n_ < N; n_+=tileN)
  for (k_ = 0; k_ < K; k_+=tileK)
  {
   instruction
   instruction
   instruction
     ...
  }
```

Figure 3. Matrix-matrix multiplication with complete unrolling of tile operations.

and most performance-critical BLAS routine, more so in the context of the factorizations that we consider in this paper. This section presents the process of building a fast matrix-matrix multiplication kernel for the GPU in double precision and in real arithmetic (DGEMM) by using the process of autotuning. The target is the NVIDA K40c card.

In its canonical form, matrix-matrix multiplication is represented by three nested loops as is shown in Figure 1. The primary tool in optimizing matrix-matrix multiplication is the technique of *loop tiling*. Tiling replaces a single loop with two loops: the inner loop that increments the iteration counter by one, and the outer loop that increments the iteration counter by the tiling factor. In the case of matrix-matrix multiplication, tiling replaces the three loops of Figure 1 with the six loops of Figure 2. Tiling of the matrix-matrix multiplication exploits the *surface to volume effect* mentioned before: execution of $O(n^3)$ floating-point operations over $O(n^2)$ data.

Next, the technique of loop unrolling is applied, which replaces the three innermost loops with a single block of straight-line of code (a single *basic block*), as shown schematically in Figure 3. The purpose of unrolling is twofold: to reduce the penalty of looping (the overhead of incrementing loop counters, advancing data pointers and conditional branching), and to increase instruction-level parallelism by creating long sequences of independent instructions, which can fill out the processor's pipeline and thus increase the instruction issue rate.

This optimization sequence is universal for almost any computer architecture, including "standard" superscalar processors with cache memories, as well as GPU accelerators and other less conventional architectures. Tiling, also referred to as blocking, is often applied multiple times to accommodate successive levels of cache, the registers file, TLB, etc.

In the case of a GPU, the C matrix (in $C \leftarrow C - A \times B$) is overlaid with a 2D grid of thread blocks, each one responsible for computing a single tile of C. Since the code of a GPU kernel spells out the operation of a single thread block, the two outer loops disappear, and only one loop remains – the loop advancing along the k dimension tile by tile. Unrolling the outer loop, and then fusing together copies of the inner loop, sometimes called unroll and jam, is usually

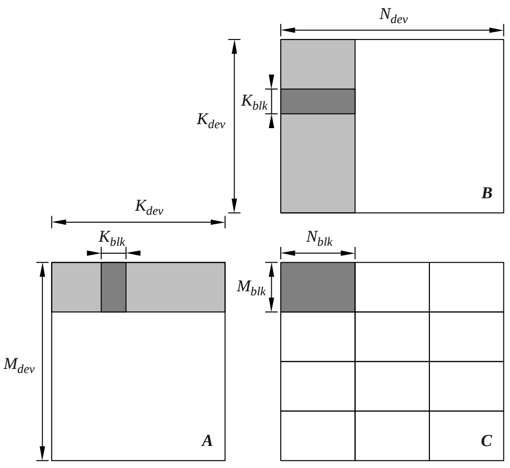


Figure 4. GEMM at the device level.

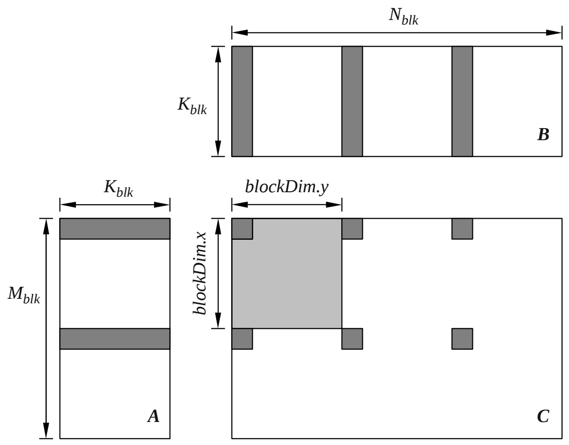


Figure 5. GEMM at the block level.

applied until the point where the register file is exhausted. Unrolling the inner loop beyond that point, can be used to further reduce the overhead of looping.

Figure 4 shows the GPU implementation of matrix-matrix multiplication at the device level. Each thread block computes a tile of C (dark gray) by passing through a stripe of A and a stripe of B (light gray). The code iterates over A and B in chunks of K_{blk} (dark gray). The thread block follows the cycle of:

- making texture reads of the small, dark gray, stripes of A and B and storing them in shared memory,
- synchronizing threads with the `__syncthreads()` call,
- loading A and B from shared memory to registers and computing the product,
- synchronizing threads with the `__syncthreads()` call.

After the complete sweep of the light gray stripes of A and B completes, the tile of C is read, updated and stored back to the device memory. Figure 5 shows closer what happens in the inner loop. The light gray area shows the shape of the thread block. The dark gray regions show how a single thread iterates over the tile.

Figure 6 shows the complete kernel implementation in CUDA. Tiling is defined by `BLK_M`, `BLK_N`, and `BLK_K`. `DIM_X` and `DIM_Y` specify how the thread block covers the tile of C, `DIM_XA` and

`DIM_YA` define how the thread block covers a stripe of A, and finally `DIM_XB` and `DIM_YB` define how the thread block covers a stripe of B.

In lines 24–28 the values of C are set to zero. In lines 32–38 a stripe of A is read (texture reads) and stored in shared memory. In lines 40–46 a stripe of B is read (texture reads) and stored in shared memory. The `__syncthreads()` call in line 48 ensures that the process of reading of A and B, and storing in shared memory, is finished before operation continues. In lines 50–56 the product is computed, using the values from shared memory. The `__syncthreads()` call in line 58 ensures that computing the product is finished and the shared memory can be overwritten with new stripes of A and B. In lines 60 and 61 the pointers are advanced to the location of new stripes. When the main loop completes, C is read from device memory, modified with the accumulated product, and written back, in lines 64–77. The use of texture reads with clamping eliminates the need for *cleanup code* to handle matrix sizes not exactly divisible by the tiling factors.

With the parametrized code in place, what remains is the actual autotuning process, i.e., finding good values for the nine tuning parameters. Here the process used in the *Bench-testing Environment for Automated Software Tuning* (BEAST) project is described. It relies on three components:

1. defining the search space,
2. pruning the search space by applying filtering constraints, and
3. benchmarking the remaining configurations and selecting the best performer.

The important point in the BEAST project is to not introduce artificial and/or arbitrary limitations on the search process.

The loops in Figure 7 define the search space for the autotuning of the matrix-matrix multiplication of Figure 6. The two outer loops sweep through all possible 2D shapes of the thread block – up to the device limit in each dimension. The three inner loops sweep through all possible tiling sizes – up to arbitrarily high values, represented by the INF symbol. In practice, the actual values to substitute for the INF symbols can be found by choosing a small starting point, e.g., (64, 64, 8), and going up until further increase has no effect on the number of kernels that pass the selection process based on pruning constraints.

The list of pruning constraints consists of nine simple checks that eliminate kernels deemed inadequate for one of several reasons:

- The kernel would not compile because it exceeded a hardware limit.
- The kernel would compile but failed to launch because it exceeded a hardware limit.
- The kernel would compile and launch, but produced invalid results due to the limitations of the implementation, e.g., unimplemented corner case.
- The kernel would compile, launch, and produce correct results, but have no chance of running fast, due to an obvious performance shortcoming, such as very low occupancy.

The nine checks rely on basic hardware parameters, which can be obtained by querying the card with the CUDA API. The parameters include:

1. The number of threads in the block is not divisible by the warp size.
2. The number of threads in the block exceeds the hardware maximum.
3. The number of registers per thread, to store C, exceeds the hardware maximum.
4. The number of registers per block, to store C, exceeds the hardware maximum.
5. The shared memory per block, to store A and B, exceeds the hardware maximum.
6. The thread block cannot be shaped to read A and B without cleanup code.

```
extern "C" __global__
void beast_gemm_kernel(
    int M, int N, int K,
    double alpha, double *A, int lda,
                  double *B, int ldb,
    double  beta, double *C, int ldc )
{
    int blx = blockIdx.x;        // block's m position
    int bly = blockIdx.y;        // block's n position
    int idx = threadIdx.x;       // thread's m position in C
    int idy = threadIdx.y;       // thread's n position in C
    int idt = DIM_X*idy+idx;     // thread's number

    int idxA = idt % DIM_XA;     // thread's m position for loading A
    int idyA = idt / DIM_XA;     // thread's n position for loading A
    int idxB = idt % DIM_XB;     // thread's m position for loding B
    int idyB = idt / DIM_XB;     // thread's n position for loading B

    __shared__ double sA[BLK_K][BLK_M+1];    // shared memory buffer for A
    __shared__ double sB[BLK_N][BLK_K+1];    // shared memory buffer for B
    double rC[BLK_N/DIM_Y][BLK_M/DIM_X];     // registers for C

    int coord_A = blx*BLK_M     + idyA*lda+idxA;     // A stripe's initial location
    int coord_B = bly*BLK_N*ldb + idyB*ldb+idxB;     // B stripe's initial location
    int m, n, k, kk;                                 // loop counters

    #pragma unroll
    for (n = 0; n < BLK_N/DIM_Y; n++)
      #pragma unroll
      for (m = 0; m < BLK_M/DIM_X; m++)
        rC[n][m] = 0.0;

    for (kk = 0; kk < K; kk += BLK_K)
    {
        #pragma unroll
        for (n = 0; n < BLK_K; n += DIM_YA)
            #pragma unroll
            for (m = 0; m < BLK_M; m += DIM_XA) {
                int2 v = tex1Dfetch(tex_ref_A, coord_A + n*lda+m);
                sA[n+idyA][m+idxA] = __hiloint2double(v.y, v.x);
            }

        #pragma unroll
        for (n = 0; n < BLK_N; n += DIM_YB)
            #pragma unroll
            for (m = 0; m < BLK_K; m += DIM_XB) {
                int2 v = tex1Dfetch(tex_ref_B, coord_B + n*ldb+m);
                sB[n+idyB][m+idxB] = __hiloint2double(v.y, v.x);
            }

        __syncthreads();

        #pragma unroll
        for (k = 0; k < BLK_K; k++)
            #pragma unroll
            for (n = 0; n < BLK_N/DIM_Y; n++)
                #pragma unroll
                for (m = 0; m < BLK_M/DIM_X; m++)
                    rC[n][m] += sA[k][m*DIM_X+idx] * sB[n*DIM_Y+idy][k];

        __syncthreads();

        coord_A += BLK_K*lda;
        coord_B += BLK_K;
    }

    #pragma unroll
    for (n = 0; n < BLK_N/DIM_Y; n++) {
        int coord_dCn = bly*BLK_N + n*DIM_Y+idy;
        #pragma unroll
        for (m = 0; m < BLK_M/DIM_X; m++) {
            int coord_dCm = blx*BLK_M + m*DIM_X+idx;
            if (coord_dCm < M && coord_dCn < N) {
                int offsC = coord_dCn*ldc + coord_dCm;
                double &regC = rC[n][m];
                double &memC = C[offsC];
                memC = alpha*regC + beta*memC;
            }
        }
    }
}
```

Figure 6. Complete dgemm (C = alpha A B + beta C) implementation in CUDA.

```
// Sweep thread block dimensions.
for (dim_m = 1; dim_m <= MAX_THREADS_DIM_X; dim_m++)
 for (dim_n = 1; dim_n <= MAX_THREADS_DIM_Y; dim_n++)
  // Sweep tiling sizes.
  for (blk_m = dim_m; blk_m < INF; blk_m += dim_m)
   for (blk_n = dim_n; blk_n < INF; blk_n += dim_n)
    for (blk_k = 1; blk_k < INF; blk_k++)
    {
     // Apply pruning constraints.
    }
```

Figure 7. The parameter search space for the autotuning of matrix multiplication.

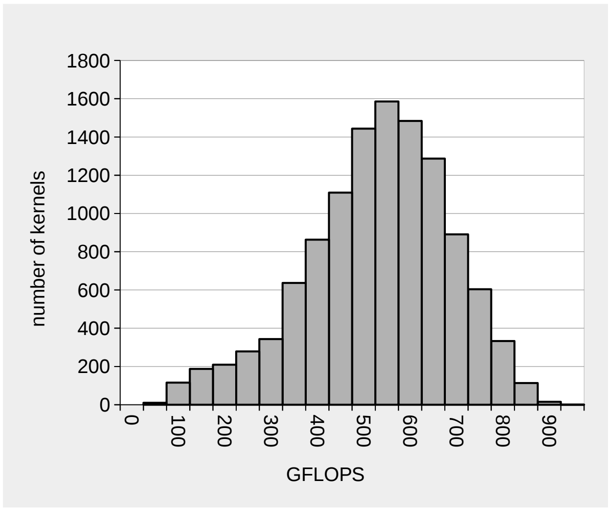


Figure 8. Distribution of the dgemm kernels.

7. The number of load instructions, from shared memory to registers, in the innermost loop, in the PTX code, exceeds the number of *Fused Multiply-Adds* (FMAs).
8. Low occupancy due to high number of registers per block to store C.
9. Low occupancy due to the amount of shared memory per block to read A and B.

In order to check the last two conditions, the number of registers per block, and the amount of shared memory per block are computed. Then the maximum number of possible blocks per multiprocessor is found, which gives the maximum possible number of threads per multiprocessor. If that number is lower than the minimum occupancy requirement, the kernel is discarded. Here the threshold is set to a fairly low number of 256 threads, which translates to minimum occupancy of 0.125 on the Nvidia K40 card, with the maximum number of 2,048 threads per multiprocessor.

This process produces 14,767 kernels, which can be benchmarked in roughly one day. 3,256 kernels fail to launch due to excessive number of registers per block. The reason is that the pruning process uses a lower estimate on the number of registers, and the compiler actually produces code requiring more registers. We could detect it in compilation and skip benchmarking of such kernels or we can run them and let then fail. For simplicity we chose the latter. We could also cap the register usage to prevent the failure to launch. However, capping register usage usually produces code of inferior performance.

Eventually, 11,511 kernels ran successfully and pass correctness checks. Figure 8 shows the performance distribution of these kernels. The fastest kernel achieves 900 Gflop/s with tiling of $96 \times 64 \times 12$, with 128 threads (16×8 to compute C, 32×4 to read A, and 4×32 to read B). The achieved occupancy number of 0.1875 indicates that, most of the time, each multiprocessor executes 384 threads (three blocks).

In comparison, CUBLAS achieves the performance of 1,225 Gflop/s using 256 threads per multiprocessor. Although CUBLAS achieves a higher number, this example shows the effectiveness of the autotuning process in quickly creating well performing kernels from high level language source codes. This technique can be used to build kernels for routines not provided in vendor libraries, such as extended precision BLAS (double-double and triple-float), BLAS for oddly shaped matrices (tall and skinny), etc. Even more importantly, this technique can be used to build domain specific kernels for much broader spectrum of application areas.

As the last interesting observation, we offer a look at the PTX code produced by NVIDIA's `nvcc` compiler (Figure 9). We can see that the compiler does exactly what is expected: it completely unrolls the loops in lines 50–56 of the C code in Figure 6 into a stream of loads from shared memory to registers followed by FMA instructions, with substantially larger number of FMAs than loads.

4. Algorithmic Advancements

In this section, we present the linear algebra aspects of our generic solution for development of either Cholesky, LU (based on Gaussian elimination), or QR (based on applying Householder reflectors) factorizations which use block outer-product updates of the trailing matrix. These three factorizations are commonly known as the *one-sided* factorizations because they apply a sequence of transformations on only one side of the original matrix and thus they do not preserve the matrix spectrum as the *two-sided* transformations do.

Conceptually, one-sided factorization maps a matrix A into a product of matrices X and Y:

$$\mathcal{F}: \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \mapsto \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \times \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}$$

Algorithmically, this corresponds to a sequence of in-place transformations of A, whose storage is overwritten with the entries of matrices X and Y (P_{ij} indicates the currently factorized panels):

$$\begin{bmatrix} A_{11}^{(0)} & A_{12}^{(0)} & A_{13}^{(0)} \\ A_{21}^{(0)} & A_{22}^{(0)} & A_{23}^{(0)} \\ A_{31}^{(0)} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \to \begin{bmatrix} P_{11} & A_{12}^{(0)} & A_{13}^{(0)} \\ P_{21} & A_{22}^{(0)} & A_{23}^{(0)} \\ P_{31} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \to$$

$$\to \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & A_{22}^{(1)} & A_{23}^{(1)} \\ X_{31} & A_{32}^{(1)} & A_{33}^{(1)} \end{bmatrix} \to \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & P_{22} & A_{23}^{(1)} \\ X_{31} & P_{32} & A_{33}^{(1)} \end{bmatrix} \to$$

$$\to \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & A_{33}^{(2)} \end{bmatrix} \to \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & X_{22} & Y_{23} \\ X_{31} & X_{32} & P_{33} \end{bmatrix} \to$$

$$\to \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & XY_{33} \end{bmatrix} \to \begin{bmatrix} XY \end{bmatrix},$$

where XY_{ij} is a compact representation of both X_{ij} and Y_{ij} in the space originally occupied by A_{ij}.

Observe two distinct phases in each step of the transformation from $[A]$ to $[XY]$: *panel factorization* (P) and trailing matrix update: $A^{(i)} \to A^{(i+1)}$. The implementation of these two

