Supercomputing Frontiers and Innovations

2017, Vol. 4, No. 3

Scope

- Enabling technologies for high performance computing
- Future generation supercomputer architectures
- Extreme-scale concepts beyond conventional practices including exascale
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Distributed operating systems, kernels, supervisors, and virtualization for highly scalable computing
- Scalable runtime systems software
- Methods and means of supercomputer system management, administration, and monitoring
- Mass storage systems, protocols, and allocation
- Energy and power minimization for very large deployed computers
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Parallel performance and correctness debugging
- Scientific visualization for massive data and computing both external and in situ
- Education in high performance computing and computational science

Contents

DOI: 10.14529/jsfi170301

Resilience Design Patterns: A Structured Approach to Resilience at Extreme Scale

***Saurabh Hukerikar*[1], *Christian Engelmann*[1]**

Reliability is a serious concern for future extreme-scale high-performance computing (HPC) systems. Projections based on the current generation of HPC systems and technology roadmaps suggest the prevalence of very high fault rates in future systems. While the HPC community has developed various resilience solutions, application-level techniques as well as system-based solutions, the solution space remains fragmented. There are no formal methods and metrics to integrate the various HPC resilience techniques into composite solutions, nor are there methods to holistically evaluate the adequacy and efficacy of such solutions in terms of their protection coverage, and their performance & power efficiency characteristics. Additionally, few of the current approaches are portable to newer architectures and software environments that will be deployed on future systems. In this paper, we develop a structured approach to the design, evaluation and optimization of HPC resilience using the concept of design patterns. A design pattern is a general repeatable solution to a commonly occurring problem. We identify the problems caused by various types of faults, errors and failures in HPC systems and the techniques used to deal with these events. Each well-known solution that addresses a specific HPC resilience challenge is described in the form of a pattern. We develop a complete catalog of such resilience design patterns, which may be used by system architects, system software and tools developers, application programmers, as well as users and operators as essential building blocks when designing and deploying resilience solutions. We also develop a design framework that enhances a designer's understanding the opportunities for integrating multiple patterns across layers of the system stack and the important constraints during implementation of the individual patterns. It is also useful for defining mechanisms and interfaces to coordinate flexible fault management across hardware and software components. The resilience patterns and the design framework also enable exploration and evaluation of design alternatives and support optimization of the cost-benefit trade-offs among performance, protection coverage, and power consumption of resilience solutions. The overall goal of this work is to establish a systematic methodology for the design and evaluation of resilience technologies in extreme-scale HPC systems that keep scientific applications running to a correct solution in a timely and cost-efficient manner despite frequent faults, errors, and failures of various types.

Keywords: high-performance computing, resilience, fault tolerance, design patterns.

Introduction

Extreme-scale, high-performance computing (HPC) will significantly advance discovery in fundamental scientific research by enabling multiscale simulations that range from the very small, on quantum and atomic scales, to the very large, on planetary and cosmological scales. Computing at scales in the hundreds of petaflops, exaflops and beyond will also provide the computing power for rapid design and prototyping and big data analysis. Yet, to build and effectively operate extreme-scale HPC systems, there are several key challenges, including management of power, massive concurrency, and resilience [22].

In the pursuit of greater computational capabilities, the architectures of future HPC systems are expected to change radically. These innovative systems require equally novel components, which are designed to communicate and compute at unprecedented rates. Traditional HPC system design methodologies have not had to account for power constraints, or parallelism on the level designers must contemplate for future extreme-scale systems [55]. The evolution in the architectures will require changes to the programming models and the software environment

[1]Oak Ridge National Laboratory, One Bethel Valley Road, Oak Ridge, TN, USA

to ensure application scalability. In the midst of these rapid changes, the resilience to faults or defects in system components, which can cause errors and failures, will be critical. The reliability of these systems will be threatened by a decrease in individual transistor reliability due to manufacturing defects prevalent at deeply scaled technology nodes, device aging related effects, etc. [9]. The chips built using these devices will be increasingly susceptible to errors due to the reduced noise margins arising from near-threshold voltage (NTV) operation [24] (that will be necessary to meet the limits on power consumption). These effects are expected to increase the rate of transient and hard errors in the system. The scientific applications running on these systems will no longer be able to assume correct behavior of the underlying machine. The errors will propagate and generate various kinds of failures, which may result in outcomes in HPC applications ranging from data corruptions to catastrophic crashes.

Managing the resilience of future extreme-scale systems is a complex, multidimensional challenge. As HPC systems approach exaflops scale, the sheer frequency of faults and errors in these systems will render many of the existing resilience solutions ineffective. Newer modes of failures due to faults and errors, which will only emerge in advanced process technologies and complex system architectures, will require novel resilience solutions. To remain viable the adaptations of existing solutions, as well as the designs of new solutions, must also navigate the complexity of the hardware and software environments of future systems. Additionally, HPC resilience solutions, both hardware and software, must optimize for some combination of performance, power consumption and cost while providing effective protection against faults, errors and failures. Therefore, addressing the resilience challenge for extreme-scale HPC systems will require integration and coordination between various hardware and software technologies that are collectively capable of handling a broad set of fault models at accelerated fault rates.

The HPC community and vendors have developed a number of hardware and software resilience solutions over the years to confront faults and their consequences in a HPC system and to limit their impact on the applications. Most of these solutions are based on a limited set of underlying detection, containment and mitigation techniques that have persisted through generations of systems and will remain important in the future. The key to the design and implementation of HPC resilience solutions is no longer the invention of novel methodologies for dealing with the various fault types that may occur, or to manage the extreme fault rates; rather, it is based on the selection and combination of the most appropriate solutions among the well-understood resilience techniques and adapting them to the design concerns and constraints of the emerging extreme-scale systems. However, there are no systematized methods to adapt the existing solutions to future architectures and software environments, nor are there formalized to integrate multiple solutions into composite solutions. There is also a lack of standardized methods to investigate and evaluate the effectiveness and efficiency of such solutions. Therefore, the designers of HPC hardware and software components have a compelling need for a systematic methodology for designing, assessing and optimizing resilience solutions.

In this work, we develop a structured approach for constructing resilience solutions for HPC systems and their applications based on the concept of design patterns. Design patterns are descriptions of well-known solutions to specific, repeatedly occurring problems that are encountered in a specific domain. In an effort to develop resilience design patterns we identify well-known techniques to handle faults and their consequences in various hardware and software components throughout the HPC system stack. In general, resilience solutions provide techniques for the detection of faults, errors or failures in a system, mechanisms to ensure that their propagation is limited, and for masking of error or failure and recovery of the system. This paper presents a complete catalog of patterns that capture the solutions for each of these three aspects. Each pattern provides a solution to a recurring HPC resilience problem under a set of clearly defined assumptions about the type of the fault, error or failure it deals with and the constraints about the system behavior it guarantees. The resilience design patterns are specified at a high level of abstraction and describe solutions that are free of implementation details. The patterns have the potential to shape the design of HPC applications' algorithms, numerical libraries, system software, and hardware architectures, as well as the interfaces between layers of system abstraction. Therefore, they are intended to be useful for HPC application, library and tool developers, hardware architects and system software designers, as well as system users and operators.

We codify the resilience design patterns in a layered hierarchy, which classifies the patterns in the catalog, and clearly conveys the relationships among them. The hierarchical scheme enables individual hardware/software component designers to focus on problems and constraints related to detection, containment and mitigation/recovery of specific fault types in specific contexts, while system architects contemplate role of the individual patterns within the context of the overall system architecture and software environment and issues related to stitching the various patterns together and refinement of their interactions. Combining these patterns according to the guidelines given by the classification scheme provides a systematic way to design and implement new resilience solutions, port existing solutions to future architectures and software environments, and to holistically evaluate the scope and efficiency of the solutions. Therefore, using the design patterns as building blocks enables:

- Systematic design and refinement of resilience solutions by using patterns to outline the overall structure of the solution (independent of a specific implementation approach), and incrementally converging towards a detailed implementation;
- Design of solutions with a clear understanding of their protection coverage and performance efficiency;
- Evaluation and comparison of alternative resilience solutions through qualitative and quantitative evaluation of the coverage and handling efficiency of each solution;
- Design of flexible solutions through integration of multiple patterns into complete resilience solutions. The individual patterns may be independently evolved and developed for portability to different HPC system architectures and software environments;
- Design of cross-layered resilience solutions that combine capabilities from different layers of the system stack; and
- Optimization of the trade-off space, at design time or at runtime, between the key system design factors: performance, resilience, and power consumption.

In this paper, we also develop a systematic methodology to combine an essential set of patterns into productive and efficient resilience solutions. We present a conceptual framework based on the notion of *design spaces* that enables HPC designers to use the patterns as reusable design elements. The framework enables designers to navigate the complexities of composing patterns into complete solutions within the constraints of performance and power overheads, the fault model and its impact on the system, hardware and software implementation challenges, etc. The overall goal of this work is to enable a systematic methodology for the design and evaluation of resilience technologies in HPC systems that keep applications running to a correct solution in a timely and cost-efficient manner despite frequent faults, errors, and failures of various types.

1. Design Patterns for HPC Resilience

The occurrences of various types of faults, errors and failures are not rare events in modern large-scale HPC system environments. The term **fault** refers to an underlying flaw or defect in a system that has potential to cause problems, an **error** refers to the result of the activation of a fault, which causes an illegal system state. A **failure** occurs if an error reaches the service interface of a system, resulting in system behavior that is inconsistent with the system's specification. The faults are due to radiation-induced effects such as particle strikes from cosmic radiation and the system environment, chip manufacturing defects and design bugs that remain undetected during post-silicon validation and manifest themselves during system operation, as well as circuit wear out, or aging failure mechanisms of CMOS integrated circuits. The faults may also occur due to software bugs, which is a growing concern as the complexity of the software environment grows. Due to the complex system interactions and dependencies between the hardware and software components, the application program, and the HPC system's physical environment, preventing the activation of these faults and containing the propagation of the resulting errors and failures to other components a significant challenge.

HPC resilience solutions seek effective and efficient management of the different types of fault and errors to ensure that the applications produce reliable outcomes despite the resulting degradations and failures. The focus of resilience solutions is on application correctness lieu of, or even at the expense of, reliability of state of the system. In general, every HPC resilience solution consists of the following core capabilities:

- **Detection:** Identifying the presence of an anomaly in the data or control value is an important aspect of any resilience management strategy. The detection and diagnosis of faults in a system may allow the remedy of the underlying defect, which may prevent the activation of an error or failure. The timely detection of errors or failures enables recovery of the system.
- **Containment:** When an error or failure is discovered in a system, containment strategies assist in limiting the impact of the event on other components in the system. Limiting the propagation enables simplified recovery strategies.
- **Recovery:** The recovery aspect of any resilience solution is necessary to ensure that the application outcome is correct in spite of the presence of an error or a failure in a system. The recovery may entail a workaround to isolate and bypass the presence of an error or a failed component, complete elimination of the error or failure, and may also seek to prevent the root cause of the underlying fault from resurfacing.

Often the solutions used to achieve these capabilities are based on well-known techniques, which have been repeatedly used by hardware and software designers to increase system reliability since the early days of computing systems. These techniques are based on the use of redundant structures to mask failed components, error-control codes and duplication or triplication with voting to detect or correct information errors, diagnostic techniques to locate failed components, automatic switchovers to replace failed subsystems, and the specification of well-defined modular structures and interfaces for containment and definition of recovery scope [5]. Many of the resilience solutions, hardware and software, used in HPC environments over the past three decades are also largely based on these set of techniques.

Our goal is to capture the best-known techniques that are used in the design of HPC resilience solutions formatted as design patterns. A design pattern describes a generalizable solution to a recurring problem that occurs within a well-defined context. It identifies the key aspects of a solution and presents it in the form of an abstract description, which provides designers with guidelines on how to solve a problem. Each pattern in this paper presents a solution to a specific problem in detecting, recovering from, or masking a fault, error or failure event. The pattern descriptions don't describe a concrete design or an implementation, and are also free from constraints of details associated with the level of system abstraction at which the solution can be implemented. Therefore, the resilience patterns may be used as design templates that may be adapted by the HPC hardware or software designers for a specific problem at hand. The design of new resilience solutions and adapting existing ones for future extreme-scale systems is accomplished by combining various patterns into complete solutions and by refining their interactions. The patterns describe the design decisions and trade-offs that must be considered when applying a certain solution, which enables designers to reason about the impact of applying a solution on a system's performance scalability and power consumption overhead as well as consider implementation issues. Since the various resilience techniques handle different types of events, and they each provide different guarantees about properties such as the time or the space overhead introduced to the normal execution of the system, number of simultaneous errors or failures it can handle, the efficiency of the reaction to a failure, the design complexity added to the system, the patterns may also be used to explore alternatives solutions to a given problem.

Based on the insight that any resilience solution is only necessary in the presence of, or sometimes in the anticipation of an anomalous event, such as a fault, error, or failure, we define the template of a resilience design pattern in an event-driven paradigm. The design pattern template consists of a *behavior* and a set of *activation* and *response interfaces*. The pattern behavior provides a description of the solution, which systematically names, explains the semantics of, and evaluates the trade-offs involved in using the solution in an HPC environment. The activation and response interfaces specify the conditions for application of the solution. The individual implementations of the same pattern may have different levels of performance, resilience, and power consumption. However, using this universal template enables a standardized approach for the evaluation of patterns and comparison between alternative solutions for a given problem. While any resilience design pattern must conform to this basic template, the instantiation of a pattern may cover combinations of detection, containment and mitigation capabilities.

2. Classification of Resilience Design Patterns

For designers of HPC resilience solutions, the patterns serve as reusable design elements. For the design of resilient hardware and software components, the patterns can be combined in different ways to produce complete solutions. For a systematic approach to transforming individual patterns into a solution consisting of a system of patterns, a classification scheme is essential. A classification outlines the relationships between the various patterns, which enables designers to understand their individual capabilities and the relationships among the patterns when seeking to integrate different patterns into composite solutions.

The resilience design patterns may be classified on the basis of the type of event handled, whether the pattern offers detection, containment, recovery or masking semantics, the scope of protection coverage offered, design complexity of the patterns, time and space overheads, power consumption overheads, etc. However, in developing a classification for the resilience design patterns, our goal is to provide designers with the guidelines to identify the patterns that make up a resilience solution, specify the roles played by individual patterns, and how they interact,

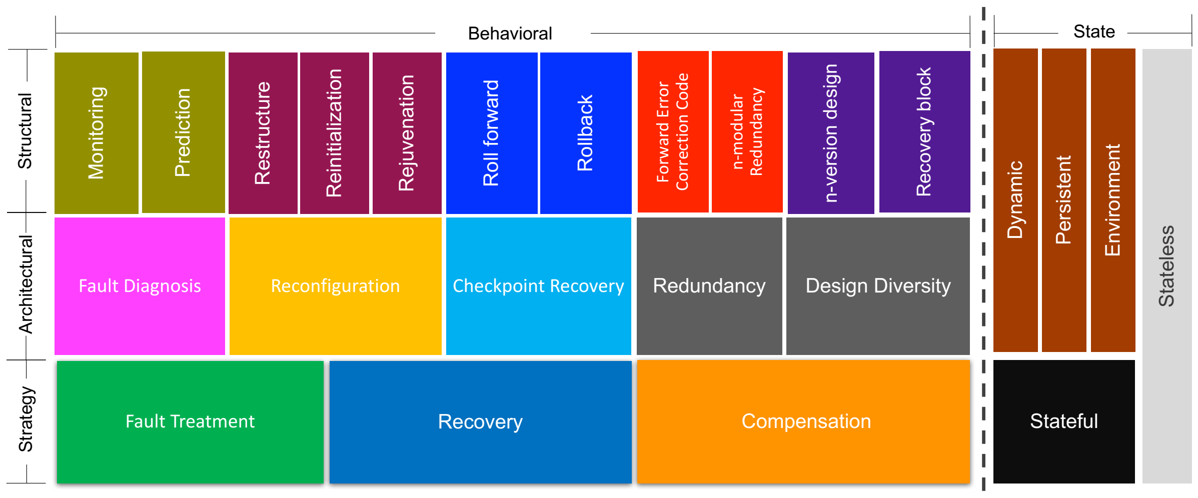


Figure 1. Classification of resilience design patterns

such that the incorporation of resilience capabilities becomes an essential part of the design process of HPC hardware and software components.

We develop a pattern classification scheme that organizes the resilience patterns in a layered hierarchy, in which each level addresses a specific aspect of the problem. Resilience in the context of HPC systems and its applications has two key dimensions: (1) forward progress of the system and (2) data consistency in the system. Based on these factors, we organize the resilience design patterns into two major categories, **state** patterns and **behavioral** patterns. These are placed side by side in Figure 1 to enable designers to separately reason about the patterns that define the scope of the protection domain and those that define the semantics of the detection, containment and mitigation. The behavioral patterns are organized in a hierarchy, as shown in Figure 1, in which the patterns in bottom layer may be used to think about the strategies suitable for confronting anomalous events depending on whether it is a fault, an error, or a failure. The patterns in the middle layer explicitly defines the architecture of a solution based on the nature of the event and considers compatibility of the pattern solution with the overall system design. The top-level patterns consider issues related to implementation of the solution, including the appropriate granularity and level of system abstraction, and the overheads incurred by the solution.

For the design and analysis of new solutions, or adapting existing solutions to emerging HPC environments, hardware and software designers can approach the hierarchy of patterns in a top down or bottom-up manner. The refinement and optimization of patterns will often require traversing the layers several times before a solution is finalized. The hierarchical organization of the patterns permits the different stakeholders to reason about resilience solutions based on their view of the system and their core expertise. Architects describe the overall organization of the solutions, analyze the integration of various resilience patterns across the system stack, and evaluate the protection coverage and overheads to overall system performance. The designers of individual hardware and software components operate within a single layer of system abstraction and focus on alternative patterns to address the problem at hand and the analyze the design complexity of instantiating a specific pattern.

2.1. State Patterns

The state patterns specify the protection domain of a resilience solution. The correctness and consistency of the system state ensures the correct operation of a system. Therefore, the precise definition of the scope of the protected system state is an important part of designing a resilience solution. From the perspective of an HPC application, the notion of state may be classified into three categories:

- *Static State*, which represents the data that is computed once in the initialization phase of the application and is unchanged thereafter.
- *Dynamic State*, which includes all the system state whose value may change during the computation.
- *Operating Environment State*, which includes the data needed to perform the computation, i.e., the program code, environment variables, libraries, etc.

The state patterns, which capture each of these aspects of the system state, are classified as *stateful* patterns. The properties of each state pattern may be used to guide the selection of a behavioral pattern. Certain resilience strategies may be applied without regard for state and apply behavioral patterns that are concerned with only the forward progress of the system (for e.g., idempotent operations). Therefore, the classification of state patterns also includes a *stateless* pattern that enable designers to create solutions that define behavior without state. This organization of the state patterns enables the behavioral patterns to be applied to individual aspects of a system's state. However, in designing a resilience solution, more than one type of state pattern may be fused to enable the use of a single behavioral pattern for more than one state pattern.

2.2. Behavioral Patterns

The behavioral patterns are concerned with forward progress of the system despite the presence of anomalous events in the system. These design patterns identify detection, containment, or mitigation actions that enable the components in a system that realize these patterns to cope with the presence of a fault, error, or failure event. The behavioral patterns are presented in a layered hierarchy to highlight the design choices when selecting one pattern over another:

- **Strategy Patterns:** These patterns define high-level polices of a resilience solution. The strategy patterns are organized by the type of event that they handle - fault, error or failure, since the techniques to handle these events are fundamentally different. The classification of the strategy patterns captures the intent behind of each solution makes the design choices in applying the patterns explicit. These patterns describe the overall structure and the key components in a solution in a manner independent of the layer of system stack and hardware/software architectural features. Their descriptions are deliberately abstract to enable hardware and software architects to reason about the overall organization of the solution and assess the suitability of the pattern to the full system design.
  The fault treatment patterns are concerned with diagnosing and preventing an imminent error or failure. The recovery and compensation patterns must limit and remove an error or failure state in the system. The recovery pattern aims to substitute an error/failure-free state in place of the erroneous/failed system state. The compensation pattern seeks to tolerate the presence of an error or failure by providing redundancy in the system design.

- **Architectural Patterns:** The architectural patterns convey specific methods necessary for the construction of a resilience solution. The patterns provide details about the key components and connectors that make up the solution and explicitly specify the type of event that they handle. These patterns are a sub-class of the strategy patterns, and they are also organized based on the type of event they handle and the intended impact of the action on the system resilience. Certain architectural patterns may be adapted to confront faults, errors or failures. Consequently, there exists an overlap between the patterns in the architectural layer with more than one type of strategy pattern in Figure 1. The classification of these architectural patterns based on the core solution is also suggestive of the design time and runtime complexity encountered when instantiating a pattern. Yet, architectural pattern descriptions are independent of the precise fault model and may be implemented at any layer of the system stack.
- **Structural Patterns:** These patterns provide concrete descriptions of the solution rather than high-level strategies. While the strategy and architectural patterns serve to provide designers with a clear overall framework of a solution and the type of events that it can handle, the structural patterns express the details such that they can contribute to the development of complete working solutions. They comprise of specific instructions for implementing the pattern, including concrete descriptions of the key parts of the solution. Their descriptions include specific details of the fault model that the pattern handles. Although the structural patterns provide more detailed solutions, their descriptions are still independent of the layer of system abstraction at which the patterns may be instantiated. The pattern descriptions are flexible enough for most, if not all structural patterns to be suitable for implementation within hardware structures as well as within algorithms in the application or system software. The various structural patterns are sub-classes of the strategy and architectural patterns. Therefore, their first-order organization is also based on the type of fault event that their solutions handle.