```
    ld.shared.f64   %fd258, [%rd3];
    ld.shared.f64   %fd259, [%rd4];
    fma.rn.f64  %fd260, %fd258, %fd259, %fd1145;
    ld.shared.f64   %fd261, [%rd3+128];
    fma.rn.f64  %fd262, %fd261, %fd259, %fd1144;
    ld.shared.f64   %fd263, [%rd3+256];
    fma.rn.f64  %fd264, %fd263, %fd259, %fd1143;
    ld.shared.f64   %fd265, [%rd3+384];
    fma.rn.f64  %fd266, %fd265, %fd259, %fd1142;
    ld.shared.f64   %fd267, [%rd3+512];
    fma.rn.f64  %fd268, %fd267, %fd259, %fd1141;
    ld.shared.f64   %fd269, [%rd3+640];
    fma.rn.f64  %fd270, %fd269, %fd259, %fd1140;
    ld.shared.f64   %fd271, [%rd4+832];
    fma.rn.f64  %fd272, %fd258, %fd271, %fd1139;
    fma.rn.f64  %fd273, %fd261, %fd271, %fd1138;
    fma.rn.f64  %fd274, %fd263, %fd271, %fd1137;
    fma.rn.f64  %fd275, %fd265, %fd271, %fd1136;
    fma.rn.f64  %fd276, %fd267, %fd271, %fd1135;
    fma.rn.f64  %fd277, %fd269, %fd271, %fd1134;
    ld.shared.f64   %fd278, [%rd4+1664];
    fma.rn.f64  %fd279, %fd258, %fd278, %fd1133;
    fma.rn.f64  %fd280, %fd261, %fd278, %fd1132;
    fma.rn.f64  %fd281, %fd263, %fd278, %fd1131;
    fma.rn.f64  %fd282, %fd265, %fd278, %fd1130;
    fma.rn.f64  %fd283, %fd267, %fd278, %fd1129;
    fma.rn.f64  %fd284, %fd269, %fd278, %fd1128;
    ld.shared.f64   %fd285, [%rd4+2496];
    fma.rn.f64  %fd286, %fd258, %fd285, %fd1127;
    fma.rn.f64  %fd287, %fd261, %fd285, %fd1126;
    fma.rn.f64  %fd288, %fd263, %fd285, %fd1125;
    fma.rn.f64  %fd289, %fd265, %fd285, %fd1124;
    fma.rn.f64  %fd290, %fd267, %fd285, %fd1123;
    fma.rn.f64  %fd291, %fd269, %fd285, %fd1122;
    ld.shared.f64   %fd292, [%rd4+3328];
    fma.rn.f64  %fd293, %fd258, %fd292, %fd1121;
    fma.rn.f64  %fd294, %fd261, %fd292, %fd1120;
    fma.rn.f64  %fd295, %fd263, %fd292, %fd1119;
    fma.rn.f64  %fd296, %fd265, %fd292, %fd1118;
    fma.rn.f64  %fd297, %fd267, %fd292, %fd1117;
    fma.rn.f64  %fd298, %fd269, %fd292, %fd1116;
    ld.shared.f64   %fd299, [%rd4+4160];
    fma.rn.f64  %fd300, %fd258, %fd299, %fd1115;
    fma.rn.f64  %fd301, %fd261, %fd299, %fd1114;
    fma.rn.f64  %fd302, %fd263, %fd299, %fd1113;
    fma.rn.f64  %fd303, %fd265, %fd299, %fd1112;
    fma.rn.f64  %fd304, %fd267, %fd299, %fd1111;
    fma.rn.f64  %fd305, %fd269, %fd299, %fd1110;
    ld.shared.f64   %fd306, [%rd4+4992];
    fma.rn.f64  %fd307, %fd258, %fd306, %fd1109;
    fma.rn.f64  %fd308, %fd261, %fd306, %fd1108;
    fma.rn.f64  %fd309, %fd263, %fd306, %fd1107;
    fma.rn.f64  %fd310, %fd265, %fd306, %fd1106;
    fma.rn.f64  %fd311, %fd267, %fd306, %fd1105;
    fma.rn.f64  %fd312, %fd269, %fd306, %fd1104;
    ld.shared.f64   %fd313, [%rd4+5824];
    fma.rn.f64  %fd314, %fd258, %fd313, %fd1103;
    fma.rn.f64  %fd315, %fd261, %fd313, %fd1102;
    fma.rn.f64  %fd316, %fd263, %fd313, %fd1101;
    fma.rn.f64  %fd317, %fd265, %fd313, %fd1100;
    fma.rn.f64  %fd318, %fd267, %fd313, %fd1099;
    fma.rn.f64  %fd319, %fd269, %fd313, %fd1098;
    ld.shared.f64   %fd320, [%rd3+776];
    ld.shared.f64   %fd321, [%rd4+8];
    fma.rn.f64  %fd322, %fd320, %fd321, %fd260;
    ld.shared.f64   %fd323, [%rd3+904];
    fma.rn.f64  %fd324, %fd323, %fd321, %fd262;
    ld.shared.f64   %fd325, [%rd3+1032];
    fma.rn.f64  %fd326, %fd325, %fd321, %fd264;
    ld.shared.f64   %fd327, [%rd3+1160];
    fma.rn.f64  %fd328, %fd327, %fd321, %fd266;
    ld.shared.f64   %fd329, [%rd3+1288];
    fma.rn.f64  %fd330, %fd329, %fd321, %fd268;
    ld.shared.f64   %fd331, [%rd3+1416];
    fma.rn.f64  %fd332, %fd331, %fd321, %fd270;
    ld.shared.f64   %fd333, [%rd4+840];
    fma.rn.f64  %fd334, %fd320, %fd333, %fd272;
    fma.rn.f64  %fd335, %fd323, %fd333, %fd273;
    fma.rn.f64  %fd336, %fd325, %fd333, %fd274;
    fma.rn.f64  %fd337, %fd327, %fd333, %fd275;
    fma.rn.f64  %fd338, %fd329, %fd333, %fd276;
    fma.rn.f64  %fd339, %fd331, %fd333, %fd277;
    ld.shared.f64   %fd340, [%rd4+1672];
    fma.rn.f64  %fd341, %fd320, %fd340, %fd279;
    fma.rn.f64  %fd342, %fd323, %fd340, %fd280;
    fma.rn.f64  %fd343, %fd325, %fd340, %fd281;
    fma.rn.f64  %fd344, %fd327, %fd340, %fd282;
    fma.rn.f64  %fd345, %fd329, %fd340, %fd283;
    fma.rn.f64  %fd346, %fd331, %fd340, %fd284;
```

Figure 9. A portion of the PTX code for the innermost loop of the fastest DGEMM kernel.

	Cholesky	QR (Householder reflectors)	LU (Gaussian Elimination)
PanelFactorize	xPOTF2 xTRSM	xGEQF2	xGETF2
TrailingMatrixUpdate	xSYRK2 xGEMM	xLARFB	xLASWP xTRSM xGEMM

Table 1. Routines for panel factorization and the trailing matrix update.

phases leads to a straightforward iterative scheme shown in Algorithm 1. Table 1 shows BLAS and LAPACK routines that should be substituted for the generic routine names in the algorithm.

Algorithm 1: Two-phase implementation of a one-sided factorization.

```
for $P_i \in \{P_1, P_2, \ldots, P_n\}$ do
    PanelFactorize($P_i$)
    TrailingMatrixUpdate($A^{(i)}$)
```

Algorithm 2: Two-phase implementation with a split update.

```
for $P_i \in \{P_1, P_2, \ldots\}$ do
    PanelFactorize($P_i$)
    TrailingMatrixUpdate$_{\text{Kepler}}$($A^{(i)}$)
    TrailingMatrixUpdate$_{\text{Phi}}$($A^{(i)}$)
```

The use of multiple accelerators for the computations complicates the simple loop from Algorithm 1: we have to split the update operation into multiple instances for each of the accelerators. This was done in Algorithm 2. Notice that PanelFactorize() is not split for execution on accelerators because it is considered a latency-bound workload which faces a number of inefficiencies on throughput-oriented devices. Due to their high performance rate exhibited on the update operation, and the fact that the update requires the majority of floating-point operations, it is the trailing matrix update that should be off-loaded. The problem of keeping track of the computational activities is exacerbated by the separation between the address spaces of main memory of the CPU and the devices. This requires synchronization between memory buffers and is included in the implementation shown in Algorithm 3.

Algorithm 3: Two-phase implementation with a split update and explicit communication.

```
for $P_i \in \{P_1, P_2, \ldots\}$ do
    PanelFactorize($P_i$)
    PanelSend$_{\text{Kepler}}$($P_i$)
    TrailingMatrixUpdate$_{\text{Kepler}}$($A^{(i)}$)
    PanelSend$_{\text{Phi}}$($P_i$)
    TrailingMatrixUpdate$_{\text{Phi}}$($A^{(i)}$)
```

The code has to be modified further to achieve closer to optimal performance. In fact, the bandwidth between the CPU and the devices is orders of magnitude too slow to sustain computational rates of modern accelerators.[4] The common technique to alleviate this imbalance is to use *lookahead* [27, 28].

Algorithm 4: Lookahead of depth 1 for the two-phase factorization.

```
PanelFactorize($P_1$)
PanelSend($P_1$)
TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$($P_1$)
PanelStartReceiving($P_2$)
TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$($R^{(1)}$)
for $P_i \in \{P_2, P_3, \ldots\}$ do
    PanelReceive($P_i$)
    PanelFactorize($P_i$)
    PanelSend($P_i$)
    TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$($P_i$)
    PanelStartReceiving($P_i$)
    TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$($R^{(i)}$)
PanelReceive($P_n$)
PanelFactor($P_n$)
```

Algorithm 4 shows a very simple case of lookahead of depth 1. The update operation is split into an update of the next panel, the start of the receiving of the next panel that just got updated, and an update of the rest of the trailing matrix R. The splitting is done to overlap the communication of the panel and the update operation. The complication of this approach comes from the fact that depending on the communication bandwidth and the accelerator speed, a different lookahead depth might be required for optimal overlap. In fact, the adjustment of the depth is often required throughout the factorization's runtime to yield good performance: the updates consume progressively less time when compared to the time spent in the panel factorization.

Since the management of adaptive lookahead is tedious, it is desirable to use a dynamic Direct Acyclic Graph (DAG) scheduler to keep track of data dependences and communication events. The only issue is the homogeneity inherent in most of the schedulers which is violated here due to the use of three different computing devices that we used. Also, common scheduling techniques, such as task stealing, are not applicable here due to the disjoint address spaces and the associated large overheads of communicating the stolen task's data across the PCI bus. These caveats are dealt with comprehensively in the remainder of the paper.

5. Lightweight Runtime for Heterogeneous Hybrid Architectures

In this section, we discuss the techniques that we developed in order to achieve an effective and efficient use of heterogeneous hybrid architectures. What we propose, considers both the

[4]The bandwidth for current generation PCI Express is at most 16 GB/s and the devices achieve over 1000 Gflop/s of floating-point performance.

higher ratio of execution and the hierarchical memory model of the modern accelerators and coprocessors. Taken together, it emerges as a multi-way heterogeneous programming model. We also redesign an existing superscalar task execution environment to handle to schedule and execute tasks on multi-way heterogeneous devices.

For our experiments, we consider shared-memory multicore machines with some collection of GPUs and MIC devices. Below, we present QUARK and the specific its modifications that facilitate execution on heterogeneous devices.

5.1. Task Superscalar Scheduling

Task-superscalar execution takes as an input a sequence of tasks in a form of a DAG and schedules them for execution in parallel while honoring the data dependences between the tasks at runtime. The dependences between the tasks are honored through the resolution of data hazards: *Read after Write* (RaW), *Write after Read* (WaR) and *Write after Write* (WaW). The dependences that form the DAG's edges are extracted from the serial formulation of the code by having the user include annotations on the data items when defining the tasks. The annotations indicate whether the data is to be read and/or written.

The RaW hazard, often referred to as the *true dependency*, is the most common one. It defines the relation between a task writing the data and another task reading that data. The reading task has to wait until the writing task completes. In contrast, if multiple tasks wish to read the same data, they need not wait on each other but can execute in parallel once the data is available. This is the classical case of Lamport's *concurrent* events [18]. Task-superscalar execution results in an asynchronous, data-driven execution that may be represented by a *Direct Acyclic Graph* (DAG), where the tasks are the nodes in the graph and the edges correspond to data movement between the tasks. Task-superscalar execution is a powerful tool for productivity. Since the original code looks as if it was serial it can be written and debugged in a serial environment. After the code becomes the input for the runtime system, the parallel execution avoids all the data hazards and thus preserves the correctness of the serial code, which in turn implies the parallel correctness of execution (granted, of course, that the runtime system does not introduce conditions that violate user's annotations).

Using task superscalar execution, the runtime can achieve parallelism by executing tasks with non-conflicting data dependences (e.g., simultaneous reads of data by multiple tasks). Superscalar execution also naturally enables lookahead into the serial code, since the tasks from the DAG representation can be executed as soon as their dependences are satisfied and not necessarily in the order they were inserted into the DAG from the the serial code.

5.2. Lightweight Runtime Environment

QUARK (QUeuing and Runtime for Kernels) is the lightweight runtime environment chosen for this research, since it provides an API that allows us to enforce at a low level a precise task placement. QUARK is a data-driven dynamic superscalar runtime environment with a simple API for serial task insertion and parallel execution on multicore processors. It is the dynamic runtime engine used within the PLASMA linear algebra library and has been shown to provide high productivity for code development and good performance on modern multicore processor systems [17, 31, 32].

6. Efficient and Scalable Programming Model Across Multiple Devices

In this section, we discuss the programming model that raises the level of abstraction above the hardware and its accompanying software stack to offer a uniform approach for algorithmic development. We describe the techniques that we developed in order to achieve an effective use of multi-way heterogeneous devices. Our proposed techniques consider both, the higher ratio of execution and the hierarchical memory model of the new emerging accelerators and coprocessors.

6.1. Supporting Heterogeneous Platforms

GPU accelerators and coprocessors have a very high computational peak compared to CPUs. For simplicity, we refer to both GPUs and coprocessors as accelerators. Also, different types of accelerators have different capabilities, which makes it challenging to develop an algorithm that can achieve high performance and exhibit good scalability. From the hardware perspective, an accelerator communicates with the CPU using I/O commands and DMA memory transfers, whereas from the software standpoint, the accelerator is a platform presented through a programming interface. The key features of our model are the processing unit capability (each of the CPUs, GPUs, Xeon Phi is assigned one such capability), the memory access overhead, and the communication cost. As with CPUs, the access time to the device memory for accelerators is slow compared to peak performance (still the accelerator memory tends to be many times faster than the CPU memory). CPUs try to improve the effect of the long memory latency and bandwidth constraint by using hierarchical caches. This does not solve the slow memory problem completely but is often effective in our target factorizations. On the other hand, accelerators use multithreading operations that access large data sets that would overflow the size of most caches. The idea is that when the accelerator's thread unit issues an access to the device memory, that thread unit stalls until the memory returns a value. In the meantime, the accelerator's scheduler switches to another hardware thread, and continues executing that thread. In this way, the accelerator exploits program parallelism to keep functional units busy while the memory fulfills the past requests. By comparison with CPUs, the device memory delivers higher absolute bandwidth (effectively around 180 GB/s for Xeon Phi and 160 GB/s for Kepler K20c). To side-step the issues related to the memory constraints, we developed a strategy that prioritizes the data-intensive operations to be executed by the accelerator and keep the memory-bound ones for the CPUs since the hierarchical caches with out-of-order superscalar scheduling are more appropriate to handle it. Moreover, in order to keep the accelerators busy, we redesign the kernels and propose dynamically guided data distribution to exploit enough parallelism to keep the accelerators and processors occupied at the same time.

From the programming model point of view, it is hard to hide the distinction between the two kinds of parallel devices. For that, we convert each algorithm into a host part and an accelerator part. Each routine to be run on the accelerator must be extracted into a separate hardware-specific kernel function. The kernel itself may have to be carefully optimized for the accelerator, including unrolling loops, replacing some memory-bound operations by compute-intensive ones even if it has a marginal extra cost, and also arranging its tasks to use the device memory efficiently. The host code must manage the device memory allocation, the CPU-device data movement, and the kernel invocation. We redesigned the QUARK runtime engine in order to alleviate the programming burden and to simplify scheduling. This often allows us to maintain

Algorithm 5: Cholesky implementation for multiple devices.

```
Task_Flags panel_flags = Task_Flags_Initializer
Task_Flag_Set(&panel_flags, PRIORITY, 10000)
$\boxed{\text{memory-bound} \rightarrow \text{locked to CPU}}$
Task_Flag_Set(&panel_flags, BLAS2, 0)
for $k \in \{0, nb, 2 \times nb, \ldots, n\}$ do
    $\boxed{\text{Factorization of the panel dA(k:n,k)}}$
    Cholesky on the tile dA(k,k)
    TRSM on the remaining of the panel dA(k+nb:n,k)
    DO THE UPDATE: SYRK task has been split into a set of parallel compute
    intensive GEMM to increase parallelism and enhance the performance. Note that
    the first GEMM consists of the update of the next panel, thus the scheduler check
    the dependency and once finished it can start the panel factorisation of the next
    loop on the CPU.
    if $panel\_m > panel\_n$ then
        $\boxed{\text{SYRK with trailing matrix}}$
        for $j \in \{k + nb, k + 2nb, \ldots, n\}$ do
            GEMM dA(j:n,k) $\times$ dA(j,k)$^T$ = dA(j:n,j)
```

a single source version that handles different types of accelerators either independently or when mixed together. Our intention is for our model to simplify most of the hardware details, but, at the same time, give us finer levels of control.

Algorithm 5 shows the pseudocode for the Cholesky factorization as an algorithm designer would see it. It consists of a sequential code that is simple to comprehend and independent of the architecture. Each of the calls represents a task that is inserted into the scheduler, which stores it to be executed when all of its dependencies are satisfied. Each task by itself consists of a call to a kernel function that results in execution on either a CPU or an accelerator – commonly it would called a *kernel*. We tried to hide the differences between hardware and let the QUARK engine handle the transfer of data automatically. In addition, we developed low-level optimizations for the accelerators, in order to accommodate hardware- and library-specific tuning and prerequisites. Moreover, we implemented a set of directives that are evaluated at runtime in order to fully map the algorithm to the hardware and to run close to the peak performance of the system. Using these strategies, we can more easily develop simple and portable code, that can run on different heterogeneous architecture and lets the scheduling and execution engine do much of the tedious bookkeeping.

In the discussion below, we will describe in detail the optimization techniques we propose, and explore some of the features of our model and also describe some directives that help with tuning performance in an easy fashion. The study here is described in the context of the Cholesky factorization but it can be easily applied to other algorithms such as the QR decomposition and LU factorization.

6.2. Resource Capability Weights (CW)

There is no simple way to express the difference in the workload-capabilities between the CPUs and accelerators in a generic fashion. Clearly, we cannot balance the load, if we treat them

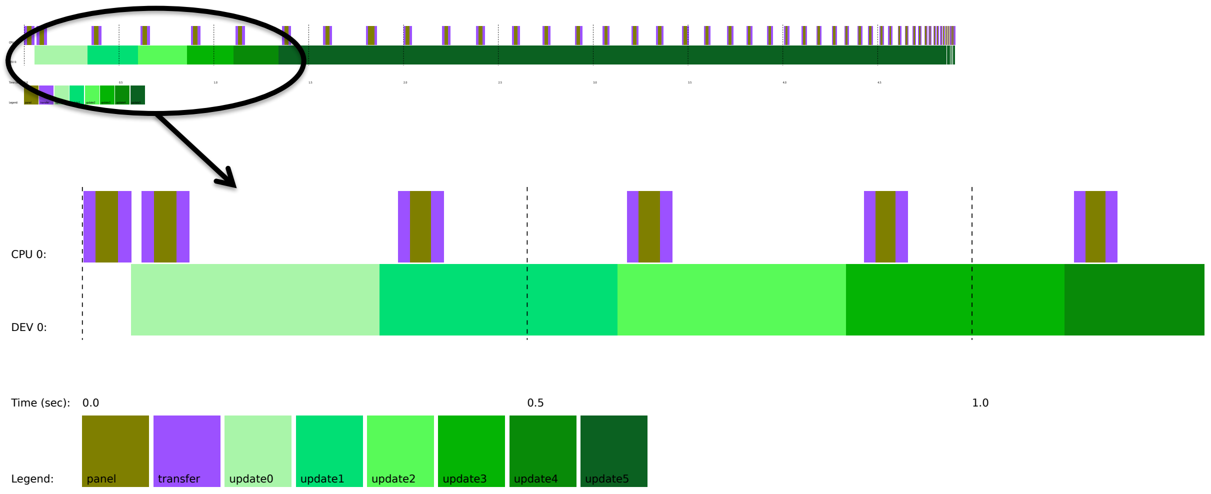


Figure 10. A trace of the Cholesky factorization on a multicore CPU with a single GPU K20c, assigning the panel factorization task to the CPU (brown task) and the update to the GPU (green task) for a matrix of size 20,000.

as similarly equipped peers and assign them equivalent amounts of work. This naïve strategy would cause the accelerator to be idle most of the time. In our model we propose to assign the latency-bound operations to the CPUs and the compute-intensive ones to accelerators. In order to support multi-way heterogeneous hardware, QUARK was extended with a mechanism for distributing tasks based on the individual capabilities of each device. For each device i and each kernel type k, extended version of QUARK maintains an α_{ik} parameter which corresponds to the effective performance rate that can be achieved on that device. In the context of linear algebra algorithms, this means that we need an estimation of performance for Level 1, 2, and 3 BLAS operations. This can be done either by the developer during the implementation where the user gives a directive to QUARK which indicates that the kernel is either bandwidth-bound or compute-bound function (as shown in Algorithm 5 with a call to Task_Flag_Set with a BLAS2 argument) or estimated according to the volume of data and the elapsed time of a kernel by the QUARK engine at runtime.

Figure 10 shows the execution trace of the Cholesky factorization on a single multicore CPU and a K20c GPU of System A (hardware labels and description is given in Section 7.1). We see that the memory-bound kernel (e.g., the panel factorization for the Cholesky algorithm) has been allocated to the CPU while the compute-bound kernel (e.g., the update performed by DSYRK) has been allocated to the accelerator. The initial data is assumed to be on the device, and when the CPU is executing a task, the data need to be copied from the device and sent back to be used for updating the trailing matrix. The data transfer is represented by the *purple* color in the trace. The CPU panel computation is represented by the *gold* color. The trailing matrix update are depicted in *green*. For clarity, we varied the intensity of the green color representing the update from light to dark for the first 5 steps of the algorithm. From this trace, we can see that the GPU is kept busy all the way until the end of execution. The use of the lookahead technique described in Algorithm 4, does not require any extra effort since it is handled by the QUARK engine through the dependence analysis. The engine will ensure that the next panel (panel of step $k+1$) is updated as soon as possible by the GPU in order to be sent to the CPU to be factorized while the GPU is continuing the update of the trailing matrix of step k. Also, the QUARK engine manages the data transfer to and from the CPU automatically. The advantage of such a strategy is not only to hide the data transfer cost between the CPU and GPU (since it is overlapped with the GPU computation), but also to keep the GPU's CUDA streams busy by providing enough tasks to execute. This is highlighted in Figure 10, where we can see that the panel of step 1 is quickly updated by the GPU and sent to the CPU to be factorized and sent back to the GPU, which is a perquisite to perform the trailing matrix update of step 1, before the GPU has already finished the update of trailing matrix of step 0, and so on.

However, it is clear that we can improve this further by fully utilizing all the available resources, particularly exploiting the idle time of the CPUs (white space in the trace). Based on the parameters defined above, we can compute a resource capability-weights for each task that reflects the cost of executing it on a specific device. This cost is based on the communication cost (if the data has to be moved) and on the type of computation (memory-bound or compute-bound) performed by the task. For a task that requires an $n \times n$ data, we define its computation type to be from one of the Levels of BLAS (either 1, 2, or 3). Thus the two factors are simply defined as:

$$\begin{aligned}\text{communication} &= \frac{n \times n}{\text{bandwidth}}\\ \text{computation} &= n^k \times \alpha_{ik}\\ &\text{where } k \text{ is the Level } k \text{ BLAS}\end{aligned}$$

The capability-weights for a task is then the ratio of the total cost of the task on one resource versus the cost on another resource. For example, the capability-weights for the update operation (a Level 3 BLAS) from the execution shown in Figure 10 is around 1 : 10 which means that the GPU can execute 10 times as many update tasks as the CPU.

6.3. Greedy Scheduling Using Capability-Weights

As each task is inserted into the runtime, it is assigned to the resource with the largest remaining capability-weights. This greedy heuristic takes into account the capability-weights of the resource as well as the current number of waiting tasks *preferred* to be executed by this resource. For example, for the CPU, the panel tasks are memory-bound and thus are preferred to be executed always on the CPU side. The heuristic tries to maintain the ratios of the capability-weights across all the resources.

In Figure 11, we can see the effect of using capability-weights to assign a subset of the update tasks to the CPU. The GPU remains as busy as before, but now the CPU can contribute to the computation and does not have as much idle time as before. Careful management of the capability-weights ensures that the CPU does not take any work that would cause a delay to the GPU, since that would negatively affect the performance. We also plot in Figure 12 the performance gain obtained when using this technique. The graph shows that on a 16-core Sandy Bridge CPU, we can achieve a gain of around 100 Gflop/s (red curve) and 80 Gflop/s (blue curve) when enabling this technique when using a single Fermi M2090 and Kepler K20c GPU on System A and B, respectively (the hardware is described in §7.1).

6.4. Improved Task Priorities

In order to highlight the importance of task priorities, we recall, that the panel factorization task of most of the one-sided factorizations (the Cholesky, QR, and LU decompositions) is on

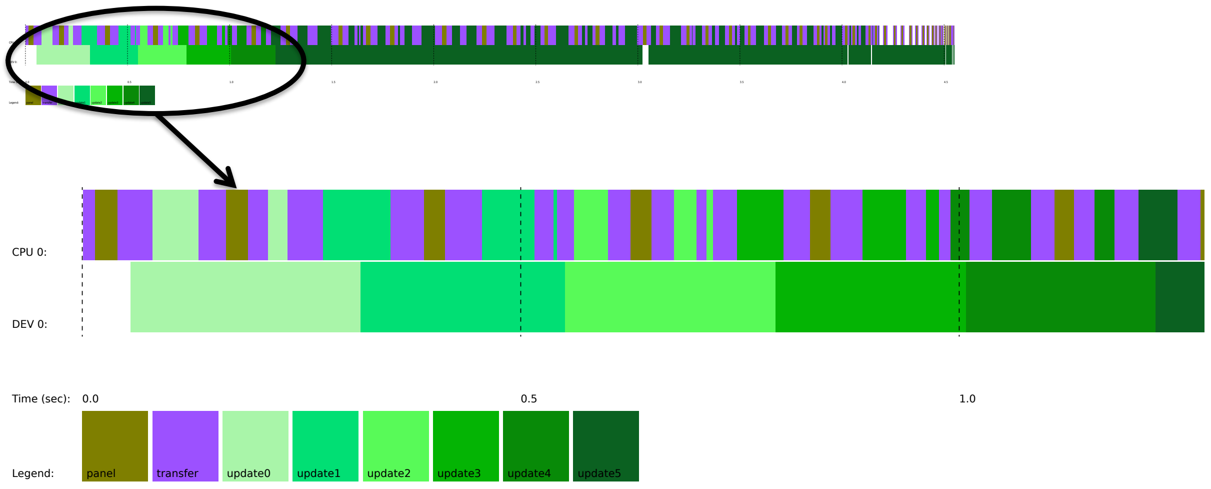


Figure 11. A trace of the Cholesky factorization on 16-core, 2-socket Sandy Bridge CPU and a K20c GPU, using capability-weights to distribute tasks.

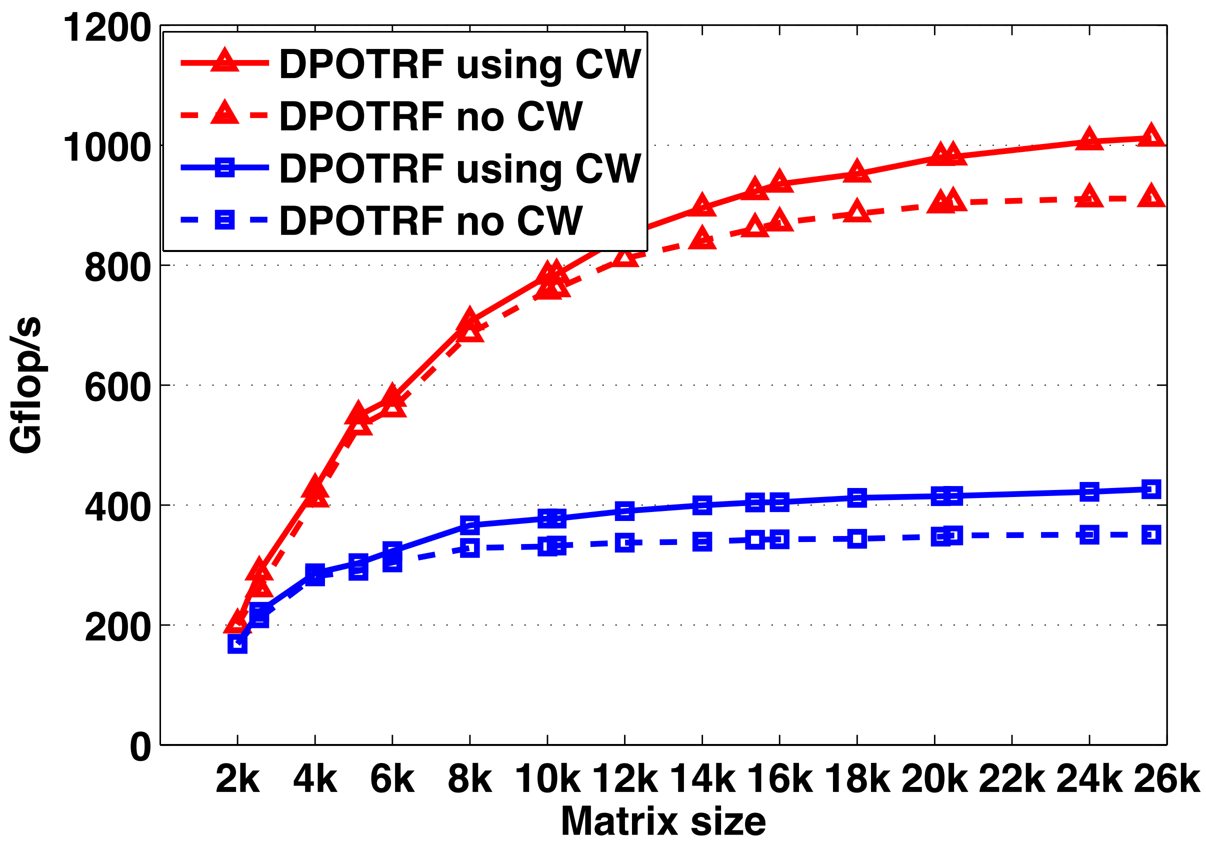


Figure 12. Performance comparison of the Cholesky factorization when using the capability-weights (CW) to distribute tasks among the heterogeneous hardware, on a node of 16 Sandy-Bridge CPU and either a Kepler (K20c) of system A "red curve" or a Fermi(M2090) of system B "blue curve".

the critical path of the execution. In other words, only if a panel computation is done in its entirety, its corresponding update computation (compute-bound operation) may proceed. In the traces in Figures 10 and 11, it can be observed that the panel factorization on the CPU occurs at regular intervals (e.g., the lookahead depth is one). By changing the priority of the panel factorization tasks (using QUARK's task priority flags as mentioned in Algorithm 5), the panel factorization can be executed earlier. This increases the lookahead depth that the algorithm exposes, increasing parallelism so that there are more update tasks available to be executed by the device resources. Using priorities to improve lookahead results in approximately 5% improvement in the overall performance of the factorization. Figure 13 shows the update tasks being executed earlier in the trace.

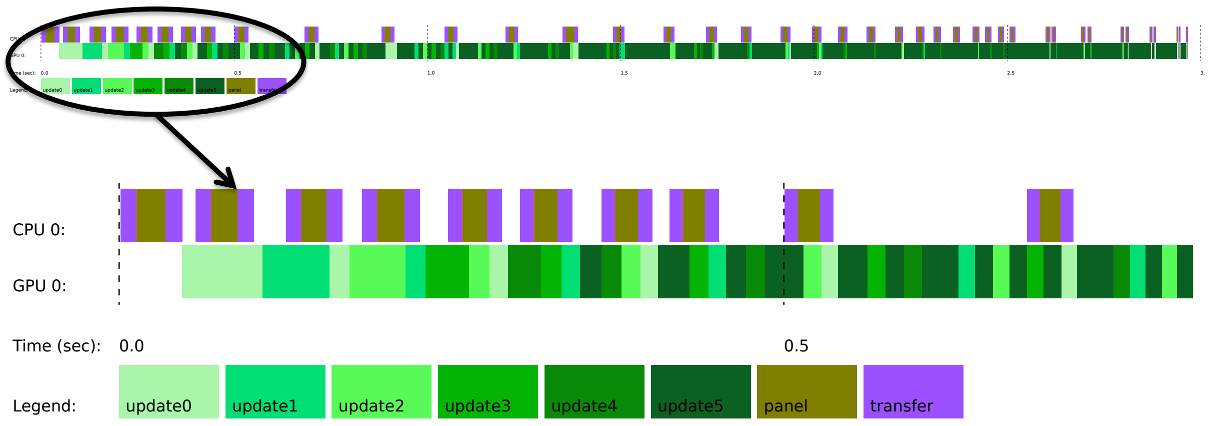


Figure 13. Trace of the Cholesky factorization on multicore 16 Sandy Bridge CPU and a K20c GPU, using priorities to improve lookahead.

6.5. Data Layout

When we proceed to the multiple accelerator setup, the data is initially distributed over all the accelerators in a 1-D block-column cyclic fashion, with an approximately equal number of columns assigned to each. Note that the data is allocated on each device as one contiguous memory block with the data being distributed as columns within the contiguous memory segment. This contiguous data layout allows large update operations to take place over a number of columns via a single Level 3 BLAS operation, which is far more efficient than having multiple calls with block columns.

6.6. Hardware-Guided Data Distribution (HGDD)

The experiments so far showed that the standard 1-D block cyclic data layout was hindering performance in heterogeneous multi-accelerator environments. Figure 14 shows the trace of the Cholesky factorization for a matrix of size 30, 000 on System D (consisting of a Kepler K20c, a Xeon Phi (MIC) and Kepler K20-beta that has half the K20c performance). The trace shows that the execution flow is bound by the performance of the slowest machine (the beta K20, second row) and thus we expect lower performance on this machine.

We propose to re-adjust the data layout distribution to be hardware-guided by the use of the capability-weights. Using the QUARK runtime, the data is either distributed or redistributed in an automatic fashion so that each device gets the appropriate volume of data to match its capabilities. So, for example, for System D, using capability weights of K20c:MIC:K20-beta of 10:8:5 would result in a cyclic distribution of 10 columns of data being assigned to the K20c, for each 8 columns assigned to the MIC, and each 5 columns assigned to the K20beta. The superscalar execution environment can do this capability-weighted data assignment at runtime.

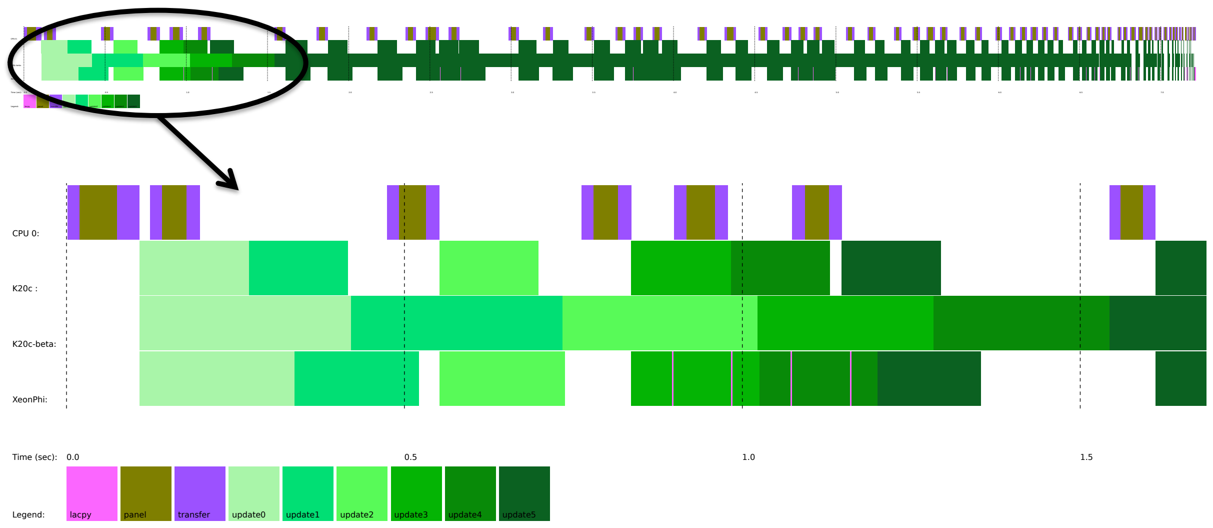


Figure 14. Cholesky factorization trace on multicore CPU and multiple accelerators (a Kepler K20c, a Xeon Phi, and an old K20-beta-release), using 1D block cyclic data distribution without enabling heterogeneous hardware-guided data distribution.

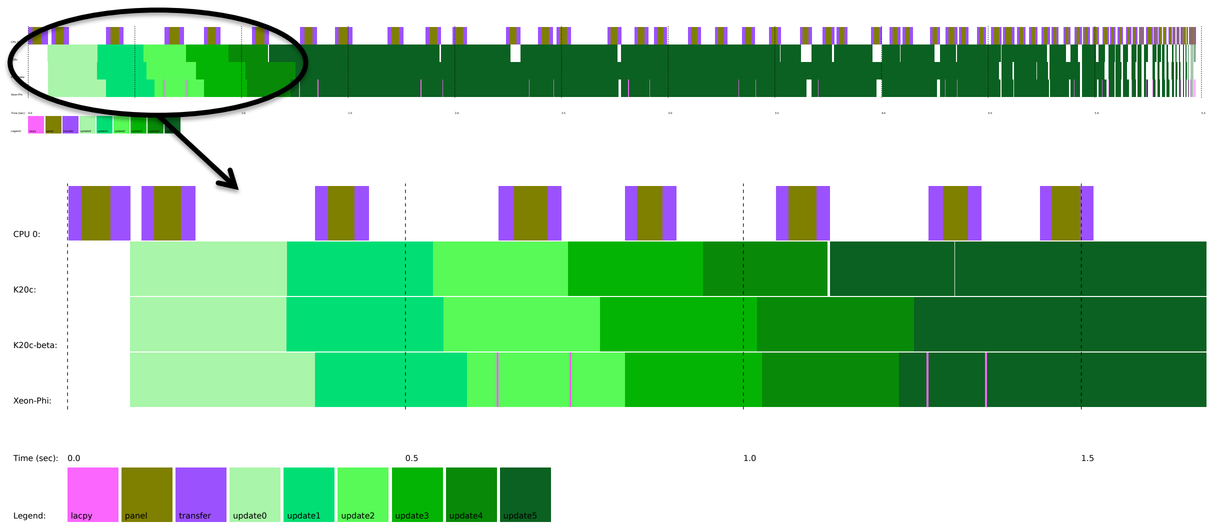


Figure 15. Cholesky factorization trace on multicore CPU and multiple accelerators (a Kepler K20c, a Xeon Phi, and an old K20-beta-release), using the heterogeneous hardware-guided data distribution techniques (HGDD) to achieve higher hardware usage.

Figure 15 shows the trace of the Cholesky factorization for the same example as above (a matrix of size $30K$ a node of the system D) when using the hardware-guided data distribution (HGDD) strategy. It is clear that the execution trace is more compact meaning that all the heterogeneous hardware are fully loaded by work and thus one can expect an increase in the total performance. For that we represent in Figure 16 and Figure 17 the performance comparison of the Cholesky factorization and the QR decomposition when using the HGDD strategy. The curves in blue show the performance obtained for a one K20c and one XeonPhi experiments. The dashed line correspond to the standard 1-D block-column cyclic distribution while the continuous line illustrate the HGDD strategy. We observe that we can reach an improvement of about 200-300 Gflop/s when using the HGDD technique. Moreover, when we add one more heterogeneous device (the K20beta), here it comes to the complicated hardware situation, we can notice that the standard distribution do not exhibit any speedup. The dashed red curve that represents the performance of the Cholesky factorization using the standard data distribution on the three devices of System D behaves closely and less efficient that the one obtained with the same standard distribution on two devices (dashed blue curve). This was expected, since adding one more device with lower capability may decrease the performance as it may slowdown the fast device. The blue and red curves in Figure 16 illustrate that the HGDD technique exhibits a very good scalability for both algorithms. The graph shows that the performance of the algorithm is not affected by the heterogeneity of the machine, our proposed implementation is appropriate to maintain a high usage of all the available hardware.

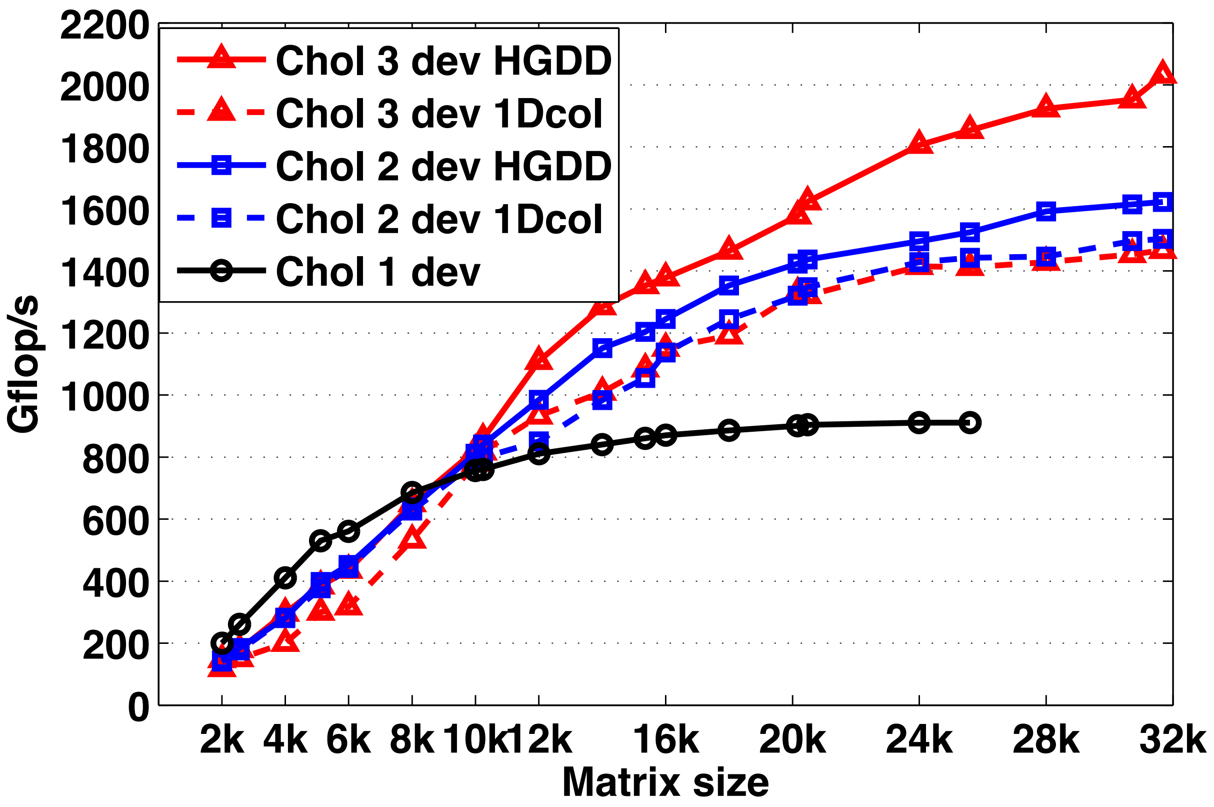


Figure 16. Performance comparison of the Cholesky factorization when using the hardware-guided data distribution techniques versus a 1-D block-column cyclic, on heterogeneous accelerators consisting of a Kepler K20c (1dev), a Xeon Phi (2dev), and an old K20-beta-release (3dev).

6.7. Hardware-Specific Optimizations

One of the main enablers of good performance is optimizing the data communication between the host and accelerator. Another one is to redesign the kernels to exploit the inherent parallelism of the accelerator even by adding extra computational cost.