A variety of **implementation patterns** may be derived from the structural patterns. These patterns are intended to bridge the gap between the design principles and the concrete details of an implementation. The pattern descriptions explicitly specify the layer of system abstraction at which they are implemented, and the activation and response interfaces. The implementation patterns also enable a standardized way for hardware and software designers to communicate about design of their resilience solutions. These patterns may be designed as composite patterns, i.e., using a combination of patterns. Defining implementation patterns enables designers to thoroughly analyze the overhead of a solution in terms of time and space, as well as the trade-off between design complexity and runtime complexity. Due to the limitless possibilities in developing implementation patterns suited for various architectures, software environments and HPC applications through pattern composition, we only provide detailed descriptions of the foundational state and behavioral resilience patterns in this paper.

3. Resilience Design Pattern Catalog

The resilience design pattern catalog contains detailed descriptions of the state and behavioral patterns. The primary objective of the catalog is to capture the best-known HPC resilience solutions and present them a standardized and accessible form. For the patterns to be useful to HPC system architects and individual hardware and software component designers alike, they are written down in a highly structured format to enable designers to quickly discover whether the pattern solution is suitable to the problem being solved.

For convenience and clarity, each resilience pattern in the catalog follows the same prescribed format. The pattern description is formatted in terms of the following key attributes:

- ***Name***: Identifies a pattern and provides a convenient way to refer to it, typically using a short phrase.
- ***Problem***: A description of a problem indicating the intent behind applying the pattern. This describes the goals and objectives that will be accomplished with the use of this specific pattern.
- ***Context***: The preconditions under which the pattern is relevant, including a description of the system before the pattern is applied.
- ***Forces***: A description of the relevant forces and constraints, and how they interact or conflict with each other, and with the intended goals and objectives. The forces highlight the intricacies of the problem and make the trade-offs that must be considered explicit.
- ***Solution***: A description of the solution that includes specifics of how to achieve the intended goals and objectives. This includes the core structure of the solution, its semantics and its interactions with other patterns. The description includes guidelines for implementing the solution, as well as descriptions of variations or specializations of the solution.
- ***Capability***: The resilience management capabilities provided by this pattern, which may include detection, containment, mitigation, or a combination of these capabilities.
- ***Protection Domain***: The resiliency behavior provided by the pattern extends over a certain scope, which may not always be explicit. The description of the nature of the fault model and its protection domain enables designers to reason about the scope of the coverage in terms of the complete system.
- ***Resulting Context***: A brief description of the post-conditions arising from the application of the pattern. There may be trade-offs between competing optimization parameters that arise due to the use of this pattern.
- ***Examples***: One or more sample applications of the pattern, which illustrate the use of the pattern for a specific problem, the context, and set of forces. This also includes a description of how the pattern is applied, and the resulting context.
- ***Rationale***: An explanation of the pattern as a whole with an elaborate description of how the pattern actually works for specific situations. This provides insight into its internal workings of a resilience pattern, including details on how the pattern accomplishes the intended goals.
- ***Related Patterns***: The relationships between this pattern and other relevant patterns. These patterns may be predecessor or successor patterns in the hierarchical classification, or patterns that provide similar capabilities.
- ***Known Uses***: Known applications of the pattern in existing HPC systems, including any practical considerations and limitations that arise due to the use of the pattern at scale in production HPC environments.

There are three key reasons behind this pattern format: (1) to present the pattern solution in a manner that simplifies comparison of the capabilities of patterns and their use in developing complete resilience solutions, (2) to present the solution in a sufficiently abstract manner that designers may modify the solution depending on the context and other optimization parameters, and (3) to enable these patterns to be instantiated at different layers in the system.

The complete catalog of resilience design patterns in the template format is available in a specification document [38]. In the remainder of this section, we summarize each design pattern, highlighting its key features. The pattern descriptions use the term *system* to refer to an entity that has the notion of a well-defined structure and behavior. A *subsystem* is a set of elements, which is a system itself, and is a component of a larger system, i.e., a system is composed of multiple sub-systems or components. For a HPC system architect, the scope of system may include compute nodes, I/O nodes, network interfaces, disks, etc., while an application developer may refer to a library interface, a function, or even a single variable as a system. A *full system* refers to the HPC system as a whole or to a collection of nodes, which is capable of running a parallel application.

3.1. Strategy Patterns

3.1.1. Fault Treatment Pattern

The emergence of a defect or anomaly in an HPC system environment has the potential to activate, which may potentially lead to an error or a failure in the system. The `Fault Treatment` pattern provides a method that attempts to recognize the presence of an anomaly or a defect within a system, and creates conditions that prevents the activation of the fault into an error or failed state. The solution requires an auxiliary monitoring system, which may be a sub-system of the monitored system or an external system, that observes the key parameters of the monitored system. The pattern applies to a system that has well-defined parameters that may be used to discover the presence of anomalies in the behavior of the monitored system. The pattern supports either one, or both of the following capabilities:

- **Fault detection**: detect anomalies during operation before they impact the correctness of the system state.
- **Fault mitigation**: methods to enable an imminent error or failure to be prevented, or a defect to be removed.

The protection domain of this pattern extends to the scope of the monitored system and implicitly extends to other systems that are interfaced to the monitored system. The benefit of incorporating fault treatment patterns in a design, or deploying it during system operation is to preemptively recognize faults in the system; the preventive actions avoid the need for expensive recovery and/or compensation actions that may be necessary if the fault activation causes an error or failure. In incorporating this pattern in the design of a HPC hardware or software component, the key considerations are the frequency of interaction between the monitoring and monitored (sub-)systems and the precision of fault detection. The frequency of these interactions must be minimized to reduce interference in the operation of the monitored system; yet, the interactions must be frequent enough to detect every defect in the monitored system. Also, fault must be detected and treated in a timely manner, i.e., the time interval for the monitoring system to gather data about the monitored system and to infer the presence of an anomaly or a defect must be rapid to prevent the activation of an error/failure. The pattern must also have few false positive and false negatives to minimize preemptive mitigation actions that are unnecessary.

In HPC systems, various hardware-based solutions for fault detection observe the attributes of a system, such as thermal state, timing violations in order to determine the presence of a defect in the behavior of the system that may potentially cause an error or failure. For example, processor chips such as the IBM Power 8 and Intel Xeon series processors contain thermal sensors that detect anomalous conditions in the cores. Software-based solutions detect the anomalies in the behavior of a system's data variables or control flow attributes to determine the presence of a fault. Heartbeat monitoring is used for liveness checking of MPI processes, which enables detection of imminent failure of the MPI communicator [6].

3.1.2. Recovery Pattern

In an HPC environment, the occurrence of errors or failures in the system results in incorrect answers, and in some cases, catastrophic application crashes. The `Recovery` pattern enables a system to survive an error or failure event. The pattern is suitable for a system whose design or runtime configuration contains no intrinsic support for tolerating the error or failure event. The solution is based on the periodic creation of snapshots of the system state during error/failure-free operation, and the maintenance of these snapshots persistently. Upon detection of an error or a failure, the preserved snapshots are used to recreate a known error/failure-free state of the system. When the system state is recovered, the operation of the system is resumed. The error or failure in the system must be detected; the pattern offers no implicit error/failure detection. The pattern applies to a system that is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization. The pattern requires the system state can be compartmentalized in a form that is accurately representative of the progress of the system since initialization. It also requires that the system has well-defined intervals that enables it to transition the system state to a known correct interval in response to an error/failure.

The protection domain for a `Recovery` pattern is determined by the scope of the state pattern that is captured during checkpoint creation operation. The size and frequency of creation of checkpoints determines the overhead to system operation; frequent checkpointing incurs proportionally greater overheads during error/failure-free operation, but reduces the amount of lost work when an error/failure event does occur. Also, the broader the scope of the system state that is preserved, the larger is the scope of the system state that may be protected from an error/failure event. The solution offered by this pattern is not dependent on the precise semantics of the error/failure propagation. Therefore, the effort and complexity in using this pattern in a hardware or software design, or in the system configuration is low. There are several instances of the usage of the pattern in HPC systems to support recovery of an application or the complete system upon detection of an error/failure. For example, various checkpoint and rollback protocols enable HPC applications and systems to capture state and commit the checkpoint files to parallel file systems [27].

3.1.3. Compensation Pattern

The occurrence of an error or failure event may cause loss in system functionality, or reduction in system capacity. The HPC applications running on such a system may produce incorrect results or experience failure. The `Compensation` pattern makes up for the deficiency or abnormality in a system that is caused by the error or failure event. The pattern solution introduces redundancy with into the system design, or in the configuration to counterbalance the (sub-)systems in error or failed state. The pattern is applicable to a system that is deterministic, and the overall system design allows for modular design with well-defined inputs and outputs for each module, about which redundant information is maintained. The redundancy may be in the form of a group of replicas of a (sub-)system, referred to as n-modular redundancy, or in the form of encoded information about the (sub-)system state. The pattern supports detection, and in some cases correction, by using the redundant information about a (sub-) system to recompense for the presence of an error/failure. The scope of the protection domain, which covers includes the part of the system designed or operated redundantly, may include a sub-system, or the cover the full system.

The replicas of the modules permit the system to continue operation even in the presence of a (sub-)system failure. When the redundancy is in the form of modular replication, an error or failure in one of the (sub-)systems may be tolerated by substituting the (sub-)system with a replica. In order to recover from 2N errors/failures in the system, there must be 2N + 1 distinct replicas. For the detection of errors, the outputs of the replicas of the system are compared by an auxiliary monitor (sub)-system. For a system to tolerate an error/failure, the number of replicas must be greater than two, in which case the monitor performs majority voting on the outputs produced by the replicas. This enables incorrect outputs from replicas in erroneous state to be filtered out. The design effort and complexity of replication of the system depends on the replication method: deploying identical replicas requires low design effort, but the design of functionally identical but independently designed versions of a (sub-)system requires much higher design and verification effort.

The scope and strength of the redundancy employed by the pattern determines the overhead to the system performance. The pattern introduces a penalty in terms of time (increase in execution time), or space (increase in resources required) independent of whether an errors or failure occurs during system operation. The N-modular redundancy approach is used at the hardware and software levels in a various HPC components; the dual-modular redundancy (DMR) for error detection and triple-modular redundancy (TMR) for error detection and correction [42] are the most widely used forms of redundancy. Redundant information in the form of error correction codes is also used at the hardware-level in the form of ECC [49] and at the application-level for application data structures [37].

3.2. Architectural Patterns

3.2.1. Fault Diagnosis Pattern

The occurrence of a defect or anomaly has the potential to activate causing an error or failure in the system. The `Fault Diagnosis` pattern, which is a derivative of the `Fault Treatment` strategy pattern, identifies the presence of the fault and determines its root cause. The solution consists of an auxiliary monitoring system that observes specific parameters of a monitored system. Until a fault has not activated into an error it does not affect the correct operation of the system. Therefore, the `Fault Diagnosis` pattern makes an assessment about the presence of a defect based on observed behavior of one or more system parameters. The inference is based on observing deviations in the standard operating behavior of the monitored system. Identifying the norm of (sub-)system parameters also enables narrowing the search for the fault type, its location and its root cause. To incorporate this pattern in an HPC environment requires inclusion of a monitoring (sub-)system, which introduces additional complexity in the overall system design. When the monitoring system is extrinsic to the monitored system, the design effort may be simplified, but the interfaces between the (sub-)systems must be well-defined. The pattern only infers the presence of a defect and reports it via its response interface, but does not act to remedy the fault. Among the key design challenges when using the pattern is the resolution limit, which is influenced by the number of parameters observed and frequency of probing the monitored (sub-)system and affects the precision of the fault detection. In the context of HPC systems, faults may be detected and diagnosed based by accumulating empirical data on the characteristics and the behavior of hardware and software components and use the information to discover faults. For example, HPC components commonly use the Intelligent Platform Management Interface (IPMI) [21], which provides standardized interfaces for monitoring hardware health information such as the system temperatures, fans, power supplies, etc. Using these interfaces, software tools may monitor the health of system resources and infer the presence of anomalies in the components.

3.2.2. Reconfiguration Pattern

In the event of a fault, error or failure event the configuration, i.e., the organization of the (sub-)systems in an HPC environment may be affected in ways that result in applications producing incorrect results, or experiencing fatal crashes. The `Reconfiguration` pattern, which derives from the `Fault Treatment` and `Recovery` strategy patterns, entails modification of the interconnection between (sub)-systems. The reconfiguration isolates the (sub-)system affected by the event to prevent it from affecting the correct operation of the overall system. The pattern assumes that the system may be partitioned into a set of logical modules and that altering the interconnection between the modules is possible. The protection domain of the `Reconfiguration` pattern covers all (sub-)systems that are interconnected to provide a specified function. The pattern may cause the system to assume several configurations in response to a fault, error or failure event, each of which is characterized by its own topology of interconnections, the system must retain functional equivalency with the original system configuration. The performance overhead of using this pattern is proportional to the number of (sub-)systems and degree of interconnection between them. The reconfiguration of the system may also result in system operation at a degraded performance level. The implementation of the pattern requires partitioning the system into modules that remain functionally correct in multiple different configurations. There is much complexity associated with defining the scope of these modules and to validate their functional equivalency in alternative configurations. Well-known use cases of the reconfiguration pattern include the NodeKARE module in the Cray Linux Environment CLE, which automatically runs diagnostics on all involved compute nodes in the cluster whenever a users program terminates abnormally and removes the failing nodes from the pool of available compute nodes so that subsequent jobs are allocated only to healthy nodes [18].

3.2.3. Checkpoint Recovery Pattern

Errors or failures in an HPC environment may result in conditions that prevent forward progress of the system until the error or failure condition is removed. The `Checkpoint-Recovery` pattern, which is a specialization of the `Recovery` strategy pattern, is based on the creation of snapshots of the system state and maintenance of these checkpoints on a persistent storage system during the error- or failure-free operation of the system. Upon detection of an error or a failure, the checkpoints/logged events are used to recreate last known error- or failure-free state of the system, after which the system operation is restarted. The solution offered by the pattern supports only recovery; the detection and containment of the error/failure is beyond the scope of the pattern's capabilities. The pattern assumes that the system is capable of compartmentalizing its state in a way that is accurately representative of the progress of the system since initialization. The techniques used by the pattern are classified into checkpoint-based and log-based strategies. The checkpoint-based solution typically captures and preserves the complete state of the system; in contrast, log-based strategies only record specific system events. Instantiations of the pattern may also use a combination of checkpointing and event logging. The pattern handles an error or a failure by retrieving a version of the error or failure-free state from the checkpointed state, and substituting the erroneous or failed state with the error or failure-free state. Therefore, the system is able to resume operation with a version of the system state that is free of any effects of the error or failure event.

However, the pattern requires interruption of the system during error or failure-free operation to record the checkpoint, which incurs an overhead. The frequency of creation of checkpoints and/or event logging determines the extent of the overhead; frequent checkpointing/logging incurs proportionally greater overheads during error- or failure-free operation. However, more frequent checkpointing and logging reduces the amount of lost work when the system encounters an error or failure event. The checkpointing/logging latency affects the overhead during error- or failure-free operation on account of the latency to write the checkpoint to a storage system. The scope of the system state captured during a checkpointing operation results in a proportionate increase in space overhead due to the storage resources needed to preserve the checkpoints. The solution offered by this pattern is independent of the type of error or failure and its mode of propagation. Therefore, the design effort and complexity in instantiating this pattern in any system design in low. In the context of HPC systems, checkpoint and restart capabilities in the software layers, including various library-based and operating system-based solutions such as BLCR [25] for Linux processes. Certain library implementations of the MPI standard, such as OpenMPI, also support transparent checkpoint-restart [39]. Log-based recovery based on message logging has been adopted by implementations of MPI [10].

3.2.4. Redundancy Pattern

When an error or failure event in an HPC environment cannot be prevented from affecting the correct operation of a component, or the full system, it must be remedied to enable forward progress of the system. The `Redundancy` pattern, which is a derivative of the `Compensation` pattern, enables offsetting the effects of the error/failure. The pattern solution entails incorporating excess resources in the (sub-)system design or in the configuration at runtime. The redundancy enables a (sub-)system to detect, and in certain cases correct an error/failure, by repetition, omission of a (sub-)system without loss of functionality, or superfluity of (sub-)system state information. The pattern applies to a (sub-)system that allows for a modular design with well-defined inputs and outputs for each module. The application of a `Redundancy` architecture pattern, the following error/failure handling capabilities can be supported:

- **Detection by comparison**: observing the likeness of each replica's outputs as means to detect the presence of an error or failure in each redundant version of a (sub-)system;
- **Fail-over mitigation**: substitution of a replica in error or failed state with another identical replica that is error/failure-free.

- **Mitigation by isolation**: creation of a group of N replicas of a (sub-)system and majority voting on the outputs produced by each replica; the outputs that fall outside the majority are excluded.; and
- **Encoding information for detection and mitigation**: maintenance of additional (sub-)system state information to identify errors within the state.

The protection domain of the pattern extends to the scope of the (sub-)system state about which redundant information is maintained. The pattern introduces penalty in terms of time (increase in execution time), or space (increase in resources required) independent of whether an errors or failure occurs. The use of dual-modular redundancy for error detection and triple-modular redundancy for error/failure detection and correction are common forms of instantiation of the pattern in various hardware and software-level modules. HPC systems contain service nodes that are responsible for system management tasks while the parallel computation is performed by a set of compute nodes. The tasks include user login, network file system, job and resource management, communication services. Various existing solutions provide hot-standby redundancy with transparent fail-over to tolerate failures in the critical services in the service nodes. Well-known examples of redundancy are the scheduling and resource management services in Simple Linux Utility for Resource Management (SLURM) [53] , as well as the metadata servers of the Parallel Virtual File System (PVFS) [15] and the Lustre file system [2]. Production HPC systems such as the Cray XC40 series [19] include redundant power supplies, voltage regulator modules and cooling fans to ensure continuous operation in the event that one of these units experience malfunction or failure.

3.2.5. Design Diversity Pattern

Design flaws on account of human error or defective tools manifest themselves as errors, which may cause failures in HPC environments. The `Design Diversity` pattern, which is also a derivative of the `Compensation` pattern, creates distinct but functionally equivalent versions of the same design specification, which are created by different individuals or teams, or developed using different tools. The intent behind applying this pattern is to eliminate the impact of design bugs during the implementation of a (sub-)system. The pattern enables systems to tolerate errors/failures due to design faults that may arise on account of incorrect interpretation of the specifications by designers, mistakes made during implementation, or due to bugs in the tools. The detection and correction of error/failures is possible due to the independent design processes reducing the likelihood that the same flaw emerges in the alternative versions of a (sub-)system. The pattern is based on the assumption that the system has a well-defined specification for which multiple implementation variants may be created. The versions of the (sub-)system specification may be applied to a system in a time or space redundant manner. The replica (sub-)systems are provided with identical inputs, and their respective outputs are compared in order to detect and potentially correct the impact of an error or a failure in either replica of the systems. The protection domain of the pattern extends to the scope of the system that is described by the design specification. However, designing multiple variants of the same (sub-)system specification requires significantly higher verification and validation effort. The design diversity solution is used in the validation of the results produced by scientific applications, particularly those that require high-precision floating point arithmetic. Such applications may be compiled and executed using alternative implementations of compiler toolchains, message passing libraries, numerical analysis libraries to verify the application results.

3.3. Structural Patterns

3.3.1. Monitoring Pattern

The various types of errors in HPC environments occur as a result of underlying defects in hardware or software components. Identifying the defects before they cause an error, which may result in a failure of one or more components, prevents incorrect behavior of a (sub-)system. The `Monitoring Pattern` is a specialization of the `Fault Diagnosis` architectural pattern, which consists of a monitoring system that observes specific parameters of a monitored system to discover the presence of anomalies in its behavior. The monitoring system may approach the problem of fault detection using two strategies:

- *Effect-Cause Diagnosis*: This approach entails observation of the parameters of the (sub-)system for anomalies. When a (sub-)system parameter deviates from a range of values considered *normal*, the monitoring system attempts to determine the root cause. The monitoring system logically partitions the system into modules and progressively eliminates the modules known to be fault-free. Through this process, it narrows the search for the fault in the (sub-)system.
- *Cause-Effect Diagnosis*: This approach is based on a set of known fault models and the monitoring system compares the (sub-)system parameters with a model developed using fault free system operation, or using simulations. When the observed set of parameters deviates from a model, the presence of and the type of fault may be inferred.

Based on these inferences, the pattern enables the monitored system to report the presence of a fault and to analyze its root cause and location. The (sub-)system design or configuration must include a monitoring (sub-)system. When the monitoring system is extrinsic to the monitored system, the design effort may be simplified, but the interfaces between the (sub-)systems must be well-defined. However, when the monitoring system is intrinsic to the design or configuration of the monitored system, the complexity of the design process increases. The `Monitoring` pattern only infers the presence of a defect and reports it, but does not remedy the defect. Various HPC system installations use the monitoring pattern through tools for collecting performance- or health-related data about the system. Popular solutions include: Ganglia Monitoring System [44], Nagios [1] and OVIS Lightweight Distributed Monitoring System [3].

3.3.2. Prediction Pattern

The accurate prediction of where faults are likely to occur in a (sub-)system enables reduction in the costs of a resilience solution by preemptively enhancing the (sub-)system's capabilities to handle any resulting errors or failures. The `Prediction Pattern`, which is also a derivative of the `Fault Diagnosis` architectural pattern, develops models that estimate future faults based on the observations of the parameters of a (sub-)system, or based historical trend analysis of these parameters. For prediction, the pattern may use: (i) *Rule-based methods* that build rules of association to capture the causal correlations between system parameter values and fault events, or (ii)*Statistical-based methods* that discover probabilistic characteristics of potential errors/failures in a system using statistical inference techniques to examine correlations between previous events. The monitoring system of this pattern contains the following components:

- *Filter/Preprocessor*: removes incomplete fault data and duplicates and produces a consistent format for analysis.
- *Regression*: seeks to analyze the parameter values and establish relationships between them.
- *Knowledge Base*: storage component that maintains the rules or statistical properties and models, which may be used for online prediction of fault events using real-time data captured from the monitored system.

Much like the `Monitoring Pattern`, the `Prediction` pattern only infers the presence of a defect and reports it, but does act to remedy the fault. Based on the prediction method and accessibility of the system parameters selected for observation, the prediction may not be very precise, which leads to false positive outcomes, or unforeseen events that are missed by the prediction algorithm. However, when errors or failures are predicted at a high degree of accuracy, avoidance or preventative actions may be applied. For example, event prediction may be used for proactive management in large-scale clusters [51].

3.3.3. Restructure Pattern

The occurrence of a fault, error, or failure event sometimes impacts a system in a way that affects the correctness of the interactions between sub-systems in an HPC environment, which causes further errors, or a failure of the system. The `Restructure Pattern`, a derivative of the `Reconfiguration` pattern, modifies the configuration between the interconnected sub-systems to isolate the specific sub-system affected by a fault, error or failure. The reconfiguration pattern alters the organization of the (sub-)systems to work around the affected (sub-)system, or it excludes the affected (sub-)system from interacting with the remaining (sub-)systems (i.e., the restructured system includes N-1 sub-systems). In either case, the pattern seeks to maintain (sub-)system functionality equivalent to that before the occurrence of the fault, error or failure event.

The protection domain of the pattern spans the part of (sub-)system whose constituent sub-systems may be reconfigured. While the pattern seeks to restructure the sub-systems in an operating state that is functionally equivalent to the fault-free state, the pattern may result in the operation of the system in degraded condition, which incurs additional time overhead to the system. Existing solutions that restructure the system in response to an event include the ULFM extension to the MPI standard [7], which allows parallel applications to get notifications of process failures. ULFM provides a set of routines to revoke and restructure a MPI communicator that consists of the remaining active processes. Dynamic page retirement is another instantiation of the restructure pattern solution, in which pages that have an history of frequent memory errors are removed from the pool of available pages.

3.3.4. Rejuvenation Pattern

When a (sub-)system in an HPC environment behaves incorrectly on account of a fault, error or failure, the correctness of the full system may be compromised. The `Rejuvenation Pattern`, which is also a derivative of the `Reconfiguration` pattern, isolates the specific part of the (sub-)system affected by a fault, error or failure and restores it to an operating state that is free of any effects of the event. Only the affected part of the system is rejuvenated to ensure correct operation of the system by the pattern. The pattern requires the system operation to be halted to identify the part of the system affected by the event.

The protection domain of the pattern spans the part of system whose state may be rejuvenated. The rejuvenation is often a slow process that requires substantial additional overhead to identify the part of the system affected by the fault, error or failure, and to selectively reinitialize the system, in addition to overhead incurred due to any lost work. The rejuvenated system may not maintain the level of performance as before the occurrence of an event. Examples of rejuvenation include the Mini-Ckpts framework, which recovers fatal operating system crashes by rejuvenating only the kernel data structures, which are preserved in persistent memory, without affecting the HPC application state [31]. Algorithm-based recovery methods for data corruptions in structures used in numerical analysis problems use interpolation of neighboring data values to rejuvenate data values in error state. Such methods have been demonstrated in the context of the HartreeFock algorithm used in computational chemistry codes [20].

3.3.5. Reinitialization Pattern

The impact of a fault, error or failure may sometimes be irreversible such that the affected (sub-)system cannot be restored to a form that permits correct operation. The `Reinitialization Pattern`, also a derivative of the `Reconfiguration` pattern, simply restores the system to its initial state. This causes system operation to *restart* with a pristine reset of state, which implicitly cleans up the effects of the fault, error or failure in the system. The pattern is applied in conditions in which the mitigation or recovery from the fault, error or failure event is deemed impossible, or excessively expensive in terms of overhead to performance. The pattern expects the fault, error or failure in the system to be detected; the pattern offers no implicit fault monitoring, prediction, or error/failure detection capability. The restoral of the system state to the initial state causes lost work, but guarantees the impact of the event is completely removed before service is resumed. Various cluster management software systems, such as the Cray Hardware Supervisory System (HSS) [18], enable malfunctioning nodes in the cluster to be reset. The HSS initiates a reboot sequence for a failing node without disrupting the remaining nodes in the system.

3.3.6. Rollback Pattern

Following an error or a failure event, the (sub-)systems in a HPC environment often lose all work performed until the occurrence of the event. The `Roll-back Pattern`, which derives from the `Checkpoint Recovery` architectural pattern, periodically captures the progress of the system and maintains these as system snapshots on a persistent storage system during the error/failure-free operation of the system. The rollback recovery is performed by restoring the system state based on the last known stable version of (sub-)system state. The solution provides rollback recovery, i.e., based on a temporal view of the system's progress, the system state restored during the error/failure recovery process is a previous error/failure-free state of the system. For a system that is deterministic, the pattern creates checkpoints of the system, which requires the capability to export the current (sub-)system state and import a new state during recovery. When the system design consists of several sub-systems, the pattern must coordinate the process of checkpointing. The instantiation of the pattern may apply the following coordination policies:

- *Coordinated rollback recovery protocol*: The (sub-)systems coordinate the process of creating checkpoints, creating globally consistent checkpoint states, which simplify the recovery.
- *Uncoordinated roll-back recovery protocol*: The (sub-)systems each independently decide when to create their respective checkpoints. This approach has the potential to cause the full-system to propagate roll-backs to the initial system state to ensure that all dependencies are met (called the *domino effect*).
- *Communication-based rollback recovery protocol*: The protocol enables each (sub-)system to create *local* checkpoints, but periodically also enforces coordinated checkpoints between all (sub-)systems. Such hybrid strategy helps avoid the *domino effect.*

For systems with non-deterministic events, the pattern employs log-based protocols, which use a combination of checkpointing and logging of non-deterministic events in the (sub-)system. The log-based rollback recovery is based on piecewise deterministic assumption, in which the system identifies and records the nondeterministic events and information necessary (encoded in tuples called *determinants*) to replay the event during recovery. The pattern may use the following logging protocols:

- *Pessimistic*: The protocol assumes that a failure occurs after a nondeterministic event in the system. Therefore, the determinant of each nondeterministic event is immediately logged to stable storage.
- *Optimistic*: The determinants are held in a volatile storage and written stable storage asynchronously. This protocol makes the assumption that the logging is completed before the occurrence of an error or failure. The error- or failure-free overhead of the optimistic approach is low.
- *Causal*: The protocol provides a balanced approach by avoiding immediate writing to stable storage (much like the optimistic protocol in order to reduce event free overhead), but each sub-system commits output independently (much like the pessimistic protocol in order to prevent creation of orphan sub-systems in the context of a multicomponent environment).

The protection domain for a `Rollback` pattern is determined by the extent of state captured during checkpoint operation and/or the number of system operations that can be recovered from the log of events. The time overhead introduced by the use of the pattern during error-free operation is correlated with the frequency of taking checkpoints. The rollback leads to loss of work due to the need to recover the system from a previous version of the system state. The amount of lost work is also correlated with the frequency of the checkpointing/logging. The worst-case scenario for recovery using this pattern is a roll-back to the initial state of the system. In the context of HPC systems, checkpoint and restart capabilities in the software layers, including various library-based and operating system-based solutions, enable recovery from process errors/failures and rollback of the applications. Well-known solutions that employ the rollback recovery pattern include the CoCheck checkpoint-restart for MPI [56], as well as BLCR [25] and SCR [48]. Message logging protocols have been implemented in OpenMPI to support faster failure recovery [10].

3.3.7. Roll-forward Pattern

When an error or failure event occurs in an HPC environment, a (sub-)system incurs loss of the work performed prior to the occurrence of the event. The `Roll-forward` pattern is a derivative of the `Checkpoint Recovery` pattern that avoids loss of work by using checkpoints to recover the (sub-)system to a stable state immediately before the error or failure event. Like the `Rollback` pattern, the solution entails the creation of snapshots of the system state and maintenance of these checkpoints on a stable storage system during the error- or failure-free operation of the system; log-based protocols use a combination of checkpointing and logging of non-deterministic events in the (sub-)system. However, the pattern uses the previously captured checkpointed state and/or logging information to recreate a stable version of the (sub-)system state identical to the one right before the error or failure occurred. This prevents the need for re-execution of all (sub-)system operations from the last stable checkpoint. The pattern must select checkpointing based on the policies similar to those used by the `Rollback` pattern: coordinated, uncoordinated, or communication-based.

The pattern may use the following protocols for roll forward:

- *Log-based protocols:* Based on the piecewise deterministic assumption, in which the (sub-) system uses the determinants to recreate state. The logging mechanisms may be based on *pessimistic*, *optimistic*, or *causal* protocols; and
- *Online recovery protocols:* Do not rely on event logging for roll forward of the (sub-) system; rather, they use inference methods to recreate state, or may permit the state to self-correct after restart.

The protection domain of a `Roll-forward` pattern is determined by the extent of state captured during checkpoint operation and/or the number of system operations that can be recovered from the log of events. The pattern solution is not dependent on either the type of event, or the precise semantics of the error propagation; therefore, the design complexity in using this pattern in any HPC (sub-)system design in low. For the pattern to be effective in an HPC environment, the overhead to bring the system state to the most recent state before the error or failure must be less than or equal to the overhead of rollback recovery. In the context of HPC systems, software solutions typically implement roll-forward recovery using algorithm-specific knowledge. For example, Global View of Resilience (GVR) [13] uses versioning of distributed arrays supports, in which roll-forward recovery is based on application-specified mechanisms for each array structure.

3.3.8. N-modular Redundancy Pattern

An error or failure of a (sub-)system in an HPC environment may cause loss in system capability or capacity, which prevents correct operation, or failure, of the full system. The `N-modular Redundancy Pattern`, which is a derivative of the `Redundancy` architectural pattern, remedies the effect of the error or failure by isolating the affected (sub-)system and compensating for its removal from the system design or configuration with a replica module. The solution entails creation of a group of N replicas of a (sub-)system. The replicated versions of a (sub-)system enables their use in various configurations to support errors or failures in one of the replicas, including fail-over, active comparison for error detection, or majority voting for detection and correction by excluding the replica whose outputs fall outside the majority. The pattern applies to a system with a modular design that has a well-defined scope and set of inputs and outputs. The scope of the pattern may be a sub-system in the HPC hardware or software architecture, or it may even encompass the complete system scope. Each of the N modules of the system exist simultaneously; the modules may be active at the same time (spatial replication), or may operate in succedent order (temporal replication), or the (sub-)system may activate the redundant modules on-demand. The protection domain of the pattern extends to the scope of the module that is replicated. Implementations of the MPI standard use these forms of redundancy for MPI messages, or even by replicating MPI process ranks; the MR-MPI [28], rMPI [29] and RedMPI [30] are well-known MPI implementations using the n-modular redundancy approach.

3.3.9. Forward Error Correction Code Pattern

When the state information of a (sub)-system is affected by an error, the incorrect state often leads to malfunctioning of the (sub-)system, which may lead to the failure of the full system. The `Forward Error Correction Code Pattern`, which is a derivative of the `Redundancy` architectural pattern, maintains redundant information about (sub-)system state. The pattern applies to a system whose state may be represented using a sequence of symbols. The solution consists of an encoder and a decoder module. In the simplest form, the encoder repeats each symbol that represents the (sub-)system state. The decoder module checks both instances of each state symbol. The general form of this pattern uses an encoder module that accepts `k` state information symbols and separately appends a set of `r` redundant symbols that are derived from the symbols representing (sub-)system state. The output of the encoder module is a (n, k) code, in which n = k+r. While the encoded redundant state information is a complex function of the original state, the encoder module does not modify the state information. The decoder module extracts the original state from the encoded state symbols. The availability of redundant state information enables recovery of system from corruption in symbols that represent the (sub-)system state by using the redundant information to reconstruct the original state information.

The protection domain of the pattern extends to the scope of the (sub-)state that is encoded and decoded using the forward error correction code. The number of errors that are detectable and correctable is limited by the amount of redundant information contained in the error correction code. Since every operation that affects the system state requires encoding/decoding operations, the pattern introduces penalty in terms of time (increase in state information access latency), and space (increase in resources required to store state information) independent of whether an errors or failure occurs. There are various schemes that enable forward error correction in memory devices, storage systems, as well as in communication channels in HPC systems. Examples of forward error correction code (FEC) in HPC environments include parity bits, checksums, Hamming codes, hash function codes; more elaborate schemes such as systematic cyclic block codes include binary BCH, Reed-Solomon, Cyclic redundancy checks (CRC). The use of ECC in memory DIMMs is another well-known example of FEC for compensation of bit flip errors within the DRAM memory lines [49]. Algorithm-based methods use FEC schemes such as checksums to detect and correct errors in application data structures [37].

3.3.10. N-version Design Pattern

When a design bug exists in a (sub-)system design or configuration, the resulting error or failure is often unavoidable. Therefore, the detection and mitigation of the impact of such errors or failures is critical. The `N-version Design Pattern`, which is a derivative of the `Design Diversity` pattern, applies distinct implementations of the same design specification created by different individuals or teams. The pattern applies N (N ¿= 2) independently implemented versions in a time or space redundant manner. The N versions of the (sub-)system are operated simultaneously, and a majority voting logic is used to compare the results produced by each design version. Due the low likelihood that different individuals or teams make identical errors in their respective implementations, the pattern enables compensating for errors or failures caused by a bug in any one implementation version.

The pattern applies to a system that has a well-defined specification for which multiple implementation variants may be designed. The protection domain extends to the scope of the system that is described by the design specification. The extent to which each of the n versions are different affects the ability of the pattern to tolerate errors/failures in the system. The use of the n-version design pattern requires significant effort for design, implementation, testing and validation of the independent versions of a (sub-)system specification. Differences in the design may cause differences in timing in generating output values for comparison and majority voting; these differences incur overhead to the overall (sub-)system operation.

3.3.11. Recovery Block Pattern

The errors and failures caused by design bugs prevent HPC (sub-)systems from operating in conformance with the (sub-)system specification. Yet, the application of the `N-version Design` pattern may be impractical in various contexts. The `Recovery Block Pattern`, which is also a derivative of the `Design Diversity` pattern, introduces an alternative implementation of the same design specification to perform detection and mitigation of errors. The pattern is a specialization of the `N-version Design` pattern since the solution also relies on multiple variants of a design that are functionally equivalent but designed independently. The recovery block is invoked when the result from the primary version of the system fails an acceptance test, which often indicates the presence of an error or failure. The instantiation of this pattern may sometimes include the function that performs the acceptance test. The consequence of applying the pattern in an HPC environment results in (sub-)system designs that consist of a module that implements the primary design and a module that serves as an exceptional case handler, i.e., the recovery block. There is also an adjudicator that applies an acceptance test to validate the results produced by the primary system. If the adjudicator does not accept the results of the primary system, it invokes the exception handler subsystem, which indicates that the primary system could not perform the requested service operation. The protection domain of the pattern extends to the scope of the primary system, i.e., the scope for which the recovery block is created. Examples of the recovery block pattern in HPC include the Containment Domains (CD) [14] programming construct, which provides a recovery routine initiated upon detection of an error in the execution of the block of code encapsulated by the CD. This enables the CD to constrain the detection and correction of errors to the boundary of the domain.

3.4. State Patterns

3.4.1. Static State Pattern

The `Static` state pattern encapsulates all aspects of a system's state that is computed when the system is initialized, but is not modified during the system operation. The static state outlives the process that creates and initializes it. From the perspective of an HPC application, the static state includes program instructions and variable state that is computed upon application initialization. The correctness of the static state at all times is essential to the correct execution and outcome of a program. The invariant property of this state enables the use of a resilience behavioral pattern that can leverage this property to detect and recovery errors/failure of such state. For example, various algorithm-based fault tolerance methods leverage the property of invariance in the static state. These methods maintain replicas of the application variables in the static state pattern; recovery entails setting these variables to their default data values. A well-known application of this pattern is in the context of algorithm-based resilience techniques used in the design of iterative linear solvers. For the solution of a system of equations A.x = b, the static data structures such as the operand matrix A, the right-hand side vector B, or the preconditioner are computed once in the initialization phase of the application and are unchanged after. Errors in these structures are recovered using maintaining checksums [37].

3.4.2. Dynamic State Pattern

The `Dynamic State` pattern encapsulates the state that changes as the system makes forward progress. In an HPC application, the pattern refers to all aspects of the program state that changes as an application program executes. The dynamic state includes the data variables that are modified by the algorithm, as well as the control-flow variables that enable forward progress of the system. The *dynamic* feature of this state pattern implies that any faults or errors in such state amounts to lost work. Separating the dynamic state enables the identification of the appropriate behavioral resilience patterns to detect and correct errors in such state. Due to the transitory nature of the variables in the dynamic state patterns, the behavioral patterns often require preservation of the state pattern, or repetition of operations from a known stable point to recreate a version of the variables in the state pattern that are free from the effects of any errors. The most well-known method for protecting dynamic state is using checkpointing-based roll-back recovery methods [27].

3.4.3. Environment State Pattern

The `Environment State Pattern` encapsulates the system state that plays a supporting role in the operation of the system. The pattern defines the scope of the system state that provides a common set of services in support of the primary function of the system. The environment also facilitates and coordinates the operation of various sub-systems in a system. In general, HPC systems navigate complexity through the definition of abstractions that hide the details of specific functions behind well-defined interfaces. When executing an HPC application, the overall system state may be partitioned into the aspects that are related to the application program state and those that provide access to the system resources and services that enable the application to fulfill its function. The pattern enables the resilience behavior of the environment state to be reasoned about separately from the resilience behavior of the primary system state, i.e., an HPC application. The separation of the environment state enables designers to instantiate behavioral patterns that are independent of the design of the algorithms of HPC applications. Any changes in the environment due an error or failure event directly affects the application program operating within the environment. While an application program does not normally have complete control over its environment, it may exert partial control to affect the environment through well-defined interfaces. The `Environment` state pattern defines the scope of the state that support resource sharing, coordination and communication between the various (sub-)systems. In a typical HPC system stack, the environment state pattern includes productivity tools and libraries, the runtime system, the operating system, file systems, communication libraries, etc. For example, operating-system based resilience mechanisms are independent of the resilience features of the application program and solely focus on the correctness of the data structures within the kernel. Mini-Ckpts is a known example of a framework that emphasizes the recovery of the OS environment by preserving kernel structures in persistent memory [31]. Similarly, the ULFM MPI provides recovery of the communication environment from the failure of processes by reconstructing the MPI communicator by creating consensus among the remaining set of processes [7].

3.4.4. Stateless Pattern

The `Stateless` pattern enables the definition of resilience solutions that are independent of system state. Since every resilience solution consists of at least a state and behavior pattern, the `Stateless` pattern provides the construct of *null* state in order to create solutions that have a well-defined notion of behavior, but don't define a scope for the behavior. From the perspective of an HPC application, the definition of the `Stateless` pattern permits the definition of the scope of operations that perform detection or recovery without explicitly specifying the variable state of the program that is affected by the operations. The solutions that are based on a `Stateless` pattern may include: (i) applications that consist of predominantly memory load operations that rarely contain state-modifying memory and I/O operations; these applications typically perform reduction operations over large number of data elements, and (ii) applications that yield imperfect results since their algorithms are based on approximation and iterative refinement, or use noisy input data to begin with. The stateless pattern is utilized together with behavioral resilience patterns whose actions do not necessitate modifying any particular aspect of the system state during the detection or recovery. However, the resilience solution that uses a stateless pattern must select and instantiate a behavioral pattern that is capable of dealing with any additional side-effects due to the inclusion of the stateless pattern. The use of the *transaction* model to provide resilient behavior is an example of the `Stateless` pattern. Transactions support execution of a sequence of operations that may complete as a unit, or fail; the notion of partial execution is not supported. For example, in the Relax framework, the idempotence property guarantees that any region can be freely re-executed, even after partial execution, and still produce the same result. Relax supports language-level constructs as well as compiler-based techniques that enable the definition of idempotent regions of execution; the recovery of such regions is stateless [43].