In this section, we describe the development of our heterogeneous multi-device kernels, which includes the consideration of hardware constraints, the methods of achieving fast communication, and approaches that allow the algorithms to reach good scalability on both CPUs and accelerators. Our target algorithms in this study are the one sided-factorization (Cholesky, QR, and LU).

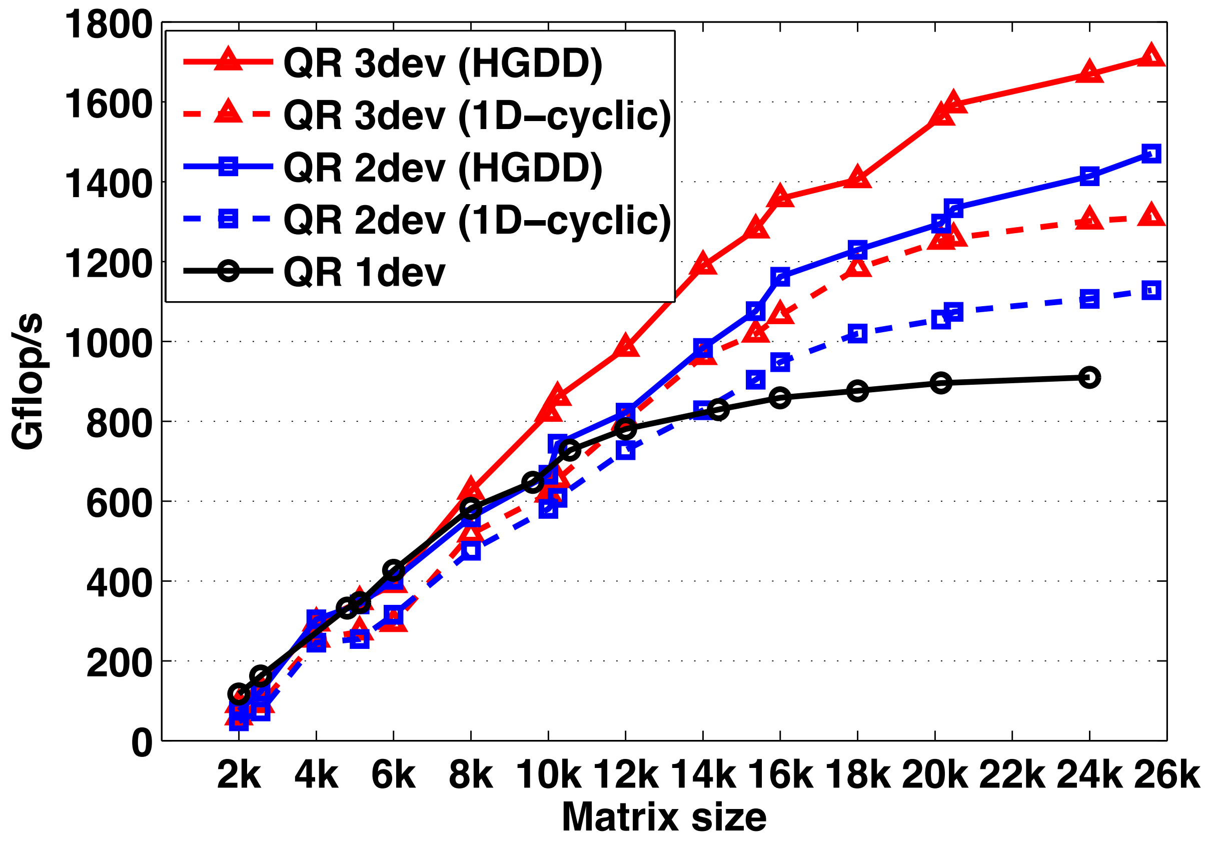


Figure 17. QR performance comparison for hardware-guided data distribution techniques *vs.* 1-D block-column cyclic, on heterogeneous accelerators consisting of a Kepler K20c (1dev), a Xeon Phi (2dev), and an old K20-beta-release (3dev).

6.7.1. Redesigning BLAS Kernels to Exploit Parallelism and Minimize the Memory Movement

The Hermitian rank-k update (SYRK) required by the Cholesky factorization implements the operation $A^{(k)} = A^{(k)} - PP^{*}$, where A is an n by n Hermitian trailing matrix of step k, and P is the result of the panel factorization done by the CPU. After distribution, the portion of A on each accelerator no longer appears as a symmetric matrix, but instead has a ragged structure shown in Figure 18.

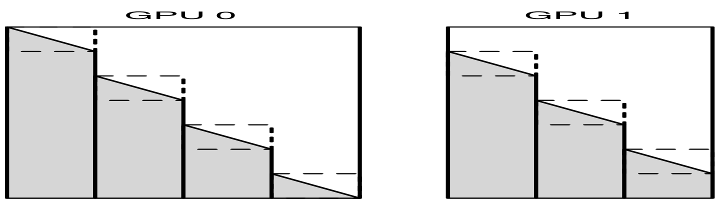


Figure 18. Block-cyclic data distribution. Shaded areas contain valid data. Dashed lines indicate diagonal blocks.

Because of this uneven storage, the multi-device SYRK cannot be assembled purely from regular SYRK calls on each device. Instead, each block column must be processed individually. The diagonal blocks require special attention. In the BLAS standard, elements above the diagonal are not accessed; the user is free to store unrelated data there and the BLAS library will not alter it. To achieve this, one can use a SYRK to update each diagonal block, and a GEMM to update the remainder of each block column below the diagonal block. However, these small SYRK operations have little parallelism and so are inefficient on an accelerator. This can be improved to some degree by using either multiple streams (GPU) or a pragma (MIC) to execute several SYRK updates simultaneously. However, because we have copied the data to the device, we can consider the space above the diagonal to be a scratch workspace. Thus, we update the entire block column, including the diagonal block, writing extra data into the upper triangle of the diagonal block, which is subsequently ignored. We do extra computation for the diagonal block, but gain efficiency overall by launching fewer BLAS kernels on the device and using the more efficient GEMM kernels, instead of small SYRK kernels, resulting in overall 5-10% improvement in performance.

6.7.2. Improving Coalesced Data Access

The LU factorization uses the LASWP routine to swap two rows of the matrix. However, this simple data copy operation might drop the performance of such routines on accelerator architecture since it is recommended that a set of threads read coalescent data from the memory which is not the case for a row of the matrix. Such routine needs to be redesigned in order to overcome this issue. The device data on the GPU is transposed using a specialized GPU kernel, and is always stored in a transposed form, to allow coalesced read/write when the LASWP function is involved. We note that the transpose does not affect any of the other kernels (GEMM, TRSM) required by the LU factorization. The coalesced reads and writes improve the performance of the LASWP function 1.6 times.

6.7.3. Enabling Specific Architecture Kernels

The size of the main Level 3 BLAS kernel that has to be executed on the devices is yet another critical parameter to tune. Every architecture has its own set of input problem sizes that achieve higher than average performance. In the context of one-sided algorithms, all the Level 3 BLAS operations depend on the size of the block panel. On the one hand, a small panel size lets the CPUs finish early but leads to lower Level 3 BLAS performance on the accelerator. On the other hand, a large panel size burdens the CPU and the CPU computation is too slow to be fully overlapped with the accelerator work. The panel size corresponds to a trade-off between the degree of parallelism and the amount of data reuse. In our model, we can easily tune this parameter or allow the runtime to autotune it by varying the panel size throughout the factorization.

6.7.4. Trading Extra Computation for Higher Performance Rate

The implementation that is discussed here is more related to the hardware architecture based on hierarchical memory. The LARFB routine used by the QR decomposition consists of two GEMM and one TRMM operation. Since accelerators are better at handling compute-bound tasks, we replace the TRMM by GEMM for computational efficiency, thus achieving 5-10% higher performance when executing these kernels on the accelerator.

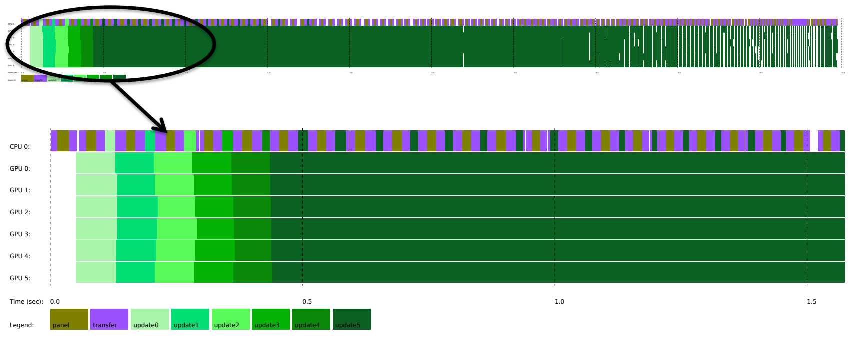


Figure 19. Trace of Cholesky factorization on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

7. Experimental Setup and Results

7.1. Hardware Description and Setup

Our experiments were performed on a number of shared-memory systems available to us at the time of writing of this paper. They are representative of a vast class of servers and workstations commonly used for computationally intensive workloads. We conducted our experiments on four different systems all of each equipped with an Intel multicore processor in a dual-socket configuration with 8-core Intel Xeon E5-2670 (Sandy Bridge) processors in a socket, each running at 2.6 GHz. Each socket had 24 MiB of shared L3 cache, and each core had a private 256 KiB Level 2 and 64 KB Level 1 cache. The system is equipped with 52 GB of memory and the theoretical peak in double precision is 20.8 Gflop/s per core.

- System A is also equipped with six NVIDIA K20c cards with 5.1 GB per card running at 705 MHz, connected to the host via two PCIe I/O hubs at 6 GB/s bandwidth.
- System B is equipped with three NVIDIA M2090 cards with 5.3 GB per card running at 1.3 GHz, connected via two PCIe I/O hubs at 6 GB/s bandwidth.
- System C is also equipped with three Intel Xeon Phi cards with 15.8 GB per card running at 1.23 GHz, and achieving a double precision theoretical peak of 1180 Gflop/s, connected via four PCIe I/O hubs at 6 GB/s bandwidth.
- System D is a heterogeneous system equipped with a K20c, and a Intel Xeon Phi card as the ones described above, and also an old K20c beta release 3.8 GB running at 600 MHz. All are connected via four PCIe I/O hubs at 6 GB/s bandwidth.

A number of high performance software packages were used for the experiments. On the CPU side, we used the MKL (Math Kernel Library) [15]. On the Xeon Phi side, we used the MPSS 2.1.5889-16 as the software stack, icc 13.1.1 20130313 which comes with the Composer XE 2013.4.183 suite as the compiler, and finally on the GPU accelerator we used CUDA version 5.0.35.

7.2. Mixed MIC and GPU Results

Getting good performance across multiple accelerators remains a challenging problem that we address with the algorithmic and programming techniques described in this paper. The efficient strategies used to schedule and exploit parallelism across multi-way heterogeneous platforms will be highlighted in this subsection through the extensive set of experiments that we performed on the four systems that we had access to.

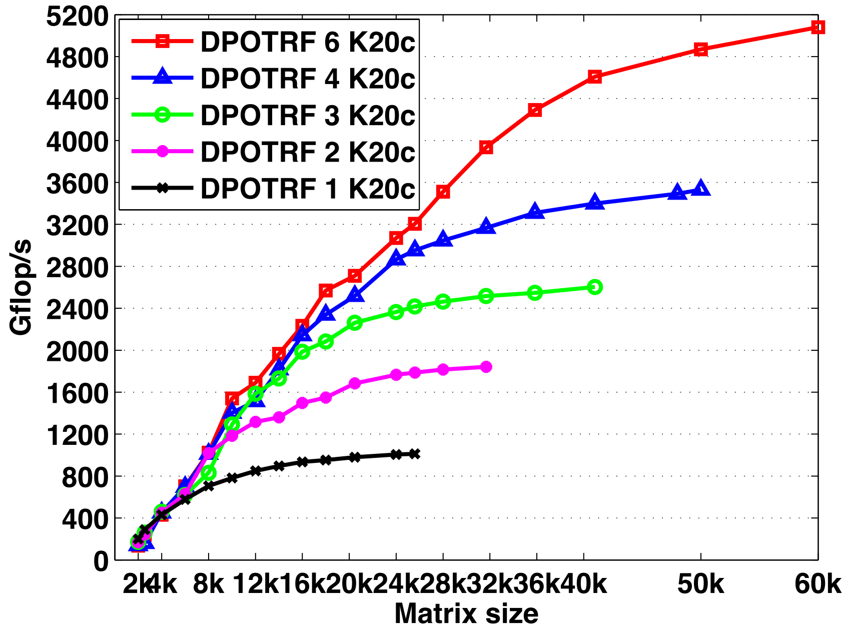


Figure 20. Performance scalability of Cholesky factorization on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

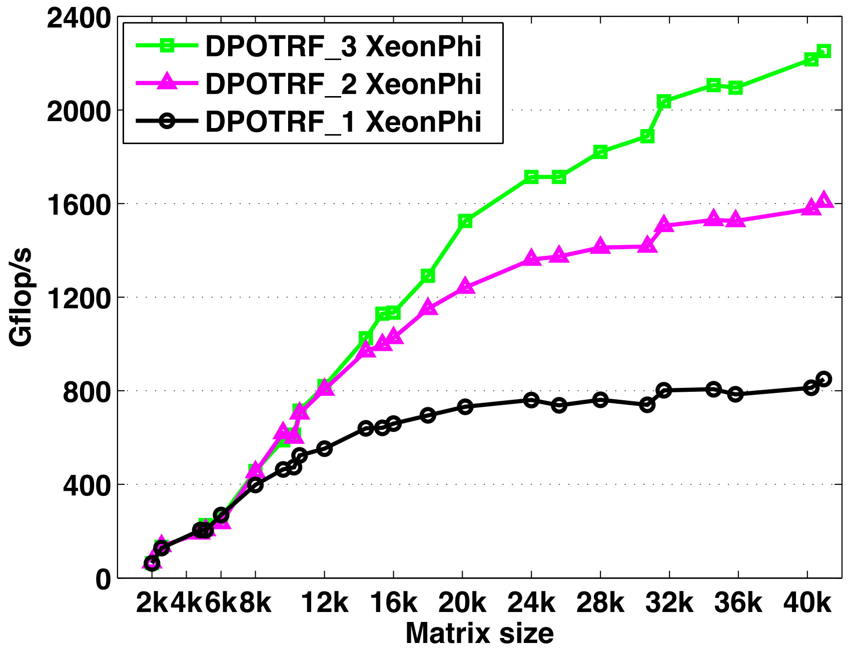


Figure 21. Performance scalability of Cholesky factorization on multicore CPU and multiple accelerators (up to 3 XeonPhi).

Figure 19 show a snapshot of the execution trace of the Cholesky factorization on System A for a matrix of size 40K using six GPUs K20c. As expected the pattern of the trace looks compressed which means that our implementation is able to schedule and balance the tasks on the whole six GPUs devices. Figures 20 and 21 show the performance scalability of the Cholesky factorization in double precision on either the 6 GPUs of System A or the 3 Xeon Phi of System C. The curves show performance in terms of Gflop/s. We note that this also reflects the elapsed time, e.g., a performance that is two times higher, corresponds to an elapsed time that is two times shorter. Our heterogeneous multi-device implementation shows very good

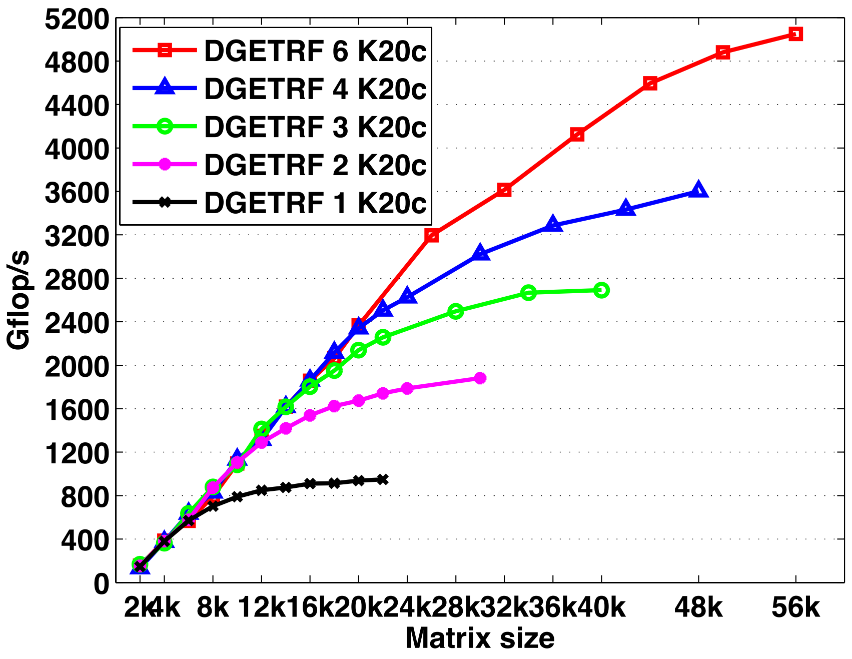


Figure 22. Performance scalability of the LU decomposition on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

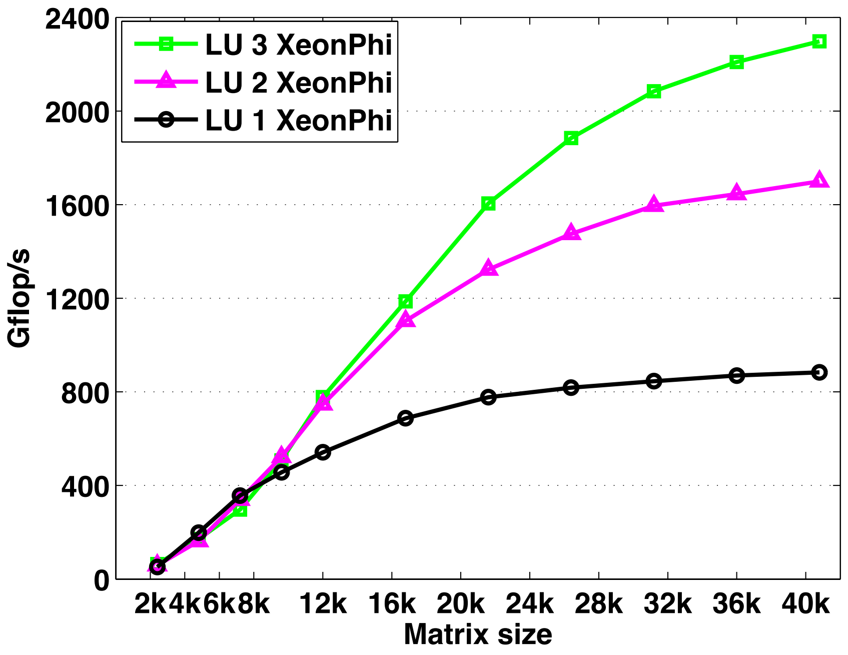


Figure 23. Performance scalability of the LU decomposition on multicore CPU and multiple accelerators (up to 3 Xeon Phi).

scalability. On System A, for a 60,000 matrix, the Cholesky factorization achieves 5.1 Tflop/s when using the 6 Kepler K20c GPUs. We observe similar performance trends when using System C. For a matrix of size 40,000, the Cholesky factorization reaches up to 2.3 Tflop/s when using the 3 Intel Xeon Phi coprocessors. Figure 22 depicts the performance scalability of the LU factorization on System A while Figure 23 shows its performance on System C. Similarly to the Cholesky factorization, the LU factorization achieves around 5.2 Tflop/s on the System A using the 6 Kepler K20c GPUs for a matrix of size 56,000, and it also reaches 2.4 Tflop/s on the

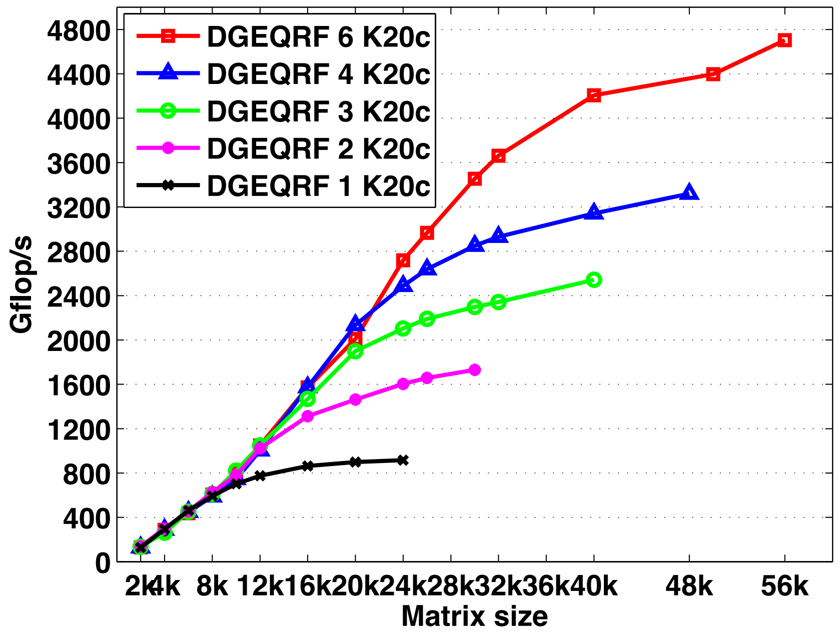


Figure 24. Performance scalability of the QR decomposition on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

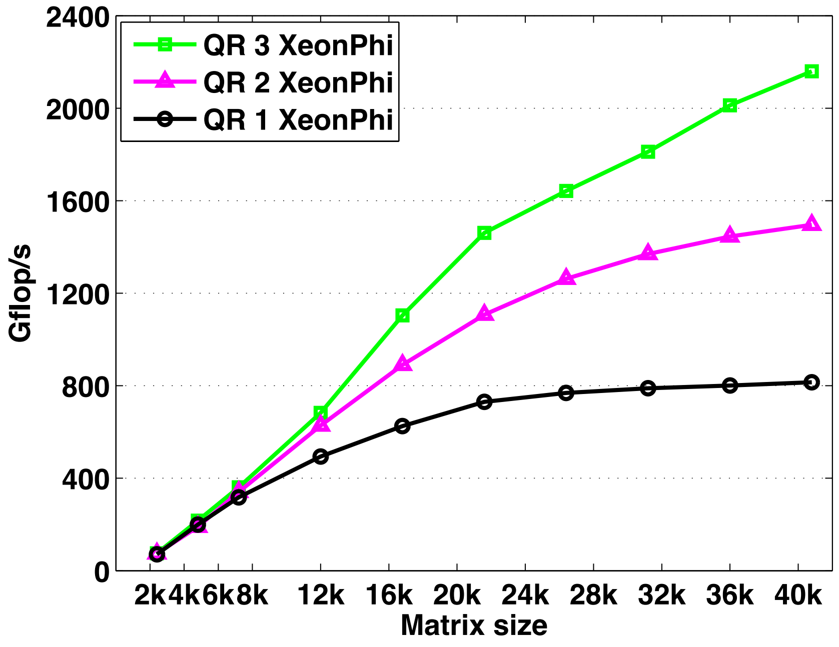


Figure 25. Performance scalability of the QR decomposition on multicore CPU and multiple accelerators (up to 3 Xeon Phi).

System C using the 3 Intel Xeon Phi coprocessors for a matrix of size 40,000. Figure 24 depicts the performance scalability of the QR factorization on System A and Figure 25 shows also the obtained results on System C. For a matrix of size 56,000, the QR factorization reaches around 4.7 Tflop/s on the System A using the 6 Kepler K20c GPUs and 2.2 Tflop/s on the System C using the 3 Intel Xeon Phi coprocessors.

8. Conclusions and Future Work

We designed algorithms and a programing model for developing high-performance dense linear algebra in multi-way heterogeneous environments. In particular, we presented best practices and methodologies from the development of high-performance DLA for accelerators. We also showed how judicious modifications to task superscalar scheduling were used to ensure that we meet two competing goals:

1. to obtain high fraction of the peak performance for the entire heterogeneous system,
2. to employ a programming model that would simplify the development.

We presented initial implementations of two algorithms. Future work will include merging MAGMA's [19] CUDA, OpenCL, and Intel Xeon Phi development branches into a single library using the new programming model.

This research was supported in part by the National Science Foundation under Grant OCI-1032815 and Subcontract RA241-G1 on NSF Prime Grant OCI- 0910735, DOE under Grants DE-SC0004983 and DE-SC0010042, NVIDIA and Intel.

References

1. Intel Xeon Phi Coprocessor System Software Developers Guide. http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-system-software-developers-guide.

2. E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA Users Guide. Technical report, ICL, University of Tennessee, 2010.

3. Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 271–276, New York, NY, USA, 2012. ACM.

4. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

5. Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 735–736, New York, NY, USA, 2009. ACM.

6. Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

7. A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, October 1966.

8. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

9. Chongxiao Cao, Jack Dongarra, Peng Du, Mark Gates, Piotr Luszczek, and Stanimire Tomov. clMAGMA: High Performance Dense Linear Algebra with OpenCL. In *International Workshop on OpenCL, IWOCL 2013*, Atlanta, Georgia, USA, May 13-14 2013.

10. Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proceedings of the nineteenth annual ACM symposium on parallel algorithms and architectures*, SPAA '07, pages 116–125, New York, NY, USA, 2007. ACM.

11. NVIDIA CUBLAS library. https://developer.nvidia.com/cublas.

12. Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire Tomov. Portable HPC Programming on Intel Many-Integrated-Core Hardware with MAGMA Port to Xeon Phi. In *10th International Conference on Parallel Processing and Applied Mathematics, PPAM 2013*, Warsaw, Poland, September 8-11 2013.

13. Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

14. Carlos H. González and Basilio B. Fraguela. A framework for argument-based task synchronization with automatic detection of dependencies. *Parallel Computing*, 39(9):475 – 489, 2013. Novel On-Chip Parallel Architectures and Software Support.

15. Intel. Math Kernel Library. http://software.intel.com/intel-mkl/.

16. Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

17. J. Kurzak, P. Luszczek, A. YarKhan, M. Faverge, J. Langou, H. Bouwmeester, and J. Dongarra. Multithreading in the PLASMA Library. In *Handbook of Multi and Many-Core Processing: Architecture, Algorithms, Programming, and Applications*, Computer and Information Science Series. Chapman and Hall/CRC, April 26 2013.

18. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

19. MAGMA library. http://icl.cs.utk.edu/magma/.

20. R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *Int. J. High Perf. Comput. Applic.*, 24(4):511–515, 2010. http://dx.doi.org/10.1177/1094342010385729 (DOI:~10.1177/1094342010385729).

21. Chris J. Newburn, Rajiv Deodhar, Serguei Dmitriev, Ravi Murty, Ravi Narayanaswamy, John Wiegert, Francisco Chinchilla, and Russell McGuire. Offload compiler runtime for the intel xeon phitm coprocessor. In *ISC*, pages 239–254, 2013.

22. Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pages 142–151. IEEE, 2008.

23. M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: a high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993. http://dx.doi.org/10.1109/2.214440 (DOI:~10.1109/2.214440).

24. J. E. Rodrigues. A graph model for parallel computations. Technical Report MIT/LCS/TR-64, MIT, Cambridge, MA, USA, September 1969.

25. Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM.

26. Fengguang Song, Stanimire Tomov, and Jack Dongarra. Enabling and Scaling Matrix Computations on Heterogeneous Multi-core and multi-GPU Systems. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 365–376, New York, NY, USA, 2012. ACM.

27. Peter E. Strazdins. Lookahead and algorithmic blocking techniques compared for parallel matrix factorization. In *10th International Conference on Parallel and Distributed Computing and Systems, IASTED*, Las Vegas, USA, 1998.

28. Peter E. Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. *Int. J. Parallel Distrib. Systems Networks*, 4(1):26–35, 2001.

29. L. G. Valiant. Bulk-synchronous parallel computers. In M. Reeve, editor, *Parallel Processing and Artificial Intelligence*, pages 15–22. John Wiley & Sons, 1989.

30. V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC'08*, Austin, TX, November 15-21 2008. IEEE Press. http://dx.doi.org/10.1145/1413370.1413402 (DOI:~10.1145/1413370.1413402).

31. Asim YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures.* PhD thesis, University of Tennessee, December 2012.

32. Asim YarKhan, Jakub Kurzak, and Jack Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.

Received June 4, 2014.

DOI: 10.14529/jsfi140106

Exascale Storage Systems – An Analytical Study of Expenses

Julian M. Kunkel[1], *Michael Kuhn*[2], *Thomas Ludwig*[1]

The computational power and storage capability of supercomputers are growing at a different pace, with storage lagging behind; the widening gap necessitates new approaches to keep the investment and running costs for storage systems at bay. In this paper, we aim to unify previous models and compare different approaches for solving these problems. By extrapolating the characteristics of the German Climate Computing Center's previous supercomputers to the future, cost factors are identified and quantified in order to foster adequate research and development. Using models to estimate the execution costs of two prototypical use cases, we are discussing the potential of three concepts: re-computation, data deduplication and data compression.

Keywords: Parallel I/O, Exascale, data center, storage expenses.

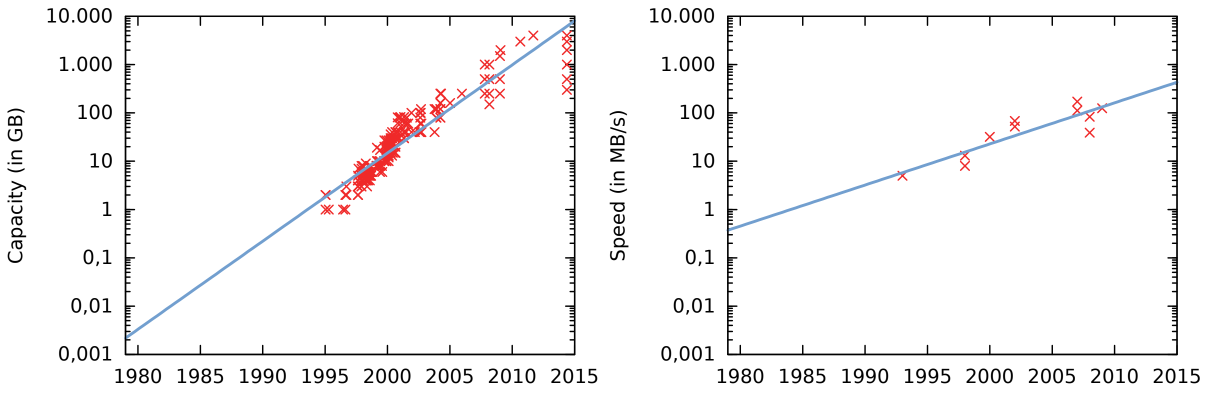
1. Introduction

Supercomputers combine the power of thousands of computers to provide enough computational power to tackle complex scientific problems. They are used to conduct large-scale computations and simulations of complex systems from basically all branches of the natural and technical sciences, such as meteorology, climatology, particle and astro physics. As these simulations have become more accurate and thus realistic over the last years, their demands for computational power have also increased. This, in turn, has also caused the simulation results to grow in size, increasing the demands for huge storage systems. High performance I/O is an important aspect, because storing and retrieving such large amounts of data can greatly affect the overall performance of these applications.

Processor speed [26] and disk capacity [11, 30] have roughly increased by factors of 500 and 100 every 10 years, respectively. The speed of disks, however, grows more slowly; this can be seen in Figures 1a and 1b, which show the increase in disk capacity and speed from roughly the same period of time. Early disks in 1989 delivered about 0.5 MB/s, while current disks manage around 200 MB/s [29]. This corresponds to a 400-fold increase of throughput over the last 25 years. Even newer technologies such as solid-state drives (SSDs) only offer throughputs of around 600 MB/s, resulting in a total speedup of 1,200. In comparison, over the same period of time, the computational power increased by a factor of 1,000,000 for supercomputers due to increasing investments. We are approaching ExaFLOPS performance by the end of the decade. While this problem cannot be solved without major breakthroughs in hardware technology, it is necessary to use the storage hardware as efficiently as possible to alleviate its effects. Moreover, the growth of disk capacity has recently also started to slow down. While the same is true for processor clock rate, this particular problem is being compensated for by growing numbers of increasingly cheap processor cores. Additional investment is required to keep up with the advancing processing power. The outcome of this is that it is not possible to increase storage speed and capacity by the same factor as processing power when keeping investment constant.

Due to the increasing electricity footprints, energy used for storage represents an important portion of the Total Cost of Ownership (TCO) [7]. For instance, the German Climate Computing Center's (DKRZ) storage system is using 7,200 disks to provide a 5.6 PB file system for earth system science. Assuming a power consumption of 20 W for a disk, this results in energy costs of 144,000 € per year for the disks alone.

[1]Deutsches Klimarechenzentrum (DKRZ), Hamburg, Germany
[2]University of Hamburg, Hamburg, Germany



a) Capacity over the years b) Speed over the years

Figure 1. Development of disk drives' characteristics

It is obvious that the increasing demand for storage, combined with the growing gap between processor and storage development, makes it necessary to take action to be able to build balanced supercomputers in the future. The contributions of this paper in this regard are: 1) The prediction of future supercomputers' characteristics for DKRZ. 2) A simplified cost model for running an application, focused on costs for storage and long-term data archival. 3) The quantification of several strategies to reduce data volume. Here, we continue our research on Total Cost of Ownership (TCO) of HPC centers [27, 28].

The rest of this paper is structured as follows: First, the characteristics of future systems at DKRZ are discussed and predicted in Section 2. Then in Section 3, scientific I/O is characterized and two typical simulation experiments are introduced. Additionally, a few observations from the system perspective are provided. The basic model for estimating costs of a job is introduced in Section 4. This model is then applied to typical runs estimating their costs. At last, a more sophisticated cost analysis for long-term data storage is conducted. In Section 5, the concepts of re-computation, deduplication, compression and user-education are introduced and their benefit for future systems is quantified and discussed. Finally, in Section 6 we summarize our findings and identify future work.

2. Characteristics of Future Systems

Table 1 shows the characteristics of the previous (2004) and current (2009) DKRZ systems as well as strawmans for future systems (2015, 2020 and 2025)[3]. An estimate for the investment costs and power consumption for DKRZ is listed in Table 2. One assumption of the extrapolation is that costs stay on the current level. The column "Exascale System" lists the range of characteristics of an Exascale class system as projected by experts [3, 5, 8–10, 17]; note that these predictions vary strongly over time. We approximate the performance characteristics for the next generations according to an extrapolation of trends and hardware roadmaps[4]. A brief explanation of the improvement factors assumed and rationales for them are given in the following:

- For the processor performance, a 20x improvement with every new generation is anticipated, leading to Exascale performance in 2025. This matches the trend that the DKRZ system is about 5 years behind the fastest supercomputer.

[3]The typical procurement cycle is 5 years.

[4]The strawmans cover only one scenario for technological advancement; there are many approaches to predict these characteristics.

- The increase in compute performance can be facilitated by increasing the sheer performance per core, node or the number of nodes. Between between 2004 and 2015, the number of nodes increased by 10x while the number of processor cores per node only increased slightly. We assume, in 2020 and 2025 the intra-node concurrency in densely packed nodes will advance significantly, therefore, the required number of nodes will advance moderately (by 5x and 2.5x, respectively). A transition to accelerators will increase intra-node concur rency significantly, for example. The achieved performance per node of the 2020 system is similar to the performance of the aggressively designed strawman architecture in [17] (4.5 TF/s); with multiple GPUs per node, a performance level which is achievable today.
- Storage throughput experienced a big jump of 13x in the 2015 system which is too optimistic for the future. We expect it to flatten out to 6x for the next few generations of systems: Historically, the performance of disks grew by about 27 % per year or 3.34x in 5 years (see Figure 1b). Current bottlenecks degrade raw performance to a mere 48 MB/s per disk; improvements in hardware and software need to exploit available performance better. We expect a 50 % gain of overall storage performance by improving scheduling with techniques such as burst-buffers, an additional 20 % will be gained by the extra disks.
- Storage capacity may improve by 6x with every new generation. This is mainly motivated by the advances in disk technology. Based on the trend, capacity will increase by 54.4 % annually (see Figure 1a); according to Gartner [22] the annual growth rate of high-density enterprise disks will even be 66.7 % until 2018. If the trend in areal density continues, disks with more than 700 TB would arise in 2025. To prove the storage capacity, the 2025 system requires 133 TB disks, which – by this extrapolation – will be available. Nevertheless, the number of deployed disks will increase by 20 % every generation; to keep the current power budget (as assumed), optimizations in energy efficiency must reduce power consumption. One recent technological advancement which bridges the gap to the next generation system are helium-filled hard disks. However, due to the slower increase in performance, the rebuild time of such a disk would be extensive (almost 3 days).
- Currently, 7 StorageTek silos provide 67,000 slots and 80 tape drives – even if no new libraries are bought, the archive capacity can be increased significantly. The technology offers a raw capacity of 800 GB, 1,500 GB and 2,500 GB for LTO-Ultrium 4, 5 and 6, respectively. Based on the LTO roadmap, the characteristics of a future archive can be approximated well; a doubling of capacity every 2.5 years is expected. Note that this assumption lags behind the historical doubling of disk storage capacity every 15 months, but matches the past development of the LTO standard. Between LTO-4 and LTO-8, throughput is predicted to increase from 120 MB/s to 472 MB/s. With LTO-6, the ALDC compression scheme is switched to LTO-DC which is predicted to increase the average compression ratio from 2 to 2.5.
- The available budget increased only slightly[5], therefore, a similar level of investment and power consumption must be kept. One interesting observation validating the potential characteristics of the 2020 system: if an Exascale system will be deployed in 2020 with a power consumption of 20 MW, the 2020 system with 60 PF/s will have an power consumption of 1.2 MW; which is a bit lower than the value indicated in the table.

This vague prediction shows interesting hurdles which the DKRZ has to overcome: 1) Without intervention, the ratio between storage capacity and archive capacity will shrink. Historically,

[5]The budget excludes the costs for the archive, tape drives and HSM software which is currently about 5M$.

	2004	2009	2015	2020	2025	Exascale (2020)
Performance	1.5 TF/s	150 TF/s	3 PF/s	60 PF/s	1.2 EF/s	1 EF/s
Nodes	24	264	2500	12,500	31,250	100k-1M
Node performance	62.5 GF/s	0.6 TF/s	1.2 TF/s	4.8 TF/s	38.4 TF/s	1-15 TF/s
System memory	1.5 TB	20 TB	170 TB	1.5 PB	12.8 PB	3.6-300 PB
Storage capacity	100 TB	5.6 PB	45 PB	270 PB	1.6 EB	0.15-18 EB
Storage throughput	5 GB/s	30 GB/s	400 GB/s	2.5 TB/s	15 TB/s	20-300 TB/s
Disk drives	4000	7200	8500	10000	12000	100k-1000k
Archive capacity	6 PB	53 PB	335 PB	1.3 EB	5.4 EB	7.2-600 EB
Archive throughput	1 GB/s	9.6 GB/s	21 GB/s	57 GB/s	128 GB/s	-
Power consumption	250 kW	1.6 MW	1.4 MW	1.4 MW	1.4 MW	20-70 MW
Investment	26 M€	30 M€	30 M€	30 M€	30 M€	\$200 M[4]

Table 1. DKRZ System characteristics; future systems are a potential scenario

	Investment	Power consumption
Compute	15.75 M€	1100 kW
Network	5.25 M€	50 kW
Storage	7.5 M€	250 kW
Archive	5 M€	25 kW
Burst-Buffer	2 M€	25 kW

Table 2. Potential investment costs and power consumptions for DKRZ for all future systems

the DKRZ could archive the capacity of the file system per year and keep it for 10 years, which will not be feasible in 2025 any more. Also, the extrapolated throughput of the archive will not suffice to store the data. While it took about 3.6 years – which is enough time for the next procurement – in 2009 to fill the whole archive, the 2015 system will require 12 years already. With the 2025 system and its 128 GB/s of archive throughput, it would take 32 years to fill it completely. However, by increasing the number of tape drives from 80 to the maximum of 448 (5.5x), a large fraction of the required performance can be covered. 2) The speed of computation increases much faster (by 20x) than the storage capacity (5x). Thus, to maintain balance, in the next generations the generated data volume per FLOP must be reduced to one quarter.

With such a system, the often discussed issue of checkpointing is not relevant for DKRZ: Over the time, full system checkpointing increases slightly from 667 s in 2009 to 853 s in 2025. Since most users perform application-specific checkpointing and mostly run small scale configurations, only a fraction of this time is required.

2.1. Burst-Buffer

There have been many thoughts in the HPC community about the potential use of storage class memory in Exascale systems. One scenario is to deploy it as globally accessible cache like in the FastForward I/O effort of Intel, the HDF group and others [5]. Another potential scenario is to utilize node-local or rack-local storage as a new storage tier.

[6]This does not include the RD&E; the US and EU are each investing about \$1 Billion for Exascale.

	2009	2015	2020	2025
Capacity	140 TByte	900 TByte	5.4 PByte	320 PByte
Throughput	83.3 GByte/s	500 GByte/s	3 TByte/s	18 TByte/s
Metadata rate	0.5 M Ops/s	3 M Ops/s	18 M Ops/s	108 M Ops/s
Power consumption	15 kW	15 kW	15 kW	15 kW
Investment	2 M€	2 M€	2 M€	2 M€

Table 3. Potential burst-buffer characteristics; future systems are extrapolated from the current trend

Deploying non-volatile memory at DKRZ could increase the efficiency of the disk-based storage tier which is required to achieve the 6x performance gain as discussed in Section 2. One concept is to modify applications to use a burst-buffer API to explicitly manage a working set of input data prior job execution. Also, temporarily storing data on the faster storage until it is post-processed is an option. Finally, a transparent fast tier for small files and metadata would reduce the seeks on the disks, allowing them to increase their sustained throughput. This approach requires a tight integration into the file system or use of a middleware that takes care of blending the namespaces together. Such a policy could be potentially based on the file extension or on hints such as the selected stripe count in Lustre. Based on the current system usage (see Section 3.3), a 400 GB SSD tier could host all of DKRZ's 50 million files with a size below 8 KiB. Also, with additional 60 TB and 70 TB, files below 1 MiB and 2 MiB could be stored, respectively.

Based on our system's dimension, deploying 1,000 non-volatile memory devices in 25–50 servers seems possible. Inspired by the characteristics of the Intel DC S3700 enterprise SSD[7], potential burst-buffer characteristics are extrapolated and listed in Table 3. With 2 % of the disks' capacity, the burst-buffer would be rather small, yet all small files could be located there easily – even if their number keeps growing according to the current trend. Sustained performance of this system might only be slightly higher than the sustained performance of the disk tier. However, as discussed it could be used to guarantee a better sustained performance of the disks.

3. Scientific I/O

The output of typical scientific applications can be classified by the semantics of the data. There are four kinds: A **data product** is data which is further processed for scientific discovery; **diagnostic data** contains information about the execution of the application – it helps to check correctness of a run; the traditional **checkpoint**; and **out-of-core data**. In this paper, we focus on earth system models; out-of-core computation and other cases are not discussed further, as these are non-typical scenarios for these models.