4. Building Resilience Solutions Using Design Patterns

4.1. Components of Resilience Solutions

Each pattern in the resilience design pattern catalog presents a solution to a specific problem in detecting, containing or mitigating a fault, error or failure event. However, ensuring that an HPC application executes to result in a correct solution despite the occurrence of the events in the systems requires that a resilience solution be constructed using multiple such patterns that are organized in a well-defined system of patterns. The artifacts of a design process that uses design patterns are complete resilience solutions that confront a specific type of event and provide detection, containment and mitigation capabilities over a well-defined protection domain. Therefore, the first step in the design of a solution is the selection of patterns for each of these capabilities. Therefore, a complete solution consists of at least one state pattern (defining scope

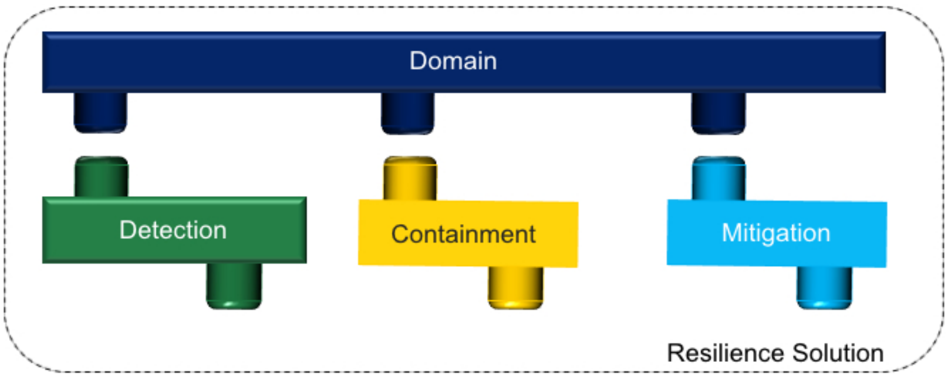


Figure 2. Elements of a resilience solution for HPC systems and applications

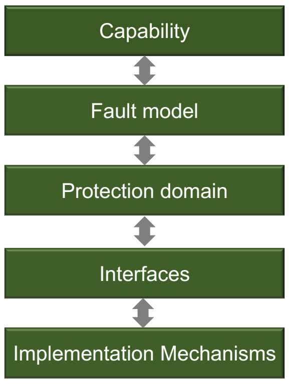


Figure 3. Design Spaces for construction of resilience solutions using patterns

of the protection domain) and one or more behavioral patterns (supporting a combination of detection, containment and mitigation solutions). These key constituents of a complete solution are shown in Figure 2.

The pattern descriptions allow for instantiating each pattern in the catalog at any layer of the system stack. The individual patterns that make up a complete solution can be implemented across layers the system stack. The architecture of a HPC system consists of various types of processor, memory, storage and networking components, and its software stack is a complex multicomponent environment consisting of communication and threading libraries, productivity software and tools, including numerical libraries, runtime systems, profiling tools, etc. To construct resilience solutions for the hardware and software components requires methodically selecting resilience patterns that may be conveniently incorporated into the design of these components. The coordination between the resilience patterns, particularly when implemented across layers of abstraction, requires well-defined activation and response interfaces for each pattern.

4.2. Design Spaces

For hardware and software designers to make practical use these patterns in the development of resilient versions of their designs, a set of guidelines are necessary to combine the patterns and refine their interactions. The hierarchical classification scheme articulates only certain aspects of the pattern selection and integration process by categorizing the patterns based on the type of event they handle and the core technique employed. However, the selection of patterns solely on the basis of their detection, containment and mitigation capabilities leaves much to skills of the designer in terms of finalizing the design and the implementation of the component or system. To build practical resilience solutions various other factors must be considered, including the layer of abstraction for their implementation, scalability of the solution, portability to other architectures, dependencies on any hardware/software features, flexibility to adapt the solution to accelerated fault rates, capability to handle other types of fault and error events, the performance and performance overheads.

To enable a systematic assessment of the suitability of a resilience pattern to a specific context and to integrate patterns into composite solutions, we develop a design framework. The framework enables the creation of an initial outline of the resilience solution that identifies the strategy patterns and the captures the dimensions and capabilities solution resulting from the composition of the patterns. The framework is based on *design spaces* that are arranged in a hierarchy. Each design space progressively refines the relationships between the patterns and optimizes the overall solution, which allows for a structured approach for constructing customized designs. By navigating over the design spaces, the framework enables the designer to approach the various issues that must be addressed in the process of developing practical resilience solutions. The framework, which is illustrated in Figure 3, consists of the following design spaces:

- **Capability**: This design space is concerned with identifying the patterns that support capabilities for the detection, containment, mitigation of a specific type of fault, errors or failure event. Based on the system context, this design space also considers the organization of the overall structure of the solution.
- **Fault model**: By identifying the root causes of fault and understanding the impact and propagation through the system enables deciding the architecture patterns. The design space emphasizes the selection of architecture patterns and the distribution of responsibility among the chosen patterns.
- **Protection domain**: This design space concentrates on the definition of the protection domain by deciding the state patterns and their composition. This enables a clear encapsulation of the system scope over which the resilience patterns operate.
- **Interfaces**: The identification and implementation of the activation and response interfaces for behavioral patterns affect the propagation of fault/error/failure event information. Within this design space, the layer of abstraction appropriate for the instantiation of the pattern, as well as the performance and power overheads are considered. The design space explores various implementation constructs that facilitate the coordination between the various patterns, particularly across system layers.
- **Implementation mechanisms**: This design space is concerned with low-level implementation details of how patterns are embedded within a hardware structure, or in software code. It considers the constraints imposed by specific features of hardware, execution or programming models, software environment and how the various pattern implementations coordinate their behavior in this context.

The design spaces represent the most important aspects of a resilience solution that a designer must contemplate in order to create effective and efficient resilience solutions. As a designer navigates through these design spaces, they are able to develop a clearer understanding of the solution profile and the general constraints, which enables them to select the appropriate patterns from the catalog and decide on implementation alternatives. The use of resilience patterns in the context of the framework provided by the design spaces enables HPC system designers, users and application developers to evaluate the feasibility and effectiveness of novel resilience techniques, as well as analyze and evaluate existing solutions. They provide a structured flow to the design process the design spaces articulate the critical decision points in the design of a resilience solution, providing guidelines for the selection of the appropriate patterns based on the requirements of protection and the cost of using specific patterns.

Designers may use various approaches to navigate the design spaces, including a strictly top-down approach, in which the design is driven by the event type and model that a system must be protected against, and the implementation of the system is adapted to enable the system to survive the different ways in which the event may impact the reliability of the system. Alternatively, in a bottom-up approach, the resilience capability must be woven into the existing hardware or software component designs and interfaces, and additional components are included to enhance the protection coverage, or to handle specific fault model behaviors. Often, designers may be required to take a hybrid approach, in which the design spaces are revisited in an effort to refine a design, to optimize the features of a solution, and to enable designers to overcome constraints imposed by any hardware or software system features.

5. Case Studies

This section explores use cases for the application of resilience design patterns to the systematic design and analysis of resilience solutions. We use the pattern-based approach for understanding existing solutions with the view to adapt the solution to future generations of HPC systems as well as for exploration and assessment of novel cross-layered solutions. The case studies describe the pattern-based design process for different fault models on a notional architecture and software environment of a HPC system.

5.1. Checkpoint and Rollback Solution for Process Failures

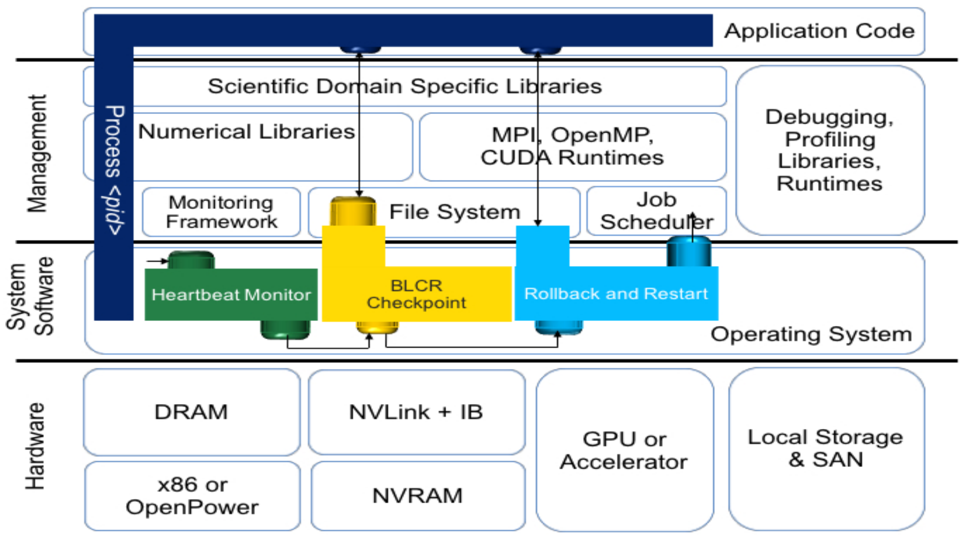


Figure 4. Case Study: Checkpoint & Restart-based Recovery

For this case study, we aim to develop a resilience solution that enables an HPC application to survive process failures. In an HPC environment, the diagnosis of the precise root cause of these failures is difficult due to the lack of sufficient hardware-level debugging information. For designing a purely software-based solution, the fault model is a process crash or hang whose cause is unknown. This type of failure results from the presence of a fault in the processor or memory that activates, which causes an error in the form of an illegal instruction, or an invalid address in the program state. When the program execution encounters the address in the program state that is in error state, the process may crash or hang.

Checkpoint and restart (C/R) solutions are the often used to support resilience to process failures in HPC systems. We reexamine this well-known software-based solution using the structured pattern-based approach to analyze composition of the constituent patterns needed to design this solution. Such analysis will be useful for adapting C/R solutions to future systems and evaluate their performance characteristics. The goal of a complete C/R solution is to recover a failed process such that the application may resume from an error-free state. This requires that the solution capture the image, or snapshot, of a running process and preserves it for later recovery. For parallel applications, the C/R framework's coordination protocols produce a global snapshot of the application by combining the state of all the processes in the parallel application. Since most parallel applications using the message passing interface (MPI) define a MPI process to be a POSIX process, the protection domain of the solution must cover the complete POSIX process state. Therefore, we fuse the `Persistent` and `Dynamic` and `Environment` state patterns, which extends the domain of our system-level checkpointing solution to the entire memory associated with a process. In a Linux-based environment, the protection domain covers the total virtual address space of a Linux process.

For the detection of a process failure, we require instantiation of the `Fault Treatment` strategy pattern. Specifically, our solution requires a `Fault Diagnosis` architecture pattern to discover the location of the failure and the type of event, which is enabled by a `Monitoring` structural pattern. The instantiation of the `Monitoring` pattern is a kernel-level heartbeat monitor, which is deployed in the system to detect whether the process is alive.

For the selection of a recovery pattern, there are key two considerations: (i) the frequency of node failures; and (ii) the performance and resource overhead of applying the pattern. The space overhead incurred by instantiating a `Compensation` strategy pattern for recovery is substantial due to the need to replicate the protection domain. For systems that experience process failures infrequently, the use of a compensation-based solution proves prohibitively expensive. Therefore, for the failure recovery we select the `Recovery` strategy pattern. The `Checkpoint-Recovery` architectural pattern is appropriate since Linux provides the capability for a running process to be interrupted and its context to be written to disk. Also, the process state is deterministic and defined by the state of the program counter and the registers; therefore, the `Roll-back` structure pattern is suitable for implementation at the operating system level. With the selection of this pattern protection domain of the failure to be limited to a single process context, which implicitly defines the containment pattern. The implementation of the recovery pattern requires a disk storage system, to which the checkpoint, i.e., the process state captured during failure-free operation is exported. The performance overhead of these patterns during failure-free operation and the recovery time are dependent on bandwidth available between memory and the disk system.

The implementation of the patterns, which is illustrated in Figure 4, is implemented using the Berkley Lab's Checkpoint/Restart (BLCR) [25] framework. Since BLCR does not provide a failure detection mechanism, the `Monitoring` pattern is implemented by a kernel-level module that uses heartbeat monitoring to check for process liveness. BLCR provides a completely transparent checkpoint of the process, which saves the current state of a Linux process. The framework uses a coarse-grain locking mechanism to momentarily interrupt the execution of all the threads

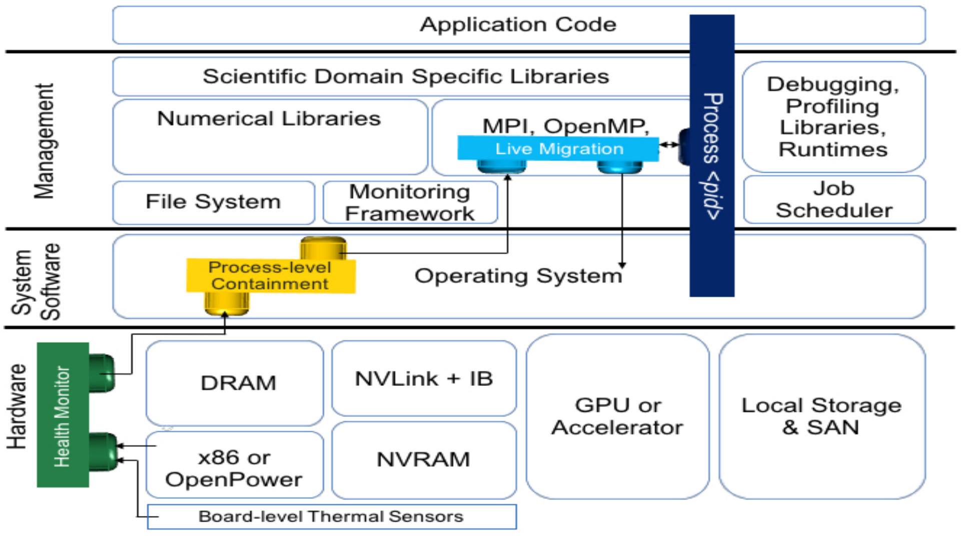


Figure 5. Case Study: Proactive Process Migration

of the process, giving them a global view of its current state. The entire state is saved, including the CPU registers, the virtual memory map as well as the function call stack. From the perspective of an application programmer, the checkpoint routine returns with a different error code, to let the caller know if this function call returns from a successful checkpoint or from a successful restart. The `Roll-back` pattern handles recovery after the detection of a process failure by restoring the context file set from the stable storage, and recreating the process on the same hardware, with the same software environment. BLCR also provides an API for applications programmers to manage pattern behavior through hooks that allow the application to block off code sections where checkpoints are not permitted. These hooks also give applications a chance to respond to checkpoint/requests and take appropriate action, which provides an application programmer with explicit control over the pattern's activation and response interfaces.

5.2. Proactive Process Migration for Failure Avoidance

In HPC environments, various fault indicators indicate the imminence of error or failure events. The goal of this case study is to design and implement a proactive resilience solution using the structured design pattern-based approach. In contrast to a reactive solution that seeks to recover from an error or a failure event after the fact, a proactive solution identifies faults in a system and seeks to remedy the anomaly or defect to prevent their activation to result in errors or failures. This analysis of this solution is intended to identify the patterns that must instantiated for a proactive design approach, and to articulate the protection domain of the solution.

The key to designing a proactive strategy is the identification of fault indicators that can sufficiently predict the activation of an error or failure. The fault model for this case study is a defect in the system that has the potential to result in an error or failure. We consider faults that are known to cause errors, which result in application crashes. Using design patterns, we seek to develop a software-based solution that can preemptively migrate parts of an application away from system resources that are about to fail. In a HPC system, the failure of a compute node causes termination of the application processes running on that node. Since the presence of a fault does not impact the correctness of an application program until it activates, the solution supports proactive failure avoidance from the application's perspective. We select the protection domain by fusing the `Persistent` and `Dynamic` and `Environment` state patterns. Much like the C/R solution, the protection domain covered by these patterns includes the complete POSIX process state in a Linux environment. The ultimate objective of the solution is to preemptively migrate the application processes from compute nodes where a failure is likely to cause them to crash to another node in the system.

To anticipate the occurrence of a failure, the solution must observe critical indicators that will predict the likelihood of a failure. We apply the `Fault Treatment` strategy pattern, which is instantiated as a `Fault Diagnosis` pattern in every node of the HPC system. This pattern is instantiated as a `Prediction` structural pattern, which enables estimating the possibility of an imminent error or failure event. Its activation interface reads health monitoring data for the various components in each compute node and its response interface signals the possibility of a node failure. The prediction pattern creates a control feedback-loop such that a mitigation pattern can take preventive action to avoid failure of the processes running on the node. Since the solution addresses faults in the computes nodes, it requires the instantiation of another `Fault Treatment` pattern for mitigation rather than a `Recovery` strategy pattern. For this solution, we assume that the number of nodes allocated for an application run are determined during startup and are fixed for the lifetime of the application run. If the application uses all nodes in the allocation at initialization and leaves no spare nodes, the inclusion of a `Compensation` strategy pattern is not a suitable alternative. The `Reconfiguration` architectural pattern is applied, which is instantiated in the form of a `Restructure` structural pattern that isolates a failing node and migrates the application processes to an alternative compute node in the system. The containment is implemented by a kernel level module provides containment for the fault by identifying the process that is executing on the node which the `Prediction` pattern has assessed vulnerable due to a specific set of changes in operating conditions of the node.

The overall structure of the pattern-based design is illustrated in Figure 5. The implementation of the `Prediction` pattern is realized as a per-node health monitoring mechanism that uses various platform-level indicators in the system. It uses platform data available through the Intelligent Platform Management Interface (IPMI) interface, which relies on the baseboard management controller (BMC) to collect sensors readings for health monitoring, including the data on temperature, fan speed, and voltage. The response interface of the pattern notifies the scheduler when the sensors indicate deterioration of a node's health. Since the behavior of the `Recovery` strategy pattern used by this solution entails performing a live migration of a POSIX process in the context of the MPI execution environment, the implementation of the `Restructure` pattern is realized within the system's job scheduler. The pattern identifies healthy nodes in the system as potential destinations for the process migration. Once a destination node has been identified, the pattern initiates the migration of the process from source to destination node. It is imperative the entire context of a process be migrated when the presence of a fault is inferred on a compute node. Therefore, the migration entails transfer of the process image, which occurs by a page-by-page copy of the address space. The implementation then synchronizes all the MPI processes to a consistent state, after which the in-flight data in the MPI communication channels is drained. Once all the MPI processes reach a consistent global state, the remaining dirty pages, which includes the registers, signal information, pid, files, etc. to the destination node. Once the mapping of the processes to nodes in the system has been restructured, the communication channels and the previously saved in-flight messages are restored. The migrated

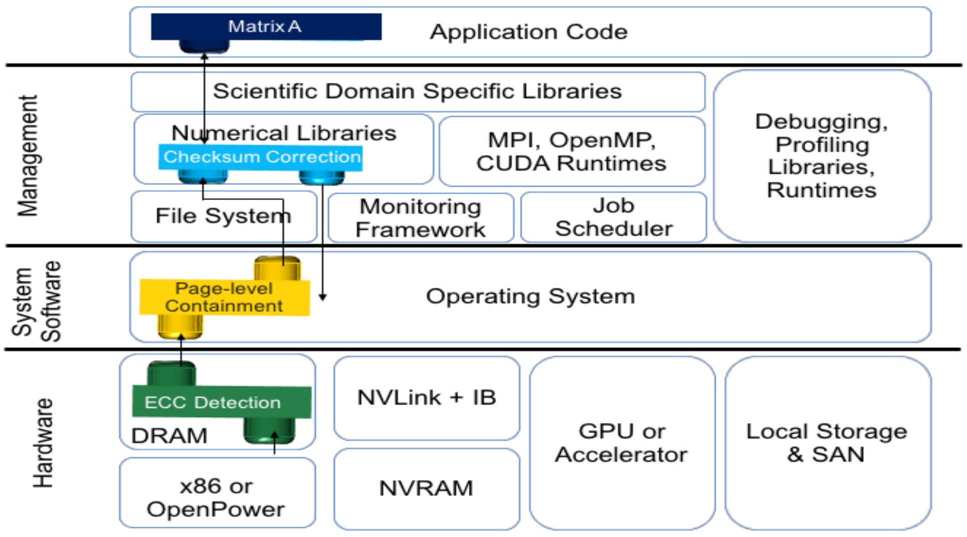


Figure 6. Case Study: Cross-Layer Design using Algorithm-based Fault Tolerance to complement Hardware-level Error Correction Codes

processes resume execution on the destination node. The implementation of the patterns in this solution ensure the transparency of the proactive migration to the HPC application.

5.3. Cross-layer Hardware/Software Solution for Soft Error Resilience

In this case study, we use design patterns as building blocks to explore novel resilience solutions that leverage capabilities from various layers of the system stack. By navigating the design spaces of the resilience design pattern framework, we can evaluate the effectiveness of instantiating a detection, containment or mitigation pattern at a specific level in the system stack and systematically construct a cross-layer resilience solution that connects patterns from multiple layers. The structured approach supported by the framework also enables refining the cross-layered solution. The aim of this case study is to develop a solution that provides soft error detection and correction for HPC application data structures. The fault model that we consider is transient errors in memory structures that cause multiple bit flips in the application's data or control variables, which may result in outcomes ranging from incorrect results to fatal program crashes.

The DRAM memory chips used in HPC systems use error correcting codes (ECC) to detect and correct bit flip errors. Similarly, algorithm-based fault tolerance techniques are available that maintain checksums for data structures to detect and correct data value errors at the application level. However, the lack of formal methods to combine these solutions often precludes cross-layer hardware-software designs that cooperative protect the application data. Our proposed solution is designed to support transient error resilience for a scientific application that uses an iterative linear solver method. In general, these methods solve a system of linear equations represented as $A.x = B$, where x is the solution vector, A is the operand matrix and b is a known vector. The iterative algorithm begins with an initial approximation of the solution x, and refines this solution until the residual norm is below a certain error bound. Therefore, the matrix A and vector b are scoped within `Static` state patterns, the solution vector x in a `Dynamic` state pattern, and the remaining variable state is contained within an `Environment` pattern. While the solution vector is often tolerant to perturbations due to the iterative nature of the algorithm, any transient errors within the scope of the two `Static` state patterns affects the correctness of the solver. Therefore, we define the protection domain of our cross-layer solution to include only these static patterns.