Weather and climate models periodically store scientific variables – such as temperature and humidity – for each of the simulated domains (cells of a grid). Based on the resolution of the domain decomposition and period of the I/O stepping, the produced data volume can vary drastically. Usually, the data products are kept and archived; sometimes selected checkpoints are also archived because re-starting a 1,000 year simulation from scratch takes considerable time. Diagnostic data can theoretically be deleted after the validity of the run has been checked.

[7]According to the datasheet [16], it offers a capacity of 800 GB, a throughput of 500 MB/s and costs about $1,800.

In the scientists' workflow, the data products are often provided to the community and analyzed with different **post-processing** tools. They are often stored in databases such as the World Data Center for Climate (WDCC) for easier accessibility by other scientists. Recently, big data techniques are discussed to identify interesting phenomena in these large data sets.

In the data model, we classify I/O that is needed to drive a model as **static I/O**; it consists of I/O required for initialization (grid, forcing data), potential data for assimilation and the final output. Its opposite is **periodic output**, in which a simulation stores variables periodically. Note that the period can depend on the variable. Next, two prototypical experiments are discussed.

3.1. Use-Case: CMIP5

As part of the assessment report of the Intergovernmental Panel on Climate Change (IPCC), the Coupled Model Intercomparison Project Phase 5 (CMIP5) [1] compares the predictions of a variety of climate models for a common set of experiments. More than 10.8 million processor hours of the DKRZ's supercomputer were used for the German contribution to CMIP5. In total, 482 runs have been conducted simulating a total of 15,280 years. Depending on the model (atmosphere, ocean, land or chemistry), the output period can fluctuate from six model hours to one model month. Together with checkpoints, a data volume of more than 640 TB has been created. During post-processing, the data products are further refined into 55 TB of data which is then published in scientific databases.

While there are different experiments, we will only discuss a prototypical low-resolution configuration accounting for 12,229 simulated years: A run for a year takes about 1.5 hours on the 2009 system and finishes by creating a checkpoint. The simulation is continued in another job that restarts from the checkpoint; the application-specific checkpointing of the model components occupies 4 GB. Every 10th checkpoint is kept and archived. A month of simulation accounts for 4 GB of data; about half of it is due to storing values in 6 hour periods; for simplicity, we will assume the monthly data volume is included in the 6 hour means. The high number of metadata operations on the file system by the fixed execution is caused by listing the directory with the created artifacts of the job series and more than 100 accessed files. In order to validate data quality and refine post-processing, the data was stored on the file system for almost three years, after which it was archived for 10 years. Relevant statistics are provided in Table 4.

3.2. Use-Case: HD(CP)^2

The High Definition Clouds and Precipitation for Climate Prediction (HD(CP)^2) is a research initiative to improve the understanding of cloud and precipitation processes and their implication for climate prediction [25]. In the project, a high-resolution model is developed. In the current stage of development, a simulation of Germany with a grid resolution of 416 m results in the statistics provided in Table 4. The run on the DKRZ system from 2009 needs 5,260 GB of memory and simulates 2 hours in a wallclock time of 86 minutes. Model results are written every 30 model minutes and a checkpoint is created when the program terminates. Later runs will cover a period of 720 hours and experiment with different parameters. Note that these values reflect the early stage of development and are subject to change; for example, at the moment diagnostics data is written in addition to the actual model output. Due to the well optimized workflow, the output has to be kept on the global file system for only one week.

		CMIP5	HD(CP)^2
General information	Compute nodes	4	64
	I/O timesteps	1460	4
	Time output is stored	3 years	1 week
	Time output is archived	10 years	10 years
	Checkpoint volume	1 GByte	115 GByte
Static I/O	storage volume	1 GByte	115 GByte
	storage volume accessed	2.5 GByte	230 GByte
	storage metadata operations	1000	5
	archived volume	0.1 GByte	115 GByte
	compute time	350s	50s
Periodic I/O	storage volume	32.8 MByte	72 GByte
	storage volume accessed	32.8 MByte	72 GByte
	storage metadata operations	2	1
	archived volume	32.8 MByte	72 GByte
	compute time	3.16s	1280s

Table 4. Job information for a prototypical run of a model configuration on the DKRZ 2009 system

3.3. System's Perspective

From a data center's perspective, it is not yet possible to track the scientific workflow directly. An analysis of the system's status quo can still provide interesting insights regarding the usage of the provided machinery. With the help of monitoring tools, the utilization can be understood better; this information can be used to support decision making for future procurements. For example, on our current system, the sustained throughput of the file system is about 15 GB/s; additionally, the deployed data compression scheme in the tape library achieves a space saving of about 30 % for our workload. Moreover, it may help directing optimization efforts towards the most valuable system aspects and illustrates the demand for user education. A few insights on user behavior are provided in the following:

To understand the demand of scientists better, the file size distribution and file types are analyzed. The following analysis covers the home and project folders with 3.8 PB of data. Overall, 155,000,000 files are stored; their size is distributed as follows: 28 % $\leq$ 8 KiB, 76 % $\leq$ 1 MiB, 90 % $\leq$ 2 MiB, 99.8 % $\leq$ 1 GiB and a few thousand files $\geq$ 8 GiB. Small files can be explained partly by application code and compilation artifacts; version control with Subversion also increases the amount of tiny files significantly.

Analyzing the file type helps to direct optimization efforts in the high-level I/O libraries, but is non-trivial. An initial scan based on the file extension lead to the following observations: 21,000,000 NetCDF files (17 % of files and 34 % of capacity), 9,000,000 GRIB files, 200,000 HDF5 files and TARs (0.5 % of files and 12 % capacity). Unfortunately, the insight provided by the trivial analysis of the file extension is skewed and can only be considered to approximate real behavior. The detailed inspection of the results revealed the habit of users to name files not necessarily based on the data format. File extensions are often made up based on the job ID, or just named "data" or "meta". Since tools such as `file` and `cdo` can not determine the type of files properly in all cases, a better tool is needed. It turned out that 20 % of the data examined has not been accessed within the last year; additionally, millions of empty directories exist.

These few examples highlight the importance of systematic monitoring of inefficient use, which must be done on billions of files in Exascale systems.

4. Modelling Storage Costs

Various cost models of a system have emerged in literature, each developed for a certain purpose. Hardy et al. designed a tool covering many design aspects for a data center [13]. There are also models describing the costs for executing a parallel application but only a few cover storage in more detail. It is also interesting to analyze the costs of storing data forever, which is possible to quantify under the assumption of unlimited increase in areal density; according to Goldstein in 2011, the cost of conserving 1 TB of data was quantified to be about \$5000 [12]. Of course, the price decreases over time – thus, the later data is stored, the cheaper it is – but the more we want to store. We have previously modeled energy consumption for storing scientific data [18]. The model introduced in Equations (4.1) and (4.5) is a simplified model – approximating the costs for running an application and storing data. With this model, we also unify the previously developed models for the purpose of analyzing several Exascale I/O scenarios.

$$costs(job) = costs_{compute}(job) + costs_{storage}(job) + costs_{archive}(job) \tag{4.1}$$

$$rCosts(component) = \frac{investment_{component}}{system_{lifeTime}} + P(component) \cdot energyCosts \tag{4.2}$$

$$costs_{compute}(job) = rCosts(compute) \cdot \frac{job_{nodes}}{system_{nodes}} \cdot job_{compute\_time} \tag{4.3}$$

$$\begin{aligned} costs_{storage}(job) = {} & rCosts(storage) \cdot \left(\frac{job_{volume}}{storage_{capacity}} \cdot job_{time\_stored} + \frac{job_{volume\_accessed}}{storage_{throughput}} \right. \\ & \left. + \frac{job_{metadata\_accesses}}{storage_{metadata\_rate}} \right) \end{aligned} \tag{4.4}$$

$$costs_{archive}(job) = \left(\frac{rCosts(archive)}{archive_{slot\_count}} job_{archival\_time} + costs_{media} \right) \frac{job_{archival\_volume}}{media_{capacity}} \tag{4.5}$$

$$\begin{aligned} costs_{checkpoint}(job) = {} & \frac{checkpoint_{volume}}{storage_{throughput}} \cdot rCosts(storage) + rCosts(compute) \cdot \frac{job_{nodes}}{system_{nodes}} \cdot \\ & \frac{checkpoint_{volume}}{\min\left(job_{nodes} \cdot storage_{throughput\_per\_node}, storage_{throughput}\right)} \end{aligned} \tag{4.6}$$

In the model, the variable *rCosts* represents the running cost for completely utilizing one of the system compartments such as storage. It consists of the compartment's costs and the pure operating expenses (P) due to power consumption (and cooling). Costs for the components are accounted for based on the fraction of a component utilized by a job. For computation, the fraction of nodes needed for the job are chosen (see Equation (4.4)). Under the assumption that the run-time and efficiency of the job is kept constant, the required number of nodes for future systems is computed based on the node performance. The costs for the storage system (see Equation (4.4)) accumulate the costs for the fraction of the throughput, metadata and occupied space over time. This considers the performance of the shared file system and its capacity. Note that stored data may be accessed multiple times by a job, which is covered by additional variables. Since the three cost factors add up, workloads which store large data volumes and access data many times are budgeted higher than the investment and energy costs suggest. We still consider this summation to be valid, as the over-provisioning of space and bandwidth is not accounted for and paid by the data center. The three cost factors can also be accounted individually to understand their impact better. In the archive costs, the number of tape media required to store the data is budgeted (see Equation (4.5)). The costs for checkpoints are given in Equation (4.6); we assume the I/O must be done synchronously and the computation of the application will not continue during checkpointing. The second summand covers the additional costs of the idle compute system; the idle time depends on the time needed to persist the checkpoint.

The model implies several basic assumptions and restrictions: Maintenance by the vendor is usually included in the acquisition costs and, therefore, can be taken into account. Acquisition costs of the data center building are not covered but this investment could be distributed among the component costs. Staff expenses for maintaining the data center are not covered but could be treated in a similar fashion as the facility. The costs for network infrastructure and its utilization are not covered. I/O is not interfering with the compute performance and is completely hidden by using asynchronous techniques; costs for synchronous I/O could be computed using Equation (4.6). While the costs for tape drives could be included in the tape archive costs; the migration of data usually conducted between tape generations is not performed. Tape compression is not covered explicitly by the model but could be addressed by increasing the tapes' capacity virtually. The significant effort of application scientists to port and optimize code for a new system is out of scope of this model as it is hard to capture. Still, by carefully modifying the input variables, it can be adjusted to many scenarios easily. Network costs, for example, could be integrated into the computational costs. The model does not cover expenses caused by idling compute nodes or empty storage space. However, the fraction of costs induced by overprovisioning of hardware could be multiplied to a job's cost. For example, data centers may observe a utilization of 95% of the compute nodes and 85% of storage space.

4.1. Predicted Execution Costs

Based on the cost model, the current and future costs of the CMIP5 and HD(CP)^2 runs can be approximated. The required job information of the models are provided in Table 4; results of the forecast are shown in Table 5. Note that the model assumes the number of nodes scales with the required peak-performance of the future nodes; thus, the number of them required to run HD(CP)^2 degrades from 64 to 32 nodes in 2015, eight nodes in 2020 and one node in 2025. Storage costs from Equation (4.4) are split into three parts: The supply costs (while data occupies capacity but is not used), costs for accessing data and meta data. It can be observed that the storage supply costs are a large fraction of the overall costs for the CMIP5 experiments – keeping the data accessible on disk is the cost driver. The archival costs are in the order of the computation costs. For $HD(CP)^2$, the archival costs are currently about half the execution costs. With future systems, the fraction of storage and archival costs increases and dominates the overall costs by far. In both experiments, the costs for synchronous checkpointing is negligible; for $HD(CP)^2$, it accounts for less than 0.2 % of the computation costs. These predictions clearly indicate the need to reduce the costs for supplying storage space but still retaining science quality.

4.2. Long-term Data Archival

So far the model assumes data is stored on the same storage for the whole life-time of the data. However, data stored on tape archives is usually migrated to newer generations with increased storage capacity; the trend indicates an annual increase of disk capacity by 54.4 % and performance by 27.3 % at almost constant costs. Under the assumption that data is migrated every

		CMIP5				HD(CP)^2			
system		2009	2015	2020	2025	2009	2015	2020	2025
compute		10.50	0.55	0.03	0.001	165.07	8.72	0.44	0.02
storage	supply costs	45.02	5.60	0.93	0.16	2.37	0.30	0.05	0.01
	access costs	0.09	0.01	0	0	0.94	0.07	0.01	0
	metadata costs	0.04	0	0	0	0	0	0	0
checkpoint		0	0	0	0	0.33	0.02	0	0
archival		10.35	1.66	0.41	0.10	86.91	13.91	3.48	0.87
sum		66.01	7.82	1.38	0.26	255.29	22.99	3.97	0.90

Table 5. Costs for one simulation run (in €)

$$\begin{aligned} costs &= \sum_{y=0}^{\infty} \frac{annualCosts \cdot size}{capacity_0 \cdot 1.54^y} = \frac{annualCosts \cdot size}{capacity_0} \cdot \sum_{y=0}^{\infty} \left(\frac{1}{1.54}\right)^y \\ &= \frac{annualCosts \cdot size}{capacity_0} \cdot \frac{1}{1-\frac{1}{1.54}} = \frac{annualCosts \cdot size}{capacity_0} \cdot 2.84 \end{aligned} \tag{4.7}$$