For achieving error detection and correction in digital data, the general approach is to add redundant information to discover errors and reconstruct the original data. This approach fits the `Compensation` strategy pattern, which may be instantiated in the form of a `Forward Error Correction` pattern. For the detection of the transient errors, we assume that this pattern is implemented in the form of ECC in the DRAM modules, which supports single-bit error correction and double-bit error detection. Therefore, the instantiation of this structural pattern handles both detection and mitigation for single-bit errors. Double-bit errors result in an ECC violation on the memory line, which is asynchronously communicated by the `Forward Error Correction` pattern to the operating system via its response interface by raising a machine check exception. For the containment of the double-bit error, we deploy a `Fault Treatment` pattern in the operating system, since the OS views the double-bit corruption as a fault. Since the pattern must discover whether the double-bit corruption maps to the protection domain specified by the state patterns, it is instantiated as a `Fault Diagnosis` pattern, specifically as a `Monitoring` structural pattern. For recovery of variable state scoped by the `Static` state pattern, the solution instantiates the `Compensation` strategy pattern. It uses the `Redundancy` architecture pattern and structures the solution based on the `Forward Error Correction` pattern.

The instantiation of the patterns across the system stack is illustrated in Figure 6. The `Monitoring` pattern for containment is implemented as a kernel-level module that maps the physical address to the virtual address space to discover whether the fault may be contained within the `Static` state pattern. The pattern's response interface treats the presence of the fault in the state pattern as an application error and notifies the numerical library. When the error is outside the scope of the `Static` state pattern, the response interfaces indicates to the kernel module that the error is unrecoverable, which results in the OS killing the application. Besides the `Forward Error Correction` pattern in ECC for single-bit error recovery, another instance of this pattern type is implemented in the numerical library to handle double-bit errors. The implementation maintains a set of checksums for the matrix A and vector b. The checksums enable the identification of the element of the matrix affected by the error, and substitution of that element with a correct value using the remaining uncorrupted elements in the row/column and the checksum values. The instantiation of the `Forward Error Correction` pattern at the application library level provides context about the significance of the error to the overall application, and is able to employ an algorithm-specific fault tolerance detection and correction method, which is more cost effective for double-bit error mitigation than system-level bulk reliability provided by hardware-level solution such as an enhanced ECC that supports double-bit correction. Therefore, the cooperation between patterns across system layers supports a flexible memory protection mechanism to single and double-bit memory errors, which allows the application to resume operation towards completion rather than experience a fatal crash with higher performance and energy efficiency.

6. Related Work

The original concept of design patterns was developed in the context of civil architecture and engineering problems where patterns were defined with the goal of identifying and cataloging solutions to recurrent problems and solutions in the building and planning of neighborhoods, towns and cities, as well as in the construction of individual rooms and buildings [4]. In the domain of software engineering, patterns were introduced in an effort to bring discipline to the art of programming and create reusable designs. The intent of software design patterns isn't to provide a finished design that may be transformed directly into code; rather, these patterns are used to systematize the software development process by using proven paradigms and methodologies in software engineering practice [12]. With the use of design patterns, there is sufficient flexibility for software developers to adapt their implementation to accommodate any constraints, or issues that may be unique to specific programming paradigms, or the target platform for the software. Related to software design patterns, the concept of algorithmic skeletons was introduced [16] and further refined [17]. In the context of object-oriented (OO) programming, design patterns provide a catalog of methods for defining class interfaces and inheritance hierarchies, and establish key relationships among the classes [34]. In many OO systems, reusable patterns of class relationships and interactions between objects are used to create flexible, elegant, and ultimately reusable software design. Pattern systems have also been developed for cataloging concurrent and networked object-oriented environments [54], resource management [41], and distributed systems [11].

In the pursuit of quality and scalable parallel software, patterns for programming paradigms were developed [45] as well as a pattern language, called Our Pattern Language (OPL) [40]. These describe the computation and communication patterns in various parallel algorithms and therefore useful for designing and implementing scalable parallel applications. For engineering parallel applications for shared-memory many-core processors, parallel programming patterns simplify the process of expressing parallelism using a number of programming interfaces such as OpenMP, OpenCL, Cilk Plus, ArBB, Thread Building Blocks (TBB) [46]. Patterns also support the implementation of parallel algorithms that automatically avoid unsafe race conditions and deadlocks [47].

Design patterns have been discovered in a variety of other domains and used to codify the best-known solutions, which include patterns for natural language processing [57], user interface design [8], web design [26], visualization [36], and software security [23]. Patterns have also been defined for enterprise applications that involve data processing in support or automation of business processes [32] in order to bring structure to the construction of enterprise application architectures. In each of these domains of design, the patterns capture the essence of solutions in a succinct form such that they may be easily applied to other contexts.

Previous efforts to develop design patterns for fault tolerance have defined a number of patterns for error detection, recovery and mitigation. These patterns are developed based on well-known fault tolerance solutions that are used in mission-critical systems such as telecommunication systems and space programs [35], distributed systems [52] and enterprise data processing systems [33]. The fault tolerant version of the Common Object Request Broker Architecture (CORBA) [50] applies patterns in the design of the middleware to improve the performance of a range of fault tolerance strategies that provide applications with capabilities for rapid recovery from service failures, including request-retry, redirection, active and passive replication. While the capabilities of some of the patterns in these domains overlap with the resilience patterns described in this document, they solve problems that are significantly different from those encountered in HPC environments in terms of the system architectures, the software stack, and the nature of the applications. The patterns in this document specifically address the challenges for maintaining resilient operation for HPC systems and their applications.

Summary

In this paper, we introduce the concept of resilience design patterns, which support a systematic approach to designing and implementing resilience solutions. The structured approach to the design of HPC resilience solutions is useful to reduce the complexity of the design process, and is particularly relevant for the future generations of extreme-scale parallel systems and their applications. The resilience design patterns are based on well-known and well-understood solutions that have been applied in HPC systems and provide solutions to specific problems encountered in the management of resilience. The patterns presented in this document support detection, containment, masking and recovery capabilities. The resilience patterns may be used by designers as reusable templates when building and refining new resilience solutions and for reengineering existing solutions for future generations of HPC systems. The paper also presents a classification scheme that organizes the resilience patterns in a layered hierarchy in order to expose the relationships between the various patterns in the catalog and their capabilities. The hierarchical organization of the patterns enables system hardware and software architects to approach the solution at an abstract level, while individual component designers and software developers may restrict their work to the level that directly impacts their portion of the solution. We have also developed a design framework to simplify the composition of design patterns into complete resilience solutions. The framework is useful for navigating the various design challenges and constraints encountered by designers and enables the creation of flexible and portable resilience solutions. The resilience patterns and the pattern-oriented framework also facilitates the exploration of alternative solutions, the refinement and optimization of solutions, and the investigation of the effectiveness and efficiency of solutions. This structured approach aims to address the resilience challenge for extreme-scale HPC systems through a systematic design of solutions with an emphasis on optimizing the trade-off, at design time or runtime, between the key system design factors: performance, resilience, and power consumption.

Acknowledgments

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

1. Nagios monitoring system (1999), `https://www.nagios.org/`, accessed (2017-08-15)

2. Lustre file system, high-performance storage architecture and scalable cluster file system,

white paper. Tech. rep., Sun Microsystems, Inc. (December 2007)

3. Agelastos, A., Allan, B., Brandt, J., Cassella, P., Enos, J., Fullop, J., Gentile, A., Monk, S., Naksinehaboon, N., Ogden, J., Rajan, M., Showerman, M., Stevenson, J., Taerat, N., Tucker, T.: Lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In: Proceedings of IEEE/ACM International Conference for High Performance Storage, Networking, and Analysis (SC14). IEEE/ACM (2014), DOI: 10.1109/sc.2014.18

4. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York (August 1977)

5. Avižienis, A.: Toward systematic design of fault-tolerant systems. Computer 30(4), 51–58 (April 1997), DOI: 10.1109/2.585154

6. Batchu, R., Dandass, Y.S., Skjellum, A., Beddhu, M.: Mpi/ft: A model-based approach to low-overhead fault tolerant message-passing middleware. Cluster Computing 7(4), 303–315 (2004), DOI: 10.1023/b:clus.0000039491.64560.8a

7. Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: Post-failure recovery of mpi communication capability: Design and rationale. International Journal of High Performance Computing Applications 27(3), 244–254 (2013), DOI: 10.1177/1094342013488238

8. Borchers, J.: A Pattern Approach to Interaction Design. John Wiley & Sons, Inc., New York, NY, USA (2001)

9. Borkar, S.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro 25(6), 10–16 (November 2005), DOI: 10.1109/mm.2005.110

10. Bouteiller, A., Bosilca, G., Dongarra, J.: Redesigning the message logging model for high performance. Concurrency and Computation: Practice and Experience 22(16), 2196–2211 (2010), DOI: 10.1002/cpe.1589

11. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture - Volume 4: A Pattern Language for Distributed Computing. Wiley Publishing (2007)

12. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture - Volume 1: A System of Patterns. Wiley Publishing (1996)

13. Chien, A., Balaji, P., Dun, N., Fang, A., Fujita, H., Iskra, K., Rubenstein, Z., Zheng, Z., Hammond, J., Laguna, I., Richards, D., Dubey, A., van Straalen, B., Hoemmen, M., Heroux, M., Teranishi, K., Siegel, A.: Exploring versioned distributed arrays for resilience in scientific applications: global view resilience. The International Journal of High Performance Computing Applications (2016), DOI: 10.1177/1094342016664796

14. Chung, J., Lee, I., Sullivan, M., Ryoo, J.H., Kim, D.W., Yoon, D.H., Kaplan, L., Erez, M.: Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 58:1–58:11 (2012)

15. Clustering, P.H.A.: Pvfs2 development team (June 2004), `https://goo.gl/VRAahX`, accessed (2017-08-15)

16. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge, MA, USA (1991)

17. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Computing 30(3), 389–406 (Mar 2004), DOI: 10.1016/j.parco.2003.12.002

18. Cray Inc.: Cray xe6 computing platform (2010), `http://www.cray.com/sites/default/files/resources/CrayXE6Brochure.pdf`

19. Cray Inc.: Cray xc40 computing platform (2014), `http://www.cray.com/Assets/PDF/products/xc/CrayXC40Brochure.pdf`

20. van Dam, Hubertus, J.J., Vishnu, A., De Jong, W.A.: A case for soft error detection and correction in computational chemistry. Journal of Chemical Theory and Computation 9(9), 3995–4005 (2013), DOI: 10.1021/ct400489c

21. Dell, I.H.P.: Intelligent platform management interface (ipmi), v2.0 specification (2015), DOI: 10.2172/1104721

22. Dongarra, J., Beckman, P., Moore, T., et al.: The International Exascale Software Project Roadmap. International Journal on High Performance Computing Applications pp. 3–60 (February 2011)

23. Dougherty, C., Sayre, K., Seacord, R., Svoboda, D., Togashi, K.: Secure design patterns. Tech. Rep. CMU/SEI-2009-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2009), DOI: 10.21236/ada636498

24. Dreslinski, R.G., Wieckowski, M., Blaauw, D., Sylvester, D., Mudge, T.: Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. Proceedings of the IEEE 98(2), 253–266 (February 2010), DOI: 10.1109/jproc.2009.2034764

25. Duell, J., Hargrove, P., Roman, E.: The design and implementation of berkeley lab's linux checkpoint/restart. Tech. rep., Lawrence Berkeley National Lab (LBNL) (December 2002), DOI: 10.2172/891617

26. Duyne, D.K.V., Landay, J., Hong, J.I.: The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

27. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34(3), 375–408 (Sep 2002)

28. Engelmann, C., Böhm, S.: Redundant execution of HPC applications with MR-MPI. In: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN). pp. 31–38 (February 2011), DOI: 10.2316/p.2011.719-031

29. Ferreira, K., Riesen, R., Oldfield, R., Stearley, J., Laros, J., Pedretti, K., Brightwell, R.: Rmpi: increasing fault resiliency in a message-passing environment. Tech. rep., Sandia National Laboratories, Technical Report SAND2011-2488 (2011), DOI: 10.2172/1012733

30. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R.: 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. pp. 78:1–78:12. SC '12 (2012), DOI: 10.1109/sc.2012.49

31. Fiala, D., Mueller, F., Ferreira, K., Engelmann, C.: Mini-ckpts: Surviving os failures in persistent memory. In: Proceedings of the 2016 International Conference on Supercomputing. pp. 7:1–7:14. ICS '16 (2016), DOI: 10.1145/2925426.2926295

32. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

33. Friedrichsen, U.: No crash allowed - patterns for fault tolerance. In: The Conference for Java and Software Innovation (October 2012)

34. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995), DOI: 10.1007/978-3-642-59412-0_40

35. Hanmer, R.: Patterns for Fault Tolerant Software. Wiley Publishing (2007)

36. Heer, J., Agrawala, M.: Software design patterns for information visualization. IEEE Transactions on Visualization and Computer Graphics 12(5), 853–860 (Sep 2006), DOI: 10.1109/tvcg.2006.178

37. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers C-33(6), 518–528 (June 1984), DOI: 10.1109/tc.1984.1676475

38. Hukerikar, S., Engelmann, C.: Resilience design patterns: A structured approach to resilience at extreme scale (version 1.1). Tech. Rep. ORNL/TM-2016/767, Oak Ridge National Laboratory, Oak Ridge, TN, USA (December 2016), `http://www.christian-engelmann.info/publications/hukerikar16rdp-11.pdf`, DOI: 10.2172/1345793

39. Hursey, J., Mattox, T.I., Lumsdaine, A.: Interconnect agnostic checkpoint/restart in open mpi. In: HPDC '09: Proceedings of the 18th ACM international symposium on High Performance Distributed Computing. pp. 49–58. ACM, New York, NY, USA (2009), DOI: 10.1145/1551609.1551619

40. Keutzer, K., Mattson, T.: Our pattern language (opl): A design pattern language for engineering (parallel) software. In: ParaPLoP Workshop on Parallel Programming Patterns (2009), DOI: 10.1109/wicsa.2007.32

41. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management. John Wiley & Sons, Inc., New York, NY, USA (2004)

42. Koren, I., Su, S.Y.H.: Reliability analysis of n-modular redundancy systems with intermittent and permanent faults. IEEE Transactions on Computers 28(7), 514–520 (July 1979), DOI: 10.1109/tc.1979.1675397

43. de Kruijf, M., Nomura, S., Sankaralingam, K.: Relax: an architectural framework for software recovery of hardware faults. In: Proceedings of the 37th annual international symposium on Computer architecture. pp. 497–508. ISCA '10 (2010), DOI: 10.1145/1815961.1816026

44. Massie, M.L., Chun, B.N., Culler, D.E.: The ganglia distributed monitoring system: design, implementation, and experience. Parallel Computing 30(7), 817 – 840 (2004), DOI: 10.1016/j.parco.2004.04.001

45. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)

46. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2012), DOI: 10.1016/b978-0-12-415993-8.00003-7

47. McCool, M.D.: Structured parallel programming with deterministic patterns. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism. pp. 5–5. HotPar'10, USENIX Association, Berkeley, CA, USA (2010)

48. Mohror, K., Moody, A., Bronevetsky, G., de Supinski, B.R.: Detailed modeling and evaluation of a scalable multilevel checkpointing system. IEEE Transactions on Parallel and Distributed Systems 99, 1 (2013), DOI: 10.1109/tpds.2013.100

49. Moon, T.K.: Error correction coding: Mathematical methods and algorithms (2005)

50. Natarajan, B., Gokhale, A., Yajnik, S., Schmidt, D.C.: Doors: towards high-performance fault tolerant corba. In: Proceedings of the International Symposium on Distributed Objects and Applications. pp. 39–48 (2000), DOI: 10.1109/doa.2000.874174

51. Sahoo, R.K., Oliner, A.J., Rish, I., Gupta, M., Moreira, J.E., Ma, S., Vilalta, R., Sivasubramaniam, A.: Critical event prediction for proactive management in large-scale computer clusters. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 426–435. KDD '03, ACM, New York, NY, USA (2003), DOI: 10.1145/956790.956799

52. Saridakis, T.: A system of patterns for fault tolerance. In: Proceedings of 2002 European Conference on Pattern Languages of Programs (EuroPLoP) (2002)

53. SchedMD: Slurm workload manager (2003), `https://slurm.schedmd.com/`

54. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. John Wiley & Sons, Inc., New York, NY, USA, 2nd edn. (2000)

55. Shalf, J., Quinlan, D., Janssen, C.: Rethinking hardware-software codesign for exascale systems. Computer 44(11), 22–30 (November 2011), DOI: 10.1109/mc.2011.300

56. Stellner, G.: Cocheck: checkpointing and process migration for mpi. In: Proceedings of International Conference on Parallel Processing. pp. 526–531 (Apr 1996), DOI: 10.1109/ipps.1996.508106

57. Talton, J., Yang, L., Kumar, R., Lim, M., Goodman, N., Měch, R.: Learning design patterns with bayesian grammar induction. In: Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology. pp. 63–74. UIST '12, ACM, New York, NY, USA (2012), DOI: 10.1145/2380116.2380127

DOI: 10.14529/jsfi170302

Performance Evaluation of Runtime Data Exploration Framework based on In-Situ Particle Based Volume Rendering

Takuma Kawamura[1], *Tomoyuki Noda*[1], *Yasuhiro Idomura*[1]

We examine the performance of the in-situ data exploration framework based on the in-situ Particle Based Volume Rendering (In-Situ PBVR) on the latest many-core platform. In-Situ PBVR converts extreme scale volume data into small rendering primitive particle data via parallel Monte-Carlo sampling without costly visibility ordering. This feature avoids severe bottlenecks such as limited memory size per node and significant performance gap between computation and inter-node communication. In addition, remote in-situ data exploration is enabled by asynchronous file-based control sequences, which transfer the small particle data to client PCs, generate view-independent volume rendering images on client PCs, and change visualization parameters at runtime. In-Situ PBVR shows excellent strong scaling with low memory usage up to $\sim$ 100k cores on the Oakforest-PACS, which consists of 8,208 Intel Xeon Phi7250 (Knights Landing) processors. This performance is compatible with the remote in-situ data exploration capability.

Keywords: in-situ visualization, volume rendering, runtime steering, strong scaling, performance evaluation.

Introduction

Recent advances in HPC technology enabled extreme scale simulations at several tens of Peta-FLOPS, while the performance gap between computation and data I/O is enhanced. Because of this severe I/O bottleneck, data handling procedures such as “data output from simulations to storage” and “data input from storage to visualization applications” are becoming very costly, and conventional post processing visualization strategies often fail. In-situ visualization is one of promising solutions to this issue. In this approach, the I/O bottleneck is avoided by combining simulations and visualization applications to visualize simulation data at runtime on the same computing nodes. This requires extreme scale parallel visualization with high scalability at the same level as the simulations.

Computational fluid dynamics (CFD) applications are widely used in various science and engineering fields, and are expected to be one of major applications on future exa-scale systems. Although variety of scientific visualization methods have been developed for CFD applications, visualization methods applicable to massively parallel processing of extreme scale volume data are still limited. Therefore, volume rendering methods for in-situ approaches should be carefully selected from the viewpoint of massively parallel processing and flexible visual analytics. Volume rendering is one of scalable visualization methods, which are suitable especially for CFD applications. In addition, in Ref. [4], it was shown that the volume rendering makes visual analytics flexible by extending the definition of transfer functions (TFs) into multi-dimension. Multi-dimensional transfer functions (MDTFs) generate not only three dimensional (3D) volume rendering, but also iso-surfaces, slice planes, image cropping, and image composition.

Another important requirement is interactivity of in-situ visualization. In the conventional in-situ visualization, visualization parameters such as viewpoint and TFs are prescribed before batch processing. But, such prescribed parameters often generate undesirable images. This problem obviously leads to the loss of computational resources, and is fatal at extreme scale. In order to extract important features from extreme scale data, one needs to change visualization

[1]Japan Atomic Energy Agency, Kashiwa, Chiba, Japan

parameters repeatedly in an interactive manner. To this end, the so-called runtime visualization approaches [18] [19] are becoming more important in modern visual analytics. In the runtime visualization approaches, runtime steering of visualization parameters enables interactive in-situ data exploration, which minimizes such a visualization failure.

However, it is not so clear whether one can design such interactive in-situ visualization frameworks based on the conventional volume rendering algorithms, which may suffer from the following issues. Firstly, the conventional algorithms of volume rendering require large rendering primitive data (e.g., splatting kernels, cells, and polygons), which can be comparable or even larger than the original volume data. Since exa-scale simulations have to overcome severe constraint on the memory size per node, this feature is a critical issue. Secondly, even if one can store such large rendering primitive data, calculation of semi-transparency attribute requires costly collective communication for sorting or searching of sub-images, which could lead to performance degradation and prevent strong scaling. Thirdly, the conventional volume rendering generates view-dependent images, and thus, interactive in-situ visualization becomes extremely costly. Existing parallel visualization libraries such as VTK-m [12], VisIt [1], and ParaView [2] support in-situ visualization based on the conventional volume rendering algorithms. However, the above bottlenecks were not resolved yet. Therefore, it is important to construct in-situ visualization frameworks based on a new volume rendering algorithm, which can resolve the above issues, and demonstrate its performance on the latest many-core platforms.

In this work, we address these issues by using the In-Situ PBVR [6], which was developed using visualization algorithm PBVR [17] [15], whose rendering primitive can be processed view-independent and data size is controllable by image quality. This framework converts extreme scale volume data into small view-independent rendering primitive (particle) data via Monte-Carlo sampling, and transfers it to client PCs, where it is rendered at interactive frame rate. Since the particle generation process does not require visibility ordering, and thus, inter-node communication, this framework works only with the embarrassingly parallel Monte-Carlo sampling, which can be computed on the same massively parallel nodes as the simulations with much less memory usage. Therefore, it is expected that this framework does not suffer from the aforementioned bottlenecks. In addition, the In-Situ PBVR supports flexible MDTF design, which is essential for visualizing complicated multivariate volume data generated from extreme scale simulations with high fidelity and reality. We examine the performance of the In-Situ PBVR, which is coupled to a multi-phase multi-component thermal-hydraulic CFD code, on the Oakforest-PACS, which consists of 8,208 Intel Xeon Phi7250 (Knights Landing) processors.