year, the aggregated costs for storing data forever can be computed according to Equation (4.7).

In the following, we will approximate the costs for a particular setting: we assume 400 € for a 6 TB enterprise disk with a transfer rate of 216 MiB/s. Additionally, for the server infrastructure and regular server and infrastructure updates every five years, 100 € are added to the disk's expenses. Personnel and facility costs could be included in the disk's annual cost, too; for DKRZ, the complete annual budget for personnel of 3,000,000 € would add 350 € to the costs. The consumed power of a single disk including cooling is about 20 W, accumulating to 17.52 €; thus, they can be ignored. One draw-back of this model is the missing redundancy; however, adding 10+2 parity would increase the overall costs by an additional 20 %. For simplicity, we expect annual costs of 1,000 €; thus, storing 6 TB of data today – forever – could be liberally accounted for by 2,840 €. This is still competitive: keeping 6 TB for one year on Amazon's Simple Storage Service (S3) already adds up to $1980.[8]

This model has one flaw: the copy speed of future storage systems will not keep up if the trends continue; Equation (4.8) computes the required time for copying a complete disk. In 2050, a future disk would store 37 EB and operate at 1.3 TB/s; it would require 335 days to copy the whole date to the next disk; the year after it, this would not even be possible within a year.

$$copyTime(y) = \frac{capacity_0 \cdot 1.54^y}{speed_0 \cdot 1.273^y} \tag{4.8}$$

Another consideration are the physical limitations to achievable capacity: The platter of a 3.5 inch disk with an inner diameter of 1 in has an area of roughly 5,700 mm^2. One mm^2 of iron crystal consists of $1.72 \cdot 10^{13}$ atoms. If groups of 12 iron atoms are needed to carry one bit of information [2], this would result in a maximum capacity of 1 PB, which would be achieved by 2026.

An interesting question is whether the annual replacement frequency should be increased; this is elaborated for the tape archive. However, the costs for tape archival are slightly different, as the tape drives are quite expensive (about 14,000 € per drive, including maintenance). Therefore, the model is extended to cover costs for power and 80 tape drives. An additional annual fixed

[8]This price is probably changing with the technological advancement, too.

Replacement frequency	Annual costs per slot	Archival for 10 years	Archival for 20 years
1	32.97	204.45	213.99
2	23.61	152.26	157.57
2.5	21.94	145.01	149.40
4	19.43	142.35	145.60
5	18.60	148.28	150.49
10	16.92	201.66	201.80
20	16.09	–	343.10

Table 6. Cost prediction for long-term archival of 2.5 TB of data in the DKRZ tape archive

	Processor	**Memory**	**Network**	**Storage**
Re-computation of results	-	-	-	+
Deduplication	-	- -	0	+
Compression (client side)	-	0	+	++
Compression (server side)	-	0	0	++
User education	+	+	+	+

Table 7. Benefits and penalties of different concepts for data reduction (+ benefits; - penalties)

costs of 1,000,000 € is added to support the libraries' basic hardware and software. The LTO's annual capacity increase of 32 % is a bit lower than the one of disks. Costs for different update frequencies are shown in Table 6. It can be seen that an update interval between 2.5–5 years is much cheaper than other intervals. Note that storing data for ten years is just slightly cheaper than keeping it for 20 years, which is already as expensive as storing the data forever. This consideration shows that the previous storage model for the archive is a bit conservative; it budgets 30 % more than if data is migrated to newer generations every four years.

5. Concepts to Dam the Data Deluge

Several concepts to reduce the amount of stored data will be discussed in this section: re-computation of results, deduplication of identical data and file-system-based as well as application-specific compression. Additional non-technical approaches such as educating users to improve storage usage will be mentioned briefly.

Table 7 gives an overview of the advantages and disadvantages of all concepts. As can be seen, re-computation has positive effects on storage use because not all results are stored; however, this has negative effects on processor, memory and network if the results have to be re-computed frequently. Deduplication can save storage space by not storing duplicate data; additional processor and memory performance are required to actually check for duplicate data. Compression promises huge storage space savings. If the compression is performed on the compute nodes instead of on the storage servers, memory and network utilization can also be reduced; however, doing this can have a negative impact on the actual computation because of the processor overhead caused by compression. In addition to these concepts that have to be implemented and evaluated by the data center's operators, user education has the potential to improve the utilization of all components by making use of more efficient code, data structures, communication schemes and file formats. As has been shown in Section 3.3, it is necessary to monitor and analyze the system to be able to mitigate these inefficiencies.

5.1. Re-computation

Instead of storing all produced data, there are several approaches to analyze data in-situ. A drawback of in-situ analysis is the fact that it requires a careful definition of the analyses to conduct before the actual application is executed since post-mortem data analysis is impossible. Consequently, whenever a new analysis is to be performed, the computation has to be restarted. Even though this approach sounds counterintuitive, re-computation can be attractive if the costs for keeping data on the storage are substantially higher than the costs for re-computation.

Table 5 shows the costs for computation, storage and archival of simulations in the frame of the CMIP5 and HD(CP)^2 projects. In both cases, the cost of computation is higher than the cost for archiving the data in 2009. As mentioned previously, however, computational power continues to improve faster than storage technology; due to the storage systems falling further behind – effectively making it more expensive to store data – the picture changes from 2015 on:

- **2015**: Archiving an HD(CP)^2 simulation result for one year costs 13.91 € while the computation only costs 9.51 €. For CMIP5, the difference is even greater as archival costs 1.66 € while computation costs 0.60 €. Consequently, if the data is only accessed once (HD(CP)^2) or twice (CMIP5) it is cheaper to not archive the data but to re-compute it on demand.
- **2020**: For HD(CP)^2 it is possible to re-compute the data more than seven times; for CMIP5 archival becomes more cost-efficient when the data is accessed more than 13 times.
- **2025**: As the gap between computation and storage increases, re-computation stays feasible until the data has to be accessed more than 43 (HD(CP)^2) or 100 (CMIP5) times.

This analysis only takes re-computations on the same supercomputer into account. If data is supposed to be archived for even longer periods of time, however, it might be necessary to re-compute the data on a future supercomputer. This can present more challenges than simply keeping the application's source code. To exactly reproduce the application's behavior and, consequently, data output, it is necessary to preserve the complete operating environment instead. This includes but is not limited to: the application, all used libraries (such as MPI, NetCDF, HDF, etc.) and even the operating system. There are basically two ways to achieve this:

1. **Binary preservation**: Preserving the compiled binary code would effectively make it impossible to execute the application on differing future architectures (such as different processor architectures) without resorting to emulating the old environment. Emulation usually has significant performance impacts, making this approach infeasible for this use case. However, performing re-computation on the same supercomputer appears feasible using this approach.
2. **Source preservation**: Preserving the source code of all involved components makes sure that all components can be compiled even on different hardware architectures that might be available in the future. Consequently, even the exact compiler version would have to be preserved. However, even minute details such as changing behavior of different processors could lead to differing results.

Discussion Overall, re-computation is only possible in the long term if the supercomputer's architecture remains stable to at least some degree. In any case, it requires long-term planning in advance, both for the hardware and software components. Additionally, it has to be considered that it might be impossible to reproduce a bit-identical version of the data a few years after the initial application run. Virtual machine images or containers may be a solution to provide the appropriate environment, easing transfer of scientific applications at the same time.

	2009	2015	2020	2025
Storage capacity	5.6+**1.68** PB	45+**13.5** PB	270+**81** PB	1.6+**0.48** EB
System memory	20+**33.6** TB	170+**270** TB	1.5+**1.62** PB	12.8+**9.6** PB
Power consumption	1.6+**0.24** MW	1.4+**0.20** MW	1.4+**0.14** MW	1.4+**0.09** MW
Investment	30+**2.52** M€	30+**2.38** M€	30+**1.62** M€	30+**1.13** M€

Table 8. Impact of deduplication for selected DKRZ systems (changes due to deduplication are marked in **bold**)

5.2. Deduplication

Deduplication works on the principle that data is split up into (possibly variably-sized) blocks and that each unique block of data is stored only once [23]. Instead of storing duplicate data, a reference to the original block is created for each repeated occurrence. Our previously conducted study for HPC data already showed great potential for data savings, allowing 20–30 % of redundant data to be eliminated on average [21]. To determine the potential savings, we independently scanned 12 sets of directories with a total amount of data of more than 1 PB. Deduplication was only applied within each set and not across the complete amount of data. We used a tool computing the hashes of data to determine the deduplication potential; instead of the traditional static chunking method, content-defined chunking has been used. As mentioned previously, 20–30 % of data could be deduplicated on average; using full-file deduplication, it was still possible to reach savings of 5–10 %. Among the most interesting findings were the facts that between 3–9 % of the data consisted of zeros and that the top 5 % of chunks accounted for 35 % of the whole data set.

Even though deduplication offers significant space savings, it also has its downsides: It can be very expensive in terms of memory overhead. For every 1 TB of deduplicated data, approximately 5–20 GB of storage are required to store the matching deduplication tables. The size of these tables heavily depends on the chosen block size. Enterprise deduplication solutions usually use block sizes between 4 KB and 16 KB to achieve high deduplication rates; we have settled on a block size of 8 KB for our following analysis. Using the SHA256 hash function (256 bits = 32 bytes) and 8 KB file system blocks (using 8 byte offsets) results in an overhead of at least 5 GB per 1 TB as shown in Equation (5.1). Additional data structure overhead will likely increase this number even further; consequently, we will assume an additional overhead of 8 bytes per hash, pushing the memory requirements to 6 GB of main memory per TB of storage. While it would be possible to increase the block size to reduce the overhead caused by the deduplication tables, this would also decrease the deduplication's effectiveness.

The deduplication tables store references between the hashes and the actual data blocks within the file system. To enable efficient online deduplication, these tables have to be kept in main memory because potential duplicates have to be looked up for each write operation.

$$\begin{gathered} 1\,\text{TB} \div 8\,\text{KB} = 125,000,000 \\ 125,000,000 \cdot (32\,\text{B} + 8\,\text{B}) = 5\,\text{GB} \quad (0.5\,\%) \end{gathered} \tag{5.1}$$

Table 8 shows an overview of the DKRZ's supercomputers from 2009 to 2025 and the influence that large-scale deduplication of the complete storage system would have. The only important characteristics are system memory, storage capacity, power consumption and investment; their base values have been taken from Table 1. The archive can be ignored because deduplication is not easily possible for tape systems as it requires random access for efficient operation. The amount of additional system memory necessary has been calculated using the previously

	2009	2015	2020	2025
Storage capacity	4.3+**1.3** PB	34.6+**10.4** PB	207.7+**62.3** PB	1.2+**0.4** EB
System memory	20+**25.8** TB	170+**207.7** TB	1.5+**1.2** PB	12.8+**7.4** PB
Power consumption	1.54+**0.19** MW	1.34+**0.15** MW	1.34+**0.1** MW	1.34+**0.07** MW
Investment	28.27+**1.94** M€	28.27+**1.83** M€	28.27+**1.25** M€	28.27+**0.87** M€

Table 9. Deduplication impact if storage capacity is preserved on selected DKRZ systems (changes due to deduplication are marked in **bold**)

established value of 6 GB main memory per 1 TB of storage space. According to our previous study [21], we have used an optimistic estimation of 30 % for the increase in storage capacity due to deduplication. Based on [17, page 165], the power consumption overhead for the additional system memory has been calculated using an estimated 9 % of the overall power distribution. The investment costs for the system memory have been estimated using data from [3, page 49] for main memory prices in 2020. Note that additional processor time and energy costs for computing the hash and the additional randomization of data accesses are not covered in this analysis.

For data center operators and vendors, it can also be interesting to quantify the potential benefits of a deduplication system. Therefore, Table 9 shows the characteristics of selected systems where the goal is to preserve the overall storage capacity. As can be seen from this analysis, the investment costs and power consumption would be larger for the 2009 and 2015 systems. Therefore, with only 30 % deduplication rate the average 8 KB blocks would imply too much overhead and costs. However, investment costs for the 2020 and 2025 systems are slightly lower and so would be the power consumption of the 2025 system.

Discussion Overall, deduplication can be used to significantly increase storage capacity by removing duplicate data blocks. However, the presented configuration using 8 KB blocks has an overall negative impact on the cost of the storage system. There are several ways to remedy this:

- Using larger block sizes can significantly reduce the memory overhead caused by the deduplication tables. When using 8 KB blocks, 0.6 % of the storage capacity have to be added as additional main memory; increasing the block size to 16 KB or 32 KB can lower this to 0.3 % or 0.15 %, respectively. However, the impact of larger block sizes on the deduplication ratio has to be considered and analyzed further.
- A cheaper alternative would be to use full-file deduplication only. This approach would not save any actual I/O bandwidth because the file has to be written completely before its hash can be determined. Additionally, the deduplication rate would be lower.