1. Related Works

Volume rendering requires visibility ordering for alpha blending over the entire volume data, which leads to high calculation and memory costs. In realizing interactive visual analytics, the development of parallel volume rendering with high frame rate and low memory cost is of critical importance. So far, such parallel volume rendering methods have been developed on various parallel platforms, and they are categorized into two types: methods optimized for multi-threaded accelerators such as Xeon Phi and GPU, and methods optimized on massively parallel CPU platforms. Since massively parallel accelerator platforms are one of promising approaches towards exa-scale machines, the former is attracting much attention in recent works.

On massively parallel platforms, most parallel volume rendering methods adopts the so-called sort-last approach, in order to efficiently use distributed memory. While this approach is suitable for processing subdivided volume data on distributed memory, visibility ordering for image composition requires costly inter-node communication. To resolve this issue, significant efforts were made in former works. By optimizing time series processing pipeline and reducing the cost of image composition using improved direct send method, Peterka et al. [13] improved the performance of parallel volume rendering using 4,096 cores of IBM BlueGene/P and processed about 644 million grids (864^3) of volume data at the frame rate of $\sim$ 0.3 frame per second (fps). However, in this work, the cost of image composition rose exponentially, and exceeded the cost of rendering at 4,096 cores. Perterka et al. further optimized this approach using a faster compositing algorithm Radix-k and extended the numerical experiment up to 32k cores [14]. In the case of about 1.4 billion grids (1120^3), the performance acceleration was saturated at 16k cores, while in the case of about 90 billion grids (4480^3), the scalability was extended up to 32k cores. Howison et al. [3] optimized the volume rendering using Levoy's method [11] with a hybrid MPI/OpenMP parallelization model on Cray XT5. They conducted scaling tests using 98 billion grids (4608^3), and the performance was scaled up to 216k cores.

There are several on-going efforts to develop parallel volume rendering methods for accelerators. Embree [21] is a photorealistic ray-tracer, which consists of a set of low-level kernels for multiple platforms, and has a simple API to port its kernels. OSPRay [20] is a multi-platform ray-tracing framework for scientific visualization on GPUs and multiple CPU architectures with varying SIMD widths, and is integrated into VisIT and ParaView. BnsView [8] is a molecular visualization framework which delivers fast volume rendering and ball-and-stick ray casting on Xeon Phi and is implemented in a SPMD language. Larsen et. al. [9] presented a method for ray-tracing consisting entirely of data parallel operators such as map, gather, scatter, reduce, and scan, which are optimized for CPU and GPU. VTK-m library [12] employs Larsen's algorithm for the ray-tracer and supports in-situ visualization mode on various multithreaded devices such as CPU, GPU, and MIC. In our previous work [5], PBVR was implemented on GPU, and an order of magnitude speedup was achieved compared with CPU rendering.

In terms of interactive data exploration, transfer of the original data requires a dedicated visualization cluster and a sustained network bandwidth equal to the solution output rate in order to support the so-called runtime visualization [10]. Tu et al. [19] proposed an online approach with image delivery and demonstrated efficient monitoring of tera-scale earthquake simulations running on supercomputers with thousands of processors. Over a wide-area network, they were able to interactively change visualization parameters to visually monitor simulation runs [18]. This kind of online approach was also extended to lightweight in-situ application called Strawman [10], which supports multiple programming languages and data models with refined system interface. In Ref. [6], we proposed a runtime visualization framework, In-Situ PBVR, which enables an online approach by transferring lightweight volume rendering primitive data.

2. In-situ PBVR

PBVR comprises two processes: particle generation and particle projection [7] (see Fig. 1). The first process constructs a particle density function based on physical variables, and generates particles for representing volume data. In Fig. 1, the left shows the cell-by-cell particle generation using Monte-Carlo sampling. The particles are randomly located in each cell and its color is given by interpolated physical values.

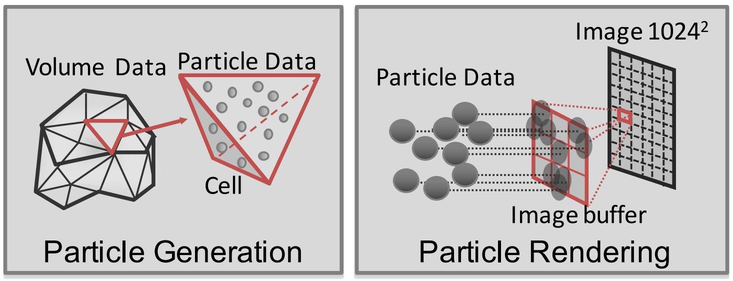


Figure 1. Image generation process of PBVR

The second process projects particles onto an image plane, and the particles are stored in the corresponding particle buffer, and final color and brightness values are synthesized from the illuminant particles in the buffer. In Fig. 1, the right shows the particle projection onto screen. The screen has the particle buffer which is consist of color and depth buffers, and each pixel is subdivided to sub-pixels. After the projection, the color and depth information in sub-pixels are synthesized to calculate the final pixel color.

PBVR generates particles by referring to the particle density function, which represents a number of particles in a unit volume. The particle density function is derived from a user-specified MDTF. The particles are generated in a cell-by-cell manner. In each cell, the locations of the particles are calculated via Monte Carlo sampling to avoid lattice patterns. The number of particles in each cell is calculated by volume integration of the particle density function. The

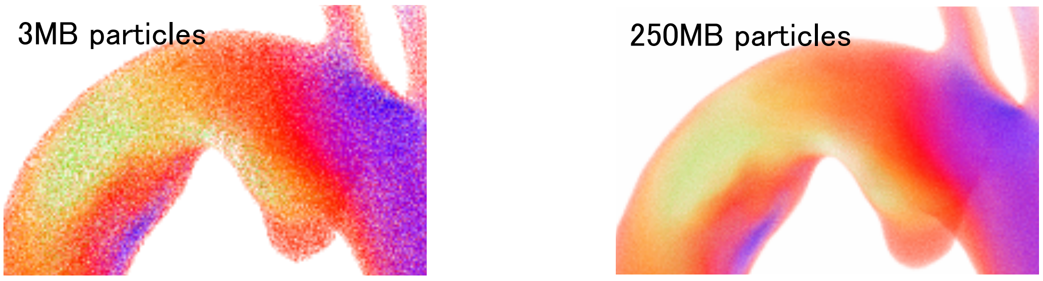


Figure 2. The PBVR image quality and the number of particles

size of particle data often becomes several orders of magnitude smaller than that of the original simulation data, and is determined by a flexible level of details (LOD) control. The left and right of Fig. 2 shows coarse and fine images with $1,024 \times 1,024$ pixels generated using $\sim$ 3MB particle data ($\sim$ 0.1M particles) and $\sim$ 250MB particle data ($\sim$ 9.3M particles), respectively. Here, a single particle has 27MB data consisting of particle position (4Byte$\times$3), RGB color (1Byte$\times$3), and normal vector (4Byte$\times$3). One can control the image resolution by varying the number of particles depending on the purpose of visualization. For example, light weight particle data may be useful for interactive data exploration via narrow bandwidth network. On the other hand, heavy particle data may be desirable for generating high fidelity images, once MDTFs are properly designed through in-situ data exploration.

In-situ PBVR consists of three main components: "Sampler" connected to the simulation solver on computing nodes, "Daemon" launched on an interactive node, and "Viewer" providing the multivariate volume renderer and GUI to design MDTFs on client PC. Particle generation and projection processes are computed by Sampler and Viewer respectively, and Daemon controls data transfer between them. Fig. 3 shows the whole design of the developed framework.

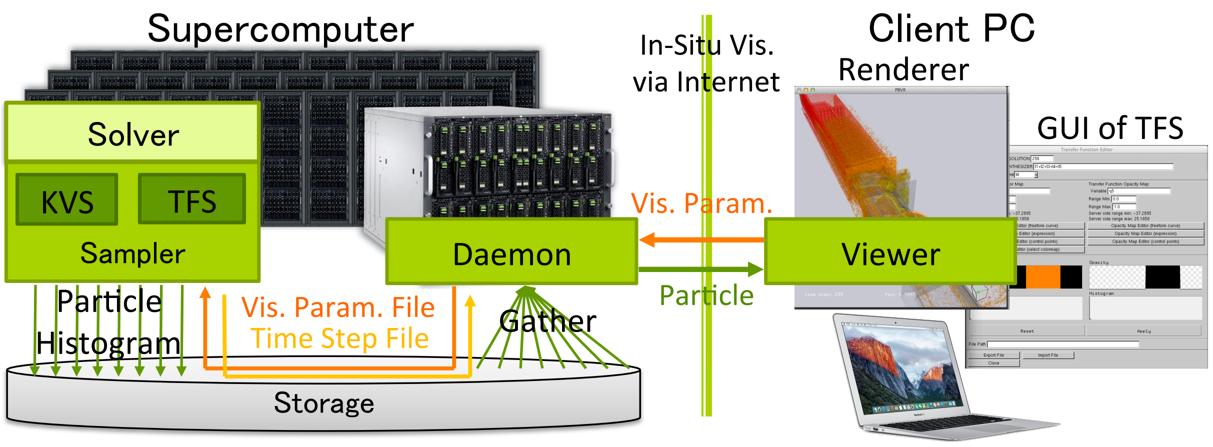


Figure 3. Three components of In-situ PBVR. From left to right, Sampler, Daemon, and Viewer

2.1. Viewer

Viewer on client PCs receives particle files, histogram files, and time step files from Daemon. Here, the particle file contains particle data, the histogram file has the information on distribution of (synthesized) variables, and the time step file contains other control parameters. Viewer renders particle data by projecting particles onto an image plane. Since the particle data is view-independent, one can easily change viewpoints, shading types, and light directions to explore the volume data. Since the maximum particle data size is several hundreds MB, Viewer works on PCs with small memory, and its thread parallel implementation with OpenMP enables interactive data exploration at interactive frame rate [5].

Another important interactive feature is flexible design environment of MDTFs supported by Transfer Function Synthesizer (TFS) [4]. Based on algebraic expressions, TFS generates new variables at runtime following definitions given as functions of existing multiple variables, coordinates, and time. The definitions of new variables are stored as character data of algebraic expressions. 1D TFs, which defines the color and the opacity as a function of a single variable, are designed using GUIs based on algebraic expressions, control points, or freeform curves. 1D TFs are stored either as character data or as data tables. Finally, algebraic synthesis of multiple 1D TFs gives a MDTF, which defines the color and the opacity independently as functions of multiple variables. The definition of MDTF is stored as character data of algebraic expressions. Together with other visualization parameters such as the number of particles, the above MDTF information is transferred to Daemon, and stored in a visualization parameter file.

2.2. Sampler

Sampler is written in C++ based on a visualization library KVS [16], which supports various data structures for visual programing. The routines of Sampler are wrapped in C interface, which can be called also from Fortran programs. Sampler is parallelized using a hybrid MPI and OpenMP parallelizaiton model, where MPI parallelization follows domain decomposition of the simulations, while OpenMP is applied to cell-by-cell parallelization in each subdomain. This parallelization strategy makes the API of Sampler quite simple. Sampler is easily coupled to the simulation by calling the following function from each MPI process at each time step.

```
void generate_particles( Type** subvolume, Param* parameters )
```

Here, `subvolume` is data array of resulting multivariate volume data in each subdomain, and `parameters` involve the information of volume data size and MPI parameters such as the rank ID, the number of MPI processes, and MPI communicators. A pseudo code of `void generate_particles` is shown in Code 1, where `parameters` are visualization parameters including MDTF read from the visualization parameter file, `multi_dim_tf` gives the color and the opacity by processing MDTF, `interpolator` calculates trilinear scalar interpolation at given point, `gradient` defines the normal vector from the gradient of the opacity around given point, `conv_opa2dens` converts the opacity to the normalized particle density [7], `particle_object` is a dynamic array for generated particles, which have point (particle position), color (8bit RGB), and gradient (normal vector), following the KVS format. Since the number of particles in each cell varies depending on multivariate volume data and MDTFs, the particle generation is parallelized in a cell-by-cell manner with dynamic scheduling. In each cell, the number of particles is computed, particles are generated following the opacity or the particle density via Monte-Carlo sampling, and the color is computed from MDTFs. The most expensive part of the particle generation is `multi_dim_tf`, in which algebraic expressions of MDTFs, which is given as character data, are interpreted and expanded into a tree structure by using a function purser, and then, volume data synthesis, 1D TF computation, and MDTF synthesis are computed sequentially. In order to reduce the cost of `multi_dim_tf`, we synthesize volume data before starting the particle generation, and keep a copy of the volume data of opacity, which is used for calculation of the normal vector given by the gradient of the opacity.

Although Sampler can be processed using the original volume data on the memory of computing nodes, we produce a copy of the volume data before the particle generation, so as not to affect the simulation. As a result, the current Sampler consumes memory space for a copy of the original multivariate volume data, the volume data of opacity, and generated particles.

When the visualization parameter file is updated, visualization parameters are distributed from the master process, and the corresponding histogram data computed by using the new visualization parameters are reduced to the master node. Apart from this initialization process, the particle generation is processed without any MPI communication, and the generated particle data is written from each process using POSIX I/O in an asynchronous manner.

2.3. Daemon

Daemon is operated on an interactive processing node or on a login node, which can access the storage and are connected to the internet, and controls the following data transfer between Viewer and Sampler.

- Sampler $\leftarrow$ Viewer: visualization parameter file
- Sampler $\rightarrow$ Viewer: particle file, histogram file, time step file

Daemon interacts with Viewer via socket communications (through ssh tunnel), while it interacts with Sampler via asynchronous file-based control sequences. When new visualization parameters are transferred from Viewer, Daemon writes them to a new visualization parameter file. As shown in Code 1, when Sampler is launched from the simulation, it detects the new parameter file, and the master process reads new visualization parameters and distributes them via MPI_Bcast. In principle, this process may be constructed without MPI_Bcast, provided that all processes can read the same visualization parameter file. However, depending on the timing of file update, all computing nodes may not access the same file, and the coherence of visualization parameters may break down. To avoid such a failure, the above file control sequence is designed. Even

Algorithm 1 `generate_particles` on each MPI rank

```
 1: MPI_Barrier
 2: if Open( visualization_parameter_file ) == true then
 3:     if mpi_rank == 0 then
 4:         Read( visualization_parameter_file ) → parameters
 5:         Rename( visualization_parameter_file → old_parameter_file )
 6:     end if
 7:     MPI_Bcast( parameters )
 8:     MPI_Reduce( histograms )
 9:     Write( histogram_file ) ← histograms
10: end if
11: #pragma omp for schedule( dynamic ) nowait
12: for each cell do
13:     scalars ← interpolator( gravity center of cell )
14:     opacity ← multi_dim_tf( scalars )
15:     density ← conv_opa2dens( opacity, visualization parameters )
16:     num_particles ← density * volume of cell
17:     c = 0
18:     while c < num_particles do
19:         point ← randomly sampled point in each cell.
20:         scalars ← interpolator( point )
21:         opacity ← multi_dim_tf( scalars )
22:         density ← conv_opa2dens( opacity, visualization parameters )
23:         r ← random number from 0 to 1.
24:         if density > r then
25:             vector ← gradient( point )
26:             color ← multi_dim_tf( scalars )
27:             particle_object ← ( point, color, vector )
28:             c++
29:         end if
30:     end while
31: end for
32: Write( particle_file ) ← particle_object
33: if mpi_rank == 0 then
34:     Write( time_step_file )
35: end if
```

with this control sequence, when Daemon and Sampler access the visualization parameter file simultaneously and the update of the visualization parameter file is not completed, Sampler often fails because of incomplete visualization parameters. To avoid this error, the exclusive control of write and read sequences of the visualization parameter file is designed, so that Daemon writes "END_OF_FILE" statement at the end of the visualization parameter file, and before Sampler reads the visualization parameters, it waits until this statement is detected in the file.

On the other hand, when the current time step in the time step file is updated by the master process of Sampler, Daemon starts to gather the particle files. This file I/O is thread parallelized via OpenMP. Here, their file names include the information of the current time step, the rank ID, and the number of MPI processes. By using this information, Daemon detects the completion of the file gather, before it launches transfer of the merged particle data to Viewer.

3. Performance Evaluation

In-Situ PBVR was originally developed on the ICEX, which consists of 2,510 Intel Xeon E5-2680v3 connected via the Infiniband FDR interconnect, and its performance showed excellent strong scaling up to 3,456 cores [6]. In order to estimate its performance on the latest many core platforms, which have significantly different hardware characteristics from the ICEX, we conduct a numerical experiment on the Oakforest-PACS, in which 8,208 computing nodes with Intel Xeon Phi7250 (Knights Landing) are connected via the Intel Omni-Path Host Fabric Interface.

Sampler is connected to the JUPITER code [22], which computes relocation behavior of molten debris in reactor pressure vessels based on thermal-hydraulic equations and multiphase simulation models. The code is based on structured grids with 3D domain decomposition, and is highly parallelized using a hybrid MPI and OpenMP parallelization model. A typical simulation duration of the JUPITER code is ∼3,000,000 time steps, and visualization is processed at every 1,000 steps. However, in this numerical experiment, Sampler is called at every time steps. A strong scaling test is performed with the fixed problem size of $240 \times 240 \times 1,920 \sim 10^8$ grids. Sampler generates $\sim 10^7$ particles (∼250MB), which is typically used for high quality image with $1,024 \times 1,024$ pixels. In the strong scaling test, the number of MPI processes per node is fixed to 4, and the number of threads per MPI process is chosen to be 16. Therefore, 64 cores per node are used without hyper-threading, while the Oakforest-PACS has 68 cores per node. The multi-channel dynamic random access memory (MCDRAM) is used in a cache mode. In the experiment, the number of nodes (cores) is increased from 24 (1,536) to 1,536 (98,304). The computational cost is measured over 20 time steps, in which visualization parameters are updated once.

The strong scaling test is summarized in Tab. 1 and in Fig. 4. In Tab. 1, "Solver" and "Sampler" mean the costs of the JUPITER code and the In-Situ PBVR, respectively. "Write" and "Update" are respectively the costs of updating visualization parameters and writing particle data, which are included in Sampler. The result shows that the cost of Sampler is suppressed between $\sim 10\%$ and $\sim 45\%$ of the JUPITER code (Solver). If one calls Sampler less frequently, this cost is negligibly small. Both Solver and Sampler scale up to $\sim$ 100k cores. However, at 1,536 nodes, the problem size per thread is reduced to $15 \times 15 \times 5$, which is almost the upper limit of strong scaling, and Solver suffers from significant performance degradation. As a result, the acceleration of Solver between 24 and 1,536 nodes (the peak performance ratio of $\times 64$) is limited to $\times 3.9$. Although Sampler shows better acceleration ratio $\times 9.5$, it is still far from the peak performance ratio. This performance degradation may be attributed to the cost of writing the particle data, and the load imbalance inherent to the particle distribution reflecting the (synthesized) multivariate volume data. The overhead of file I/O increases from $\sim 1\%$ to $\sim 79\%$ of the total Sampler cost. It is noted that the update of visualization parameters includes MPI_Bcast and MPI_Reduce, and thus, the overhead of synchronization among all nodes was anticipated. However, the result shows very small impact from the update.

Another important result is the memory usage summarized in Tab. 2 where "Solver" and "Sampler" mean the memory usage of the JUPITER code and the In-Situ PBVR, respectively. The memory usage of Solver is based on the maximum memory size reported from the job scheduler, while the memory usage of Sampler is calculated from the difference of the memory usage before and after the memory allocation for the particle generation. Here, the instantaneous memory usage is obtained from "/proc/[process ID]/stat" file. Even with the fixed problem size, the memory usage of Solver shows explosive growth due to increasing halo regions and MPI buffers. On the other hand, the memory usage of Sampler shows moderate growth, and the memory usage per MPI process is suppressed between ~ 3.3MB and ~ 92MB. This extreme low memory usage is an important advantage of In-Situ PBVR.

The particle data was transferred to client PCs via socket communication, and the multivariate volume data was rendered on Viewer, in which the viewpoint is changed at about 10 fps.

Table 1. Distribution of computational costs per time step observed in the numerical experiment on Oakforest-PACS

Nodes	24	96	384	1,536
Cores	1536	6,144	24,576	98,304
Solver [sec/step]	26.7	10.3	7.3	6.8
Sampler [sec/step]	7.6	2.8	1.0	0.8
Write [sec/step]	0.078	0.064	0.124	0.645
Update [sec/step]	0.05	0.03	0.04	0.03

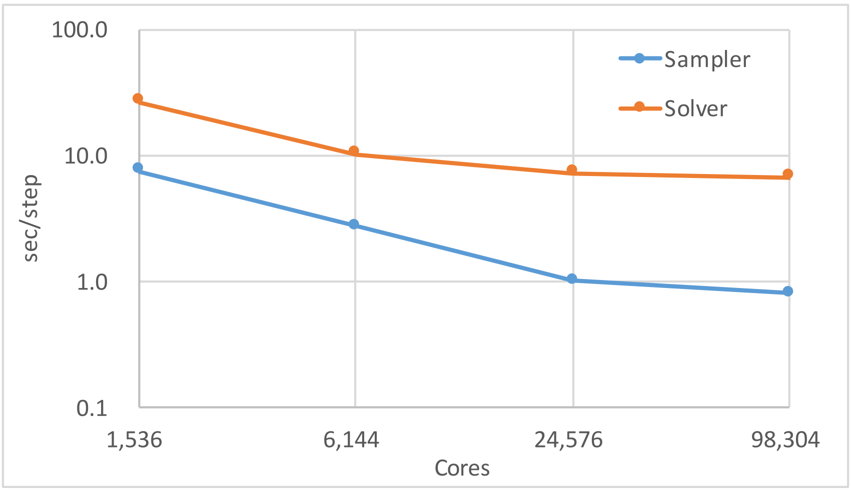


Figure 4. Strong scaling of the JUPITER code (Solver) and the In-Situ PBVR (Sampler) on Oakforest-PACS

Table 2. Total memory consumption observed in the numerical experiment on Oakforest-PACS

Cores	1536	6,144	24,576	98304
Solver [GB]	106.7	135.0	256.7	773.2
Sampler [GB]	8.9	9.5	11.6	20.4

Conclusion

We have examined the performance of the In-Situ PBVR framework on the latest many-core platform based on Knights Landing processors. The proposed framework is designed to process parallel in-situ visualization with the minimum memory usage and to enable interactive in-situ data exploration. Sampler is parallelized by MPI for each decomposed subdomain and by OpenMP for each cell, respectively. The result of strong scaling test shows that Sampler performance scales up to $\sim$ 100k cores with extremely low memory usage. These are promising features for future exa-scale in-situ visualization frameworks. Interactive data exploration is realized by the following two features. Firstly, we designed asynchronous file-based control sequences for interactive update of visualization parameters between Sampler and Daemon, so that the performance of Sampler is not affected by the interactive procedure. Secondly, In-Situ PBVR is based on view-independent rendering primitives, which are given as relatively small particle data. The particle data is easily transferred to remote PCs via socket communication, and is rendered at interactive frame rate.

Acknowledgments

This work is partially supported by "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures" and "High Performance Computing Infrastructure" in Japan. This research was supported by MEXT as "Post-K priority issue No.6: Development of Innovative Clean Energy".

References

1. Visit user's manual. Tech. rep., Lawlence Livermore National Laboratory (2005), `https://wci.llnl.gov/simulation/computer-codes/visit/manuals`

2. Henderso, A.: Paraview guide, a parallel visualization application. Tech. rep., Kitware Inc. (2005), `http://www.paraview.org`

3. Howison, M., Bethel, E.W., Childs, H.: Hybrid parallelism for volume rendering on large-, multi-, and many-core systems. IEEE Trans. Vis. Comput. Graph. 18(1), 17–29 (2012), DOI: 10.1109/TVCG.2011.24

4. Kawamura, T., Idomura, Y., Miyamura, H., Takemiya, H.: Algebraic design of multi-dimensional transfer function using transfer function synthesizer. Journal of Visualization

20(1), 151–162 (Feb 2017), DOI: 10.1007/s12650-016-0387-1

5. Kawamura, T., Idomura, Y., Miyamura, H., Takemiya, H., Sakamoto, N., Koyamada, K.: Remote visualization system based on particle based volume rendering. In: Proceedings of the conference on VDA. Visualization and Data Analysis 2015, SPIE (2015), DOI: 10.1117/12.2083501

6. Kawamura, T., Noda, T., Idomura, Y.: In-situ visual exploration of multivariate volume data based on particle based volume rendering. In: Proceedings of the 2Nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization. pp. 18–22. ISAV '16, IEEE Press, Piscataway, NJ, USA (2016), DOI: 10.1109/ISAV.2016.9

7. Kawamura, T., Sakamoto, N., Koyamada, K.: Level-of-detail rendering of a large-scale irregular volume dataset using particles. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 25(5), 905–915 (2010), DOI: 10.1007/s11390-010-9375-4

8. Knoll, A., Wald, I., Navrátil, P.A., Papka, M.E., Gaither, K.P.: Ray tracing and volume rendering large molecular data on multi-core and many-core architectures. In: Proceedings of the 8th International Workshop on Ultrascale Visualization. pp. 5:1–5:8. UltraVis '13, ACM, New York, NY, USA (2013), `http://doi.acm.org/10.1145/2535571.2535594`

9. Larsen, M., Meredith, J., Navrátil, P., Childs, H.: Ray-Tracing Within a Data Parallel Framework. In: Proceedings of the IEEE Pacific Visualization Symposium. pp. 279–286. Hangzhou, China (Apr 2015), `10.1109/PACIFICVIS.2015.7156388`

10. Larsen, M., Brugger, E., Childs, H., Eliot, J., Griffin, K., Harrison, C.: Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), held in conjunction with SC15. pp. 30–35. Austin, TX (Nov 2015), `http://doi.acm.org/10.1145/2828612.2828625`

11. Levoy, M.: Display of surfaces from volume data. IEEE Comput. Graph. Appl. 8(3), 29–37 (May 1988), DOI: 10.1109/38.511

12. Moreland, K., Sewell, C., Usher, W., ta Lo, L., Meredith, J., Pugmire, D., Kress, J., Schroots, H., Ma, K.L., Childs, H., Larsen, M., Chen, C.M., Maynard, R., Geveci, B.: Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. IEEE Computer Graphics and Applications 36(3), 48–58 (2016), DOI: 10.1109/MCG.2016.48

13. Peterka, T., Yu, H., Ross, R., Ma, K.L.: Parallel volume rendering on the ibm blue gene/p. In: Proceedings of the 8th Eurographics Conference on Parallel Graphics and Visualization. pp. 73–80. EGPGV '08, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2008), DOI: 10.2312/EGPGV/EGPGV08/073-080

14. Peterka, T., Yu, H., Ross, R., Ma, K.L., Latham, R.: End-to-end study of parallel volume rendering on the ibm blue gene/p. In: Proceedings of ICPP 09. Vienna, Austria (2009), DOI: 10.1109/ICPP.2009.27

15. Sakamoto, N., Kawamura, T., Koyamada, K., Nozaki, K.: Improvement of particle-based volume rendering for visualizing irregular volume data sets. Computers & Graphics 34(1), 34–42 (2010), DOI: 10.1016/j.cag.2009.12.001

16. Sakamoto, N., Koyamada, K.: Kvs: A simple and effective framework for scientific visualization. Journal of Advanced Simulation in Science and Engineering 2(1), 76–95 (2015), `http://doi.org/10.15748/jasse.2.76`

17. Sakamoto, N., Nonaka, J., Koyamada, K., Tanaka, S.: Particle-based volume rendering. Asia-Pacific Symposium on Visualization 2007 pp. 129–132 (2007), DOI: 10.3154/tvsj.27.7

18. Tu, T., Yu, H., Bielak, J., Ghattas, O., López, J.C., Ma, K., O'Hallaron, D.R., Ramírez-Guzmán, L., Stone, N., Taborda-Rios, R., Urbanic, J.: Analytics challenge - remote runtime steering of integrated terascale simulation and visualization. In: Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA. p. 297 (2006), DOI: 10.1145/1188455.1188767

19. Tu, T., Yu, H., Ramirez-Guzman, L., Bielak, J., Ghattas, O., Ma, K.L., O'Hallaron, D.R.: From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. SC '06, ACM, New York, NY, USA (2006), DOI: 10.1145/1188455.1188551

20. Wald, I., Johnson, G.P., Amstutz, J., Brownlee, C., Knoll, A., Jeffers, J., Gunther, J., Navrátil, P.A.: Ospray - A CPU ray tracing framework for scientific visualization. IEEE Trans. Vis. Comput. Graph. 23(1), 931–940 (2017), DOI: 10.1109/TVCG.2016.2599041

21. Woop, S., Feng, L., Wald, I., Benthin, C.: Embree ray tracing kernels for cpus and the xeon phi architecture. In: ACM SIGGRAPH 2013 Talks. pp. 44:1–44:1. SIGGRAPH '13, ACM, New York, NY, USA (2013), DOI: 10.1145/2504459.2504515

22. Yamashita, S., Yoshida, H., Takase, K.: Development of numerical simulation method for relocation behavior of molten debris in nuclear reactors (1) preliminary analysis of relocation of molten debris to lower plenum. In: Proceedings of 21st International Conference on Nuclear Engineering (ICONE-21). vol. 4, p. V004T09A109. Nuclear Engineering Division (2013), DOI: 10.1115/icone21-16604

DOI: 10.14529/jsfi170303

Development and Integration of an In-Situ Framework for Flow Visualization of Large-Scale, Unsteady Phenomena in ICON

Michael Vetter[1], *Stephan Olbrich*[2]

With large-scale simulation models on massively parallel supercomputers generating increasingly large data sets, in-situ visualization is a promising way to avoid bottlenecks. Enabling in-situ visualization in a simulation model asks for special attention to the interface between a parallel simulation model and the data analysis part of the visualization, and to presentation and interaction scenarios. Modifications to scientific workflows would potentially result in a paradigm shift, which affects compute and data intensive applications generally. We present our approach for enabling in-situ visualization within the highly parallelized climate model ICON using the DSVR visualization framework. We focus on the requirements for generalized grid and data structures, and for universal, scalable algorithms for volume and flow visualization of time series. In-situ pathline extraction as a technique for the visualization of unsteady flows has been integrated in the climate simulation model ICON and verified in first studies.

Keywords: DSVR, ICON, in-situ visualization, visualization framework.

Introduction

High-resolution simulation of climate phenomena tends to produce very large data sets, which hardly can be processed in classical post-processing visualization applications. Typically, the visualization pipeline consisting of the processes data generation, visualization mapping, and rendering is distributed into two parts over the network or separated via file transfer [4, 6, 20]. Within most traditional post-processing scenarios, the simulation is executed on a supercomputer whereas data analysis and visualization is done on a graphics workstation. That way temporary data sets with huge volume have to be transferred over the network, which leads to bandwidth bottlenecks and volume limitations. A solution to this issue is the avoidance of temporary storage, or at least significant reduction of data complexity. This can be achieved by in-situ visualization, where the visualization is tightly coupled to the data generation [9]. In our work we focus on this topic, as well as on further challenges in extreme-scale visual analytics [19].

One actual climate simulation model is the ICON (Icosahedral non-hydrostatic) general circulation model, which was initiated by the Max Planck Institute for Meteorology (MPI-M) and the German weather service "Deutscher Wetterdienst" (DWD) [21]. Within the Climate Visualization Lab – as part of the Cluster of Excellence "Integrated Climate System Analysis and Prediction" (CliSAP) at Universität Hamburg, in cooperation with the German Climate Computing Center (DKRZ) – we enhance our in-situ approach and integrate it into the ICON framework.

The article is organized as follows. Section 1 introduces our DSVR framework [13] in the context of state-of-the-art visualization approaches. In section 2 we present the development steps to support generic grids. Section 3 addresses the integration and application of DSVR based flow visualization in ICON. The Conclusion summarizes the study and points directions for further work.

[1]Centrum für Erdsystemforschung und Nachhaltigkeit (CEN), Universität Hamburg, Germany
[2]Regional Computing Center (RRZ), Universität Hamburg, Germany

1. Parallel Data Extraction and Visualization in "DSVR"

As the data exchanged between the processes of the visualization pipeline decreases in volume, the level of exploration also decreases. So the partitioning of the visualization pipeline while designing a visualization system is always a trade-off decision between those two parameters. Common in-situ approaches fulfill the full visualization task on the supercomputer running the simulation and just store 2D pixel data within a movie. In another common approach, the visualization framework let a remote user connecting directly into the running simulation in order to allow live visualization of the running simulation. Well-known visualization systems utilizing one or both of these approaches are: Paraview Catalyst [1, 5] or VisIt [18]. An overview of most common in-situ visualization frameworks is given at [2]. We decided to follow the second approach, separating the process chain between the mapping and the rendering processes, since this enables real-time streaming while preserving user interactions like explorative camera changes, setting of lighting or time shifting within the visualization (see fig. 1).

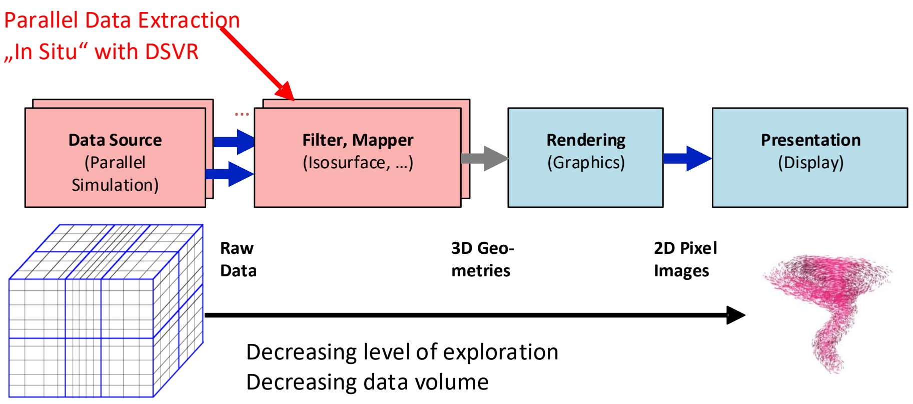


Figure 1. The visualization pipeline of DSVR

1.1. Distributed Streaming and Visualization Framework

In contrast to other visualization frameworks, where in-situ capabilities typically are built on top of existing post processing visualization applications, the DSVR framework [7, 13] was designed for in-situ visualization from scratch. This includes design and development of distributed software components, as well as network protocols and interfaces. As shown in fig. 2, the DSVR framework consists of three main components:

3D Generator: To enable in-situ visualization, the mapping process is tightly coupled to the simulation by calling methods of a parallelized data extraction library called libDVRP. This way the visualization mapping algorithms have access to the transient raw data in memory, using the domain decomposition of the calling simulation. The visualization library also combines 3D data compression with the mapping algorithms. This includes polygon reduction by adaptive vertex clustering within the isosurface generation [10], as well as compression of pathlines. Pathlines could be enhanced by local properties of the simulation to allow interactive post-filtering [15]. This results in a time-based sequence of geometric 3D scenes, interleaved with samples of raw data or extracts, which can be streamed and visualized in 3D.

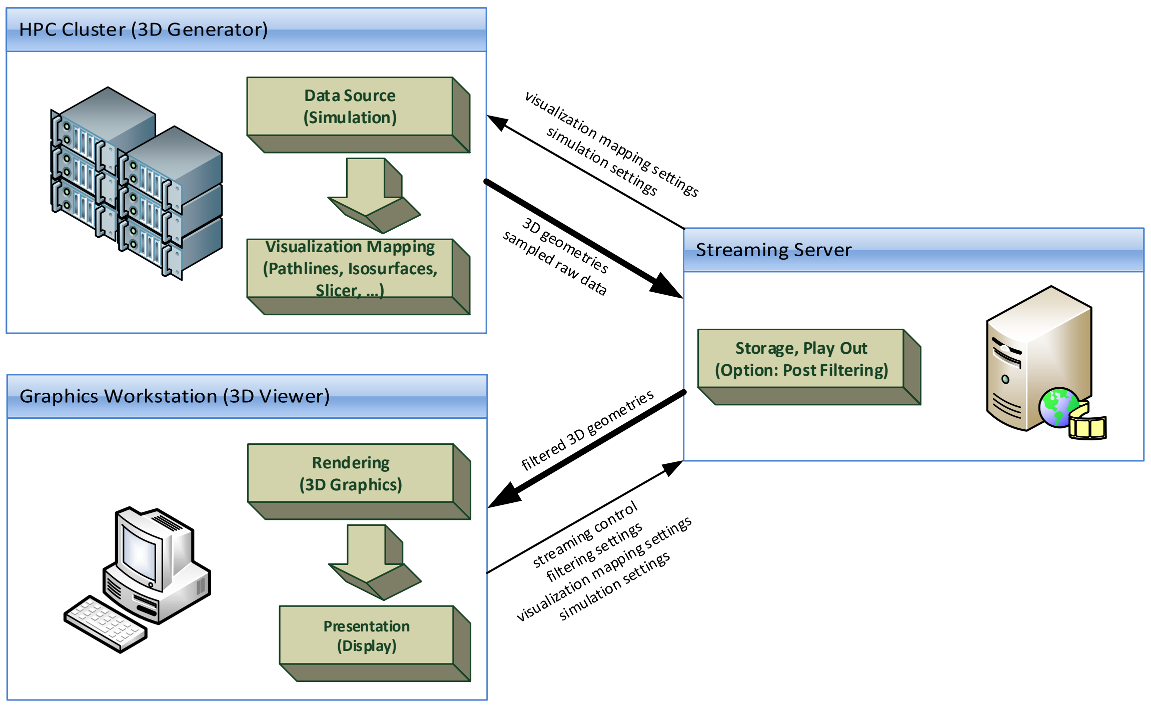

HPC Cluster (3D Generator)
Data Source (Simulation)
Visualization Mapping (Pathlines, Isosurfaces, Slicer, ...)
visualization mapping settings
simulation settings
3D geometries
sampled raw data
Streaming Server
Storage, Play Out (Option: Post Filtering)
Graphics Workstation (3D Viewer)
Rendering (3D Graphics)
Presentation (Display)
filtered 3D geometries
streaming control
filtering settings
visualization mapping settings
simulation settings

Figure 2. The DSVR framework consisting of libDVRP, streaming server, and rendering client

Streaming Server: A unique, advanced streaming feature of DSVR also has capabilities for further reduction of transferred data by server-side filtering of geometric objects based on sampled raw data as well as support of synchronization and control of frame rates. To support long running simulations, the 3D models can be stored on a separate persistent storage component, which is realized as a streaming server providing record and play functions and then played-out asynchronously.

3D Viewer: The 3D viewer client was first realized as a multi-platform browser plugin based on the NPAPI, and is now provided as a lightweight stand-alone version. Since the OpenGL based rendering is part of this viewer application, the scenes can be navigated interactively, optionally supported by stereoscopic 3D and tracking systems. In contrast to other in-situ approaches where 2D images are created as part of the simulation or a synchronous co-visualization takes place, our method supports interaction in 3D space and in time, as well as control of frame rates.

1.2. In-Situ Pathline Extraction

Techniques for visualization mapping of raw data in a three dimensional domain can be classified into volume visualization for scalar data and flow visualization for vector data. The integration of streamlines or pathlines using particle tracking algorithms is a common techniques for visualization of unsteady flow fields. Parallelization of particle tracking algorithms utilizes basically one of the following parallelization schemes: parallelization over seeds, parallelization over blocks (spatial domain and optional time), or hybrid approaches. For parallel performance of one or another parallelization scheme, key influencing factors are data set size, seed set size, seed distribution as well as vector field complexity [14]. Seeding strategies for pathline visualization not only affect the algorithm's performance but also the visualization's insight. A lot of seeding strategies have been discussed for the integration of streamlines within steady flows, but they could not be applied straight forwardly to unsteady flows. Seed points can be evenly distributed all over the volume with some termination or filtering function to avoid visual cluttering. For better results the data set is preprocessed to find critical points of the vector field. An overview about geometric flow visualization including common seeding strategies is given in [12].

In our work we especially focus on techniques for visualization of time dependent scalar and vector fields. Some algorithms we implemented in DSVR are already well proven and highly optimized for the parallel visualization mapping and data reduction. Since the DSVR library was originally designed to be used with rectilinear grids, the implemented algorithms take direct advantages of the grid structure and are tightly bound to the grid. Parallel isosurface extraction with integrated flexible polygon simplification, as well as parallel pathline extraction for feature enhanced pathlines had been implemented and applied in oceanic and atmospheric simulations based on the parallel large-eddy simulation model PALM [11, 16]. Using the simulation's domain decomposition the parallelization of the pathline extraction is natively given as parallelization over blocks. Although time is a fourth dimension in time dependent flow visualization, parallelization over time steps is not an option due to mainly two reasons: (1) within an in-situ approach, only data of a few time steps can be hold in transient memory because of memory limitations, and (2) pathline extraction uses iterative, serial algortihms to trace particles over time steps. The seeding strategy would typically distribute a very large amount of seed points over the simulated volume, using a predefined pattern. Visual cluttering is reduced by interactive filtering during the streamed presentation, based on the enhancement of the pathlines with a set of properties gained from the simulation.

1.3. Support of Simulation Applications

To enable DSVR based in-situ visualization within a simulation, the data extraction library, called libDVRP, needs to be integrated within the simulation code. The library is written in C and provides an additional Fortran interface. libDVRP supports MPI-based parallel environments, and it encapsulates the handling of the transient data parts in 3D space and time, according to the given domain decomposition and iterative solution, and the serialization for file or streaming output of extracted 3D primitives or other data. The integration of the library into a simulation can be done easily by adding a few lines of code as shown in fig. 3. In most simulation codes a simple scheme can be found: First there is a bunch of initialization routines, then within a main loop for each simulated time step the model data is calculated, followed by finalization routines.

Within the simulation's initialization routines, libDVRP needs to be configured. First of all libDVRP needs to be initialized to set internal data structures to defaults and the output mode has to be selected. Here the options are writing 3D sequences directly to file or to the DSVR streaming server via TCP/IP. Then the simulation's grid configuration needs to be given to the library. After this the visualization has to be parametrized by setting seed points for pathlines or thresholds for isosurfaces for example.