- Deduplication could be performed offline. Using modern copy-on-write-capable file systems that already hash all data blocks would allow integrating a deduplication tool that uses the existing information to find duplicate blocks. This would be especially useful for full-file deduplication; however, it could also be used to perform normal block-based deduplication because offline deduplication is not as performance critical as online deduplication, potentially relaxing the requirement that hash tables have to be kept in main memory.

Especially the last point could be an interesting starting point for future work because the tight integration of advanced file system features and offline deduplication could help to alleviate some of the problems present with the current approach.

	`none`	`zle`	`lzjb`	`lz4`	`gzip-1`	`gzip-9`	`lzma`	`mafisc`
Compression Ratio	1.00	1.13	1.57	1.52	2.04	2.08	2.60*	2.81*
Processor Utilization (%)	23.7	23.8	24.8	22.8	56.6	83.1	n/a	n/a
Runtime Ratio	1.00	1.04	1.09	1.09	1.06	13.66	n/a	n/a

Table 10. Comparison of different compression algorithms (`lzma` and `mafisc`'s ratios are estimated)

5.3. Compression

Similar to deduplication, compression can be used to reduce the amount of data that has to be stored on the storage system. While deduplication can only be employed meaningfully on the storage servers, compression can be used on both the clients and the servers.

On the one hand, processor overhead caused by client-side compression might negatively influence applications; however, data can be compressed before being sent to the servers, possibly reducing the required network bandwidth. On the other hand, server-side compression can be completely transparent to the clients. Independently of the location, compression uses additional processor time, which might require more powerful processors; this, in turn, can increase the overall power consumption.

In our previous HPC compression study, we leveraged Lustre 2.5 with its ZFS back-end file system to achieve transparent file system level compression [6]. ZFS is a local file system that offers a rich set of advanced features. Among others, it provides advanced storage device management, data integrity, as well as transparent compression and deduplication. It supports several compression algorithms: zero-length encoding `zle`, `lzjb` (a modified version of `LZRW1` [31]), `lz4` [32] and `gzip` (a variation of LZ77 [33]). We evaluated all of them using a scientific data set containing data from the Max Planck Institute Ocean Model (MPI-OM). The data set consists of around 73 % binary data and 27 % NetCDF data [24]. We have not considered decompression yet, because the evaluated compression algorithms are expected to have higher overheads when compressing than when decompressing.

Table 10 shows a comparison of the most interesting compression algorithms supported by ZFS; the data set was copied from an uncompressed ZFS pool into another separate compressed ZFS pool and processor utilization as well as runtime were recorded. All compression algorithms increase the runtime ratio only slightly; however, while `zle`, `lzjb` and `lz4` only use negligible amounts of processor time, `gzip` adds significant overhead. `gzip` additionally allows the compression level (1–9) to be specified; higher compression levels do not significantly improve the compression ratio but increase the runtime and processor utilization considerably. Consequently, we chose `lz4` and `gzip-1` for further analysis.

Additionally, Table 10 also contains information about the `lzma` and `mafisc` algorithms applied to the CMIP5 data set. Since this data is not as compressible as the MPI-OM data set, our previous results of 2.1 and 2.25 are scaled by an estimated correction factor (based on `gzip`'s compression ratio when applied to the CMIP5 data set). `mafisc` applies a range of different filters and transformations to the data and then compresses it using the lzma algorithm [14]; since this information is only available on the clients, it is necessary to use `mafisc` on the compute nodes.

Table 11 shows the results for runs of the IOR benchmark, which we adapted to simulate realistic write activities on the Lustre storage servers. Interestingly, our experiments have shown that application performance is not reduced by the compression on the servers; in the tested cases it was actually increased because the storage servers were able to deliver more throughput. Overall,

Comp. Algorithm	Runtime Ratio	Power Ratio	Energy Ratio
`none`	1.0	1.0	1.0
`lz4`	0.92	1.01	0.93
`gzip-1`	0.92	1.10	1.01

Table 11. Power and energy consumption of parallel I/O benchmark using selected compression algorithms

	2009	2015	2020	2025
Storage capacity	5.6+**2.8** PB	45+**22.5** PB	270+**135** PB	1.6+**0.8** EB
Power consumption	1.6+**0.025** MW	1.4+**0.025** MW	1.4+**0.025** MW	1.4+**0.025** MW

Table 12. Impact of server-side `lz4` compression for selected DKRZ systems (changes are marked in **bold**)

the energy consumption was decreased for `lz4` because of the lower runtime combined with the negligible increase in power consumption. For `gzip-1`, the total energy consumption increased by 1 % due to the higher power consumption caused by the elevated processor utilization.

An overview of the DKRZ's supercomputers from 2009 to 2025 and the influence that large-scale compression of the complete storage system would have is shown in Table 12. The only important characteristics are storage capacity and power consumption; their base values have been taken from Table 1. We ignore the archive because compression is already enabled by default for the tape systems. The amount of additional storage has been calculated using a compression ratio of 1.5 for `lz4` as shown in Table 10. The power consumption overhead has been pessimistically estimated to be at most 10 % of the storage system's overall power consumption; these numbers are based on Table 11, assuming a worst-case runtime ratio of 1.0. Additionally, we assume that it is not necessary to purchase more powerful processors; consequently, there is no negative influence on the investment cost.

Discussion Overall, compression can significantly increase the available storage capacity with only negligible overhead. In contrast to approaches like deduplication, no additional hardware investments are necessary. Coupled with the marginal increase in the storage system's power consumption, the overall effect is very beneficial.

Application-specific compression algorithms can use knowledge about the stored data to improve compression ratios. In contrast to the file system level compression, however, the compression has to be performed by the applications themselves; even though the actual process is usually encapsulated in some I/O library such as HDF5, the involved compression overhead can negatively influence application performance. Application-specific knowledge can also be used by lossy compression schemes such as the floating point compressors proposed by Lindstrom and Isenburg [20], ISABELA [19] or GRIB and APAX to achieve larger gains for climate data [15]; this, however, forces the scientist to define boundaries for the tolerated loss in precision [15]. Recently, Baker et al. presented an approach to determine the required precision for climate data; they achieve a compression ratio of 5 without noticeable impact on the scientific result [4].

6. Summary, Conclusions and Future Work

To estimate current and future expenses, firstly, we predicted potential characteristics of the next DKRZ supercomputers, discussing the hardware trend in relation to Exascale studies. While the computational power grows by 20x every generation, the storage capacity increase of 6x lags behind. Applying the developed cost model to two typical experiment runs, the approximate costs for computation, storage and archival are quantified. Moreover, we discussed cost models for long-term archival under the trend of increasing areal densities and performance. It becomes apparent that keeping data available – yet not using it – is dominating the costs and its fraction will increase significantly with the next systems.

We compare concepts to handle the increasing storage costs: For low numbers of accesses, re-computation of data is promising, but requires concepts to guarantee reproducibility of the results. Deduplication may be beneficial but is not really promising in its current form. However, it is a tool to identify inefficient use of storage space which could be mitigated by proper user education. By improving performance and capacity, server-side compression bears the potential to reduce the TCO significantly; it is especially useful for data that is not compressed by the scientists explicitly. An advantage of client-side compression is the chance for achieving better compression ratios by taking the data format into account. Of these concepts, lossy compression achieves the best compression ratio but forces the user to define the required accuracy.

In the future, we plan to elaborate the cost model and use it to make decisions for new applications, as it is a potential source of future cost savings. We also continue our work on tools to analyze the users' workflow in order to identify suboptimal usage scenarios and mitigate their impact. Finally, we will explore the benefits of adaptive compression and interfaces that enable more intelligent compression by providing semantical information. We believe that a proper analysis of all cost factors of an HPC system and the usage characteristics allows for an optimal configuration of the system and thus a maximum cost-benefit ratio.

Acknowledgment: We thank Carsten Beyer, Joachim Biercamp, Hinrich Reichardt, Michael Probst, Panagiotis Adamidis, Stefanie Legutke for providing valuable input and Michaela Zimmer and Konstantinos Chasapis for their reviews.

References

1. CMIP5 – Overview. `http://cmip-pcmdi.llnl.gov/cmip5/`. Last accessed: 2014-06.
2. IBM researchers make 12-atom magnetic memory bit. `http://www.bbc.com/news/technology-16543497`, 2012. Last accessed: 2014-06.
3. Technical Challenges of Exascale Computing. Technical report, The MITRE Corporation, April 2013. `http://institute.lanl.gov/resilience/docs/JSR-12-310-Challenges_of_exascaleFINAL.pdf`.
4. Allison H. Baker, Haiying Xu, John M. Dennis, Michael N. Levy, Doug Nychka, and Sheri A. Mickelson. A Methodology for Evaluating the Impact of Data Compression on Climate Simulation Data. *ACM Symposium on High-Performance Parallel and Distributed Computing*, 2014, to appear.
5. John Bent. Exascale Storage for HPC: "Burst Buffers" with a new storage API. Presentation at the Exascale10 satellite event during ISC. `http://www.exascale10.com/`, 2013.

6. Konstantinos Chasapis, Manuel Dolz, Michael Kuhn, and Thomas Ludwig. Evaluating Power-Performace Benefits of Data Compression in HPC Storage Servers. In Steffen Fries and Petre Dini, editors, *IARIA Conference*, pages 29–34. IARIA XPS Press, 04 2014.

7. Matthew L. Curry, H. Lee Ward, Gary Grider, Jill Gemmill, Jay Harris, and David Martinez. Power Use of Disk Subsystems in Supercomputers. In *Proceedings of the Sixth Workshop on Parallel Data Storage*, PDSW '11, pages 49–54, New York, NY, USA, 2011. ACM.

8. Department of Energy. Exascale Strategy – Report to Congress. Technical report, United States Department of Energy, June 2013. `http://assets.fiercemarkets.net/public/sites/govit/perera_fgit_foia_doe_exascale%20report.pdf`.

9. Jack Dongarra. Impact of Architecture and Technology for Extreme Scale on Software and Algorithm Design. presentation, 2010.

10. Giovanni Erbacci, Vincent Bergeaud Francois Bodin, Alberto Pasanisi, Simon McIntosh-Smith, Thomas Ludwig, Franck Cappello, Carlo Cavazzoni, and Marie-Christine Sawley. European Exascale Software Initiative 2 Deliverable D5.1 – WP5 First Intermediate Report – Cross Cutting Issues Working Groups. `http://www.eesi-project.eu/modules/download_pictures/dlc.php?file=343&id=1389118661&sid=233`, 2013.

11. Richard Freitas, Joseph Slember, Wayne Sawdon, and Lawrence Chiu. GPFS scans 10 billion files in 43 minutes. *IBM Advanced Storage Laborator. IBM Almaden Research Center. San Jose, CA*, 95120, 2011.

12. Serge Goldstein. DataSpace: A Model for Long-Term Preservation and Dissemination of Research Data. University of Massachusetts and New England Area Librarian e-Science Symposium, presentation, 2011.

13. Damien Hardy, Isidoros Sideris, Ali Saidi, and Yiannakis Sazeides. EETCO: A tool to estimate and explore the implications of datacenter design choices on the tco and the environmental impact. In *Workshop on Energy-efficient Computing for a Sustainable World*, 2011.

14. Nathanael Hübbe and Julian Kunkel. Reducing the HPC-Datastorage Footprint with MAFISC – Multidimensional Adaptive Filtering Improved Scientific data Compression. In *Computer Science - Research and Development*, Hamburg, Berlin, Heidelberg, 2012. Executive Committee, Springer.

15. Nathanael Hübbe, Al Wegener, Julian Kunkel, Yi Ling, and Thomas Ludwig. Evaluating Lossy Compression on Climate Data. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, number 7905 in Lecture Notes in Computer Science, pages 343–356, Berlin, Heidelberg, 06 2013. Springer.

16. Intel. Intel Solid State Drive DC S3700 Series – Product Specification, March 2014.

17. Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems, DARPA report. `http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf`, Sep 2008.

18. Julian Kunkel, Olga Mordvinova, Michael Kuhn, and Thomas Ludwig. Collecting Energy Consumption of Scientific Data. *Computer Science - Research and Development*, pages 1–9, 2010.

19. S. Lakshminarasimhan, N. Shah, S. Ethier, S.H. Ku, CS Chang, S. Klasky, R. Latham, R. Ross, and N.F. Samatova. Isabela for effective in situ compression of scientific data. *Concurrency and Computation: Practice and Experience*, 2012.

20. P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):1245–1250, Sept 2006.

21. Dirk Meister, Jürgen Kaiser, Andre Brinkmann, Michael Kuhn, Julian Kunkel, and Toni Cortes. A Study on Data Deduplication in HPC Storage Systems. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)*, 11 2012.

22. Chris Mellor. Spin that disk drive forecast, Gartner: Watch those desktop units dive. `http://www.theregister.co.uk/2014/03/31/gartner_disk_drive_forecast/`, 2014.

23. Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

24. Russ Rew and Glenn Davis. Data Management: NetCDF: an Interface for Scientific Data Access. *IEEE Computer Graphics and Applications*, (10-4):76–82, 1990.

25. The HD(CP)^2 Project. HD(CP)^2. `http://hdcp2.eu/`. Last accessed: 2014-06.

26. The TOP500 Editors. TOP500. `http://www.top500.org/`, 06 2013. Last accessed: 2013-06.

27. Thomas Ludwig. The Costs of HPC-Based Science in the Exascale Era. Invited talk at the Supercomputing Conference SC12, Salt Lake City, 2012.

28. Thomas Ludwig and Albert Reuther and Amy Apon. Cost-Benefit Quantification for HPC: An Inevitable Challenge. Birds of a Feather session at the Supercomputing Conference SC13, Denver, 2013.

29. Wikipedia. Festplattenlaufwerk – Geschwindigkeit. `http://de.wikipedia.org/wiki/Festplattenlaufwerk#Geschwindigkeit`, 02 2013. Last accessed: 2013-02.

30. Wikipedia. Mark Kryder – Kryder's Law. `http://en.wikipedia.org/wiki/Mark_Kryder#Kryder.27s_Law`, 02 2013. Last accessed: 2013-02.

31. R.N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conference, 1991. DCC '91.*, pages 362–371, Apr 1991.

32. Yann Collet. LZ4 Explained. `http://fastcompression.blogspot.com/2011/05/lz4-explained.html`, 2008. Last accessed: 2013-12.

33. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, May 1977.

Received June 8, 2014.