At the end of the simulation's main loop, when all calculations are done, the data fields needs to be provided to the library. Then `DVRP_Visualize()` can be called to let libDVRP extract the 3D geometries. Within the simulation's finalization our library should be finalized.

```
PROGRAM simulation
    ! initialize simulation
    CALL DVRP_Initialize
    CALL DVRP_SetOutput
    CALL DVRP_SetGrid
    ! call methods for setting visualization parameters,
    ! eg. DVRP_Threshold, DVRP_Material, DVRP_Cubic_Seeding
    DO ! simulations main loop
        ! do the simulation step and calculate all those fields
        CALL DVRP_SetDataFields
        CALL DVRP_Visualize
    END DO
    CALL DVRP_Finalize
    ! finalize simulation
END PROGRAM simulation
```

Figure 3. Integration of DSVR in-situ visualization in a simulation code

2. Development of Visualization Methods on Generic Grids

ICON internally uses an icosahedral grid structure, which fits better for a spherical problem domain found in earth sciences. Since ICON also includes output modules for rectilinear grids, we had to do a design decision for integrating DSVR in ICON: Would we like to make usage of the ICON output routines and fit them for our needs or reimplement our algorithms for the ICON grid? Mapping of the ICON raw data onto rectilinear grids is attended by several issues: (1) grid transformation requires recalculating all needed data fields on all grid points, (2) using the same amount of grid points will lead to oversampling at the poles and to undersampling at the equator and (3) matching the grid point distance of the original grid with the rectilinear grid at the equator, won't lead to undersampling but needs a higher memory footprint due to oversampling. We decided to implement a highly generalized approach.

Given that other grid structures beside rectilinear and prism grids will get required in the future, we decided for a paradigm shift for the development of libDVRP. The overall goal is to implement visualization algorithms independent of the simulations' grid structure while preserving the possibility to optimize the in-situ visualization for the individual simulation data structures. This results in a gridAPI written in C, which abstracts the grid and data relevant operations from the visualization algorithms as known from object oriented programming approaches. For each new grid, a realization of each of the gridAPI's functions has to be implemented. The gridAPI will then act like a proxy redirecting the function calls.

The API is designed as an integrated software layer between the simulation and the visualization algorithms (fig. 4). It includes methods to be called by the libDVRP visualization routines as well as function to be called by the simulation. While the simulation uses "setter"-functions for setting grid type, grid parameters and data fields, the visualization routines typically use "getter"-functions to get data values or the MPI process, having the data at given coordinates or grid indices. In addition, it implements some sort of iterator to be used on the grid cells. The design of the gridAPI still allows for grid specific optimization, since the major getter methods allows unspecific optional optimization parameters. This could be used to store previously calculated cell indices or starting positions for search operations for example.

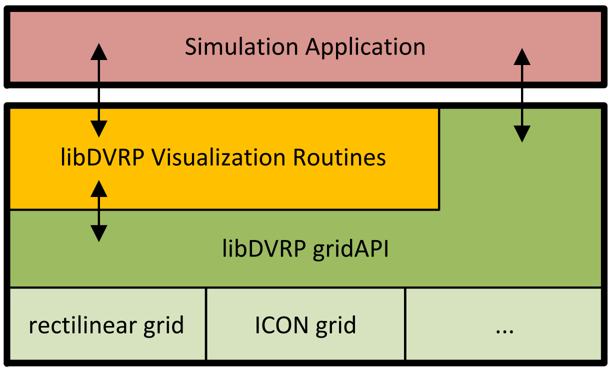


Figure 4. libDVRP gridAPI layout

In order to take usage of the gridAPI, a reimplementation of the visualization algorithms is needed. We implemented universal parallel algorithms for pathline extraction as well as for isosurface extraction, both applying the gridAPI.

2.1. Pathline Extraction

The algorithm for pathline extraction using the gridAPI was based on the MPI-parallelized pathline algorithm already introduced in [15, 16] with an optimized communication scheme as in fig. 5.

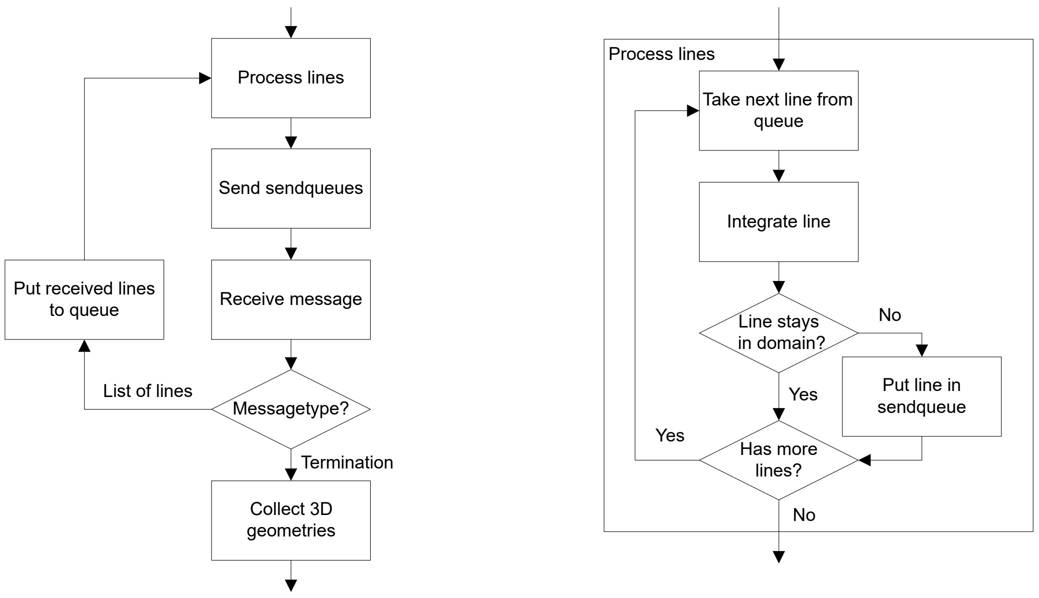


Figure 5. Process diagram of pathline extraction

As the parallelization is done using 1D, 2D, or 3D domain decomposition of the 3D data grid given by the simulation model, integration of pathlines is limited to the local domain by each MPI process. Parallelization of visualization over time steps is not possible (see 1.2), and generally not realized in simulation models. The data of at most two time steps is cached in libDVRP, to enable higher order numerical integration. On every call of `DVRP_Visualize()` each MPI process iterates over all lines within the domain and integrates the next supporting point for each line using Euler or Runge-Kutta integration. When a line leaves local domain of one process, it is sent to the MPI process holding the needed data. For optimized MPI communication, traversing lines are buffered and asynchronously communicated after all local lines have been processed. After the new pathlines entering from neighbor domains are received, the line processing starts again.

Within this algorithm, only two functions of the gridAPI were used: while integrating the line, the data values at a given position are requested by calling `DVRP_gridAPI_getValAtPos()` up to four times for the Runge-Kutta 4th order integration. After this the MPI process handling the data at the resulting position is requested to find out if the line stays within the local domain. This is done by calling `DVRP_gridAPI_getMPIAtPos()`. This way the task of finding a specific grid cell, as well as the data layout of the raw data, is shifted into the gridAPI. This allows for optimizations regarding the grid and data access without touching the pathline algorithm anymore.

2.2. Isosurface Extraction

In order to be independent of the used grid structure, the isosurface algorithm has to be as general as possible. The marching cubes algorithm [8] used for isosurface extraction within libDVRP could not be abstracted from grid and data layout. Contemplable algorithms have to meet several constraints: parallelization based on domain decomposition as well as data volume optimized resulting 3D meshes are hard requirements. This leads to two possible algorithms: Complex-Valued Contour Meshing [17] and Isosurface Construction using Convex Hulls [3]. The first algorithm fractionizes all 3D cells into a bunch of tetrahedrons and creates an isosurface for each tetrahedron using a lookup table. The second algorithm would dynamically generate the appropriate mesh pattern lookup tables for the necessary cell configurations during the run-time initialization. Both algorithms have their own advantages and disadvantages. Though the first algorithm is more generalized since it is not limited to convex cells, in a comparison test on a prism grid it generates 6.7 times more triangles for the same isosurface. Therefore, we favored the second algorithm, as it would generate the appropriate mesh pattern lookup tables for the necessary cell configurations dynamically during the run-time initialization and result in lesser amount of 3D geometric primitives.

For this algorithm we need 6 additional functions within the gridAPI: `startCellIteration`, `getNextCellId`, `getCellPoints`, `getCellValues`, `getEdgeDefinition` and `getCellDefinition`.

We have evaluated the isosurface algorithm using the artificial tornado like swirl by Crawfis[3] generated by a self-written test application. The data field was generated and visualized on a prism grid and on a rectilinear grid using the same grid points. The new algorithm was compared with the Marching Cubes algorithm. Using our new algorithm on a prism grid with the same amount of grid point but double amount of cells, the algorithm produces 47 percent more triangles.

3. Integration of DSVR Based Pathline Extraction in ICON

In order to integrate an in-situ processing based on our DSVR framework and methods in the state-of-the-art climate simulation model ICON, we are continuously evolving data structures of the framework to support the ICON model's native grid structures. Therefore, we implemented

[3]http://www.cse.ohio-state.edu/ crawfis/Data/Tornado/tornadoSrc.c

a realization corresponding to the ICON grid within our gridAPI. ICON uses a dual grid, where the primary grid consists of a triangle mesh around the globe with parallel layers for the height axis. The secondary, indirectly stored grid consists of hexagons connecting adjacent triangles.

After reimplementation of the pathline extraction algorithm we have implemented a gridAPI realization for rectilinear grids as well as for the ICON grid. To get scientists a better understanding about what DSVR is capable of, we implemented a stand-alone NetCDF post-processor, based on libDVRP (fig. 6). By using the NetCDF post-processor the 3 processes simulation, visualization mapping, and rendering are separated completely: the data set is processed in a batch mode – e.g. using the same supercomputer on which the data is generated – and the interactive 3D rendering is done afterwards on the scientist's local system.

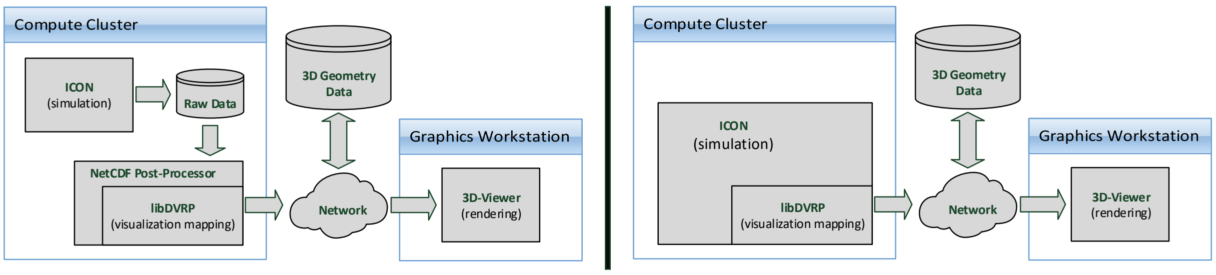


Figure 6. Architecture diagram of ICON visualization with DSVR using the NetCDF post-processor on the left side, and an in-situ visualization on the right

Since the NetCDF files do not necessarily contain any information about grid cells but only the coordinates of where the data is stored at, a gridAPI realization supporting leveled point sets was implemented. At the actual status of implementation, the post-processor supports the generation of isosurfaces and colored slicers on volume data set time series based on rectilinear grids as well as the visualization of pathlines on time varying flow fields based on either rectilinear grids or leveled point set grids (see fig. 7).

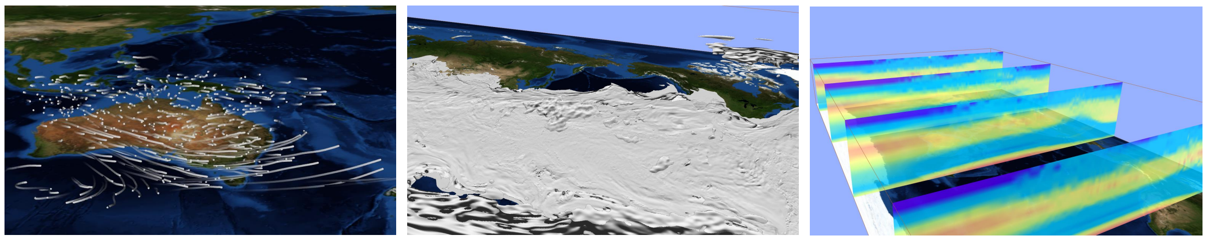


Figure 7. Sample visualizations of ICON data using the post-processor (from left to right): (a) pathlines of wind speed on ICON grid, (b) isosurface showing an atmospheric temperature of 273.15 K on rectilinear grid, and (c) colored slicer of atmospheric temperature on rectilinear grid

We have now implemented a new output module to ICON to take advantage of the DSVR visualization, which is called in the ICON simulation loop. The visualization can be configured as most output modules by using a specific namelist and is exemplarily integrated within the non-hydrostatic atmospheric model time loop. The module is initialized by a subroutine called `init_dvrp_output`. Here the ICON grid structure is collected, conditioned and set within libDVRP. Also the major configuration of the visualization is made here. The processing of every time step is done by a subroutine called `write_dvrp_output`, were data fields are copied to libDVRP and the visualization routine `DVRP_Visualize` is called. ICON calculates most flow vectors like wind speed at the cell center point. So the original ICON prism grid could be used assuming the flow is the same everywhere within the cell. On the other hand, the hexagonal grid can be used, with the original cell center points becoming the new grid points. The second option would allow interpolation of flow values at every position leading to better visualization results, so the decision fell on the usage of the hexagonal grid for pathline extraction. Since each hexagonal cell can be broken down on triangles, we implemented the interpolation method as well as the cell-searching algorithm on prism cells, taking advantage of the known clustering.

With the integration of a DSVR-based in-situ pathline extraction within ICON, the next milestone is reached. The pathline algorithm as well as the grid data structures has been optimized for the domain decomposition used for the parallelization of ICON based on MPI and OpenMP. Software implementation and evaluation is also done on the supercomputer "Mistral" at DKRZ. All computation was done using Mistral's standard compute nodes with two 12-core Intel Xeon E5-2680 v3 processors at 2.5 GHz and 64 GB main memory each. In principle, the data complexity is reduced from $O(n^3)$ to $O(m)$, where n is the compute grid resolution of the simulation model and m is the number of supporting point of all pathlines. Since the amount of pathlines as well as the amount of supporting points per pathline are given by the users, m is adjustable. The number of supporting points per pathline should be choosen relating to the simulation's resolution. The number of pathlines, on the other hand, should be constant in order to prevent visual cluttering. Therefore, m will somehow be scaling with n, and the overall reduction of complexity can be estimated with n^2. The evaluation of stability and scalability is done using Atmospheric Model Intercomparison Project (AMIP) runs which were used for testing the model's changes.

Number of Nodes	ICON runtime without visualization	ICON runtime with visualization
1	437 s	486 s
2	231 s	257 s
4	131 s	149 s
8	78 s	96 s
16	68 s	86 s
32	47 s	65 s

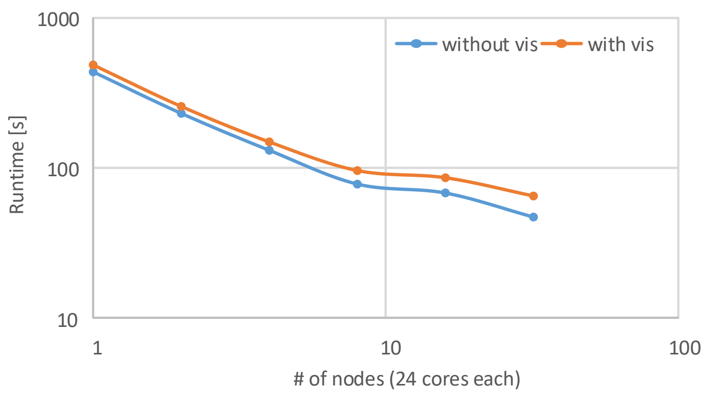


Figure 8. Time measurement of ICON ATM AMIP runs on a 20480 * 47 grid (160 km) for 7 days (1008 time steps). Simulation based on 2D domain decomposition. In-situ visualization of 2000 pathlines with 100 supporting points. Strong scaling

In fig. 8 the results of such test run are shown, comparing the overall run times of ICON with and without DVSR visualization. This case uses a low grid resolution of 160 km (20480 grid points per layer) and runs for 7 simulated days writing out 1008 time steps. Scalability tests have been done up to 32 compute nodes, which means a maximum of 768 cores. Pathline algorithms generally don't scale very well on domain decomposition, since the extraction of pathlines does not take that much computation time at all. Most time is consumed by finding the value for a given position, and this has only to be done four times per pathline assuming the usage of the Runge-Kutta 4th order. Thereby the visualization time depends mainly on the MPI communication. Rising visualization time with increasing core count is a tribute to the domain decomposition, which may lead to more or less lines to alternate processes. Up to the tested 32 nodes, the visualization does not cause exceeding increase in the overall runtime of this strong scaling test. We expect that weak scaling should show better results, since the relation between computational load and communication cost would be better in that scenario.

Beside the runtime test within ICON, we have also tested the new pathline algorithm using an artificial tornado by Crawfis on a large prism grid containing 1.44 million grid points per layer multiplied with 200 layers to get an impression on the scalability (see fig. 9). This is approximately the same amount of grid points as the ICON run on the 20 km grid has. The prism grid was generated by cutting each voxel of a regular grid into two prisms.

Number of Nodes	runtime with visualization
1	1358 s
2	442 s
4	128 s
8	26 s
16	25 s
32	21 s

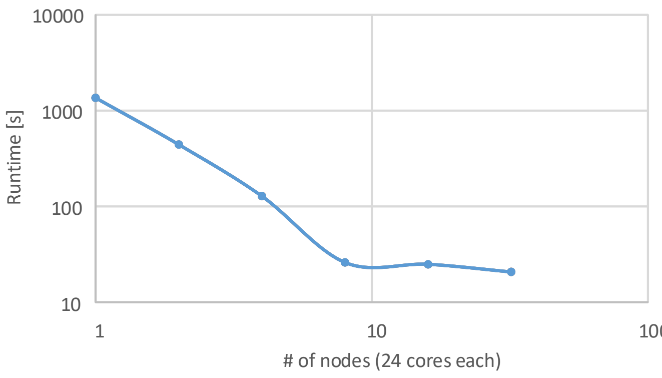


Figure 9. Visualization time measurement of an artificial tornado 1.44 million x 200 ICON grid. Simulation based on 2D domain decomposition. In-situ visualization of 2000 pathlines with 100 supporting points

The higher runtime is caused by the `DVRP_gridAPI_getValAtPos()` function scaling with the grid size. This also scales only up to 8 compute nodes, since the simulation part was not compute-intensive, and because of the strong scaling method.

Conclusion and Future Work

In order to enable in-situ visualization within the ICON climate model using our DSVR framework, we had to redesign one of its core components. With the design of the gridAPI, a fundamental generalization is introduced for the parallelized data extraction library libDVRP. This way, a support for actual and next-generation simulation models can easily be added. Also, redesign and implementation of our visualization algorithms are required. For a start, algorithms for pathline extraction as well as isosurface generation have been implemented. Both algorithms have already been tested on artificial test scenarios. The pathline extraction has been integrated in the NetCDF post-processor, as well as an in-situ visualization option in ICON.

In future work we plan to integrate the isosurface algorithm in ICON. Furthermore, large scale performance and stability evaluation of flow visualization and volume visualization will be done on high resolution scenarios.

Acknowledgement

We would like to thank Tim Rolff for supporting our work with his bachelor thesis, as well as Marco Giorgetta from Max Planck Institute for Meteorology for the successful cooperation. We also like to thank the DKRZ for providing access to their supercomputer as well as fruitful discussions. This work was supported through the Cluster of Excellence CliSAP (EXC177), Universität Hamburg, funded by the German Research Foundation (DFG).

References

1. Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., Mauldin, J.: Paraview catalyst: Enabling in situ data analysis and visualization. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization. pp. 25–29. ISAV2015, ACM, New York, NY, USA (2015), DOI: 10.1145/2828612.2828624

2. Bauer, A.C., Abbasi, H., Ahrens, J., Childs, H., Geveci, B., Klasky, S., Moreland, K., O'Leary, P., Vishwanath, V., Whitlock, B., Bethel, E.W.: In situ methods, infrastructures, and applications on high performance computing platforms. In: Proceedings of the Eurographics / IEEE VGTC Conference on Visualization: State of the Art Reports. pp. 577–597. EuroVis '16, Eurographics Association, Goslar Germany, Germany (2016), DOI: 10.1111/cgf.12930

3. Bhaniramka, P., Wenger, R., Crawfis, R.: Isosurface construction in any dimension using convex hulls. IEEE Transactions on Visualization and Computer Graphics 10(2), 130–141 (Mar 2004), DOI: 10.1109/TVCG.2004.1260765

4. Brodlie, K.W., Duce, D.A., Gallop, J.R., Wood, J.D.: Distributed cooperative visualization. In: Proceedings of Eurographics'98 State of the Art Reports. pp. 27–50 (1998)

5. Fabian, N., Moreland, K., Thompson, D., Bauer, A.C., Marion, P., Gevecik, B., Rasquin, M., Jansen, K.E.: The paraview coprocessing library: A scalable, general purpose in situ visualization library. In: 2011 IEEE Symposium on Large Data Analysis and Visualization. pp. 89–96 (Oct 2011), DOI: 10.1109/LDAV.2011.6092322

6. Haber, R.B., Mc Nabb, D.A.: Visualization Idioms: A Conceptual Model for Scientific Visualization Systems. In: Visualization in Scientific Computing (1990)

7. Jensen, N., Olbrich, S., Pralle, H., Raasch, S.: An efficient system for collaboration in tele-immersive environments. In: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization. pp. 123–131. EGPGV '02, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2002), `http://dl.acm.org/citation.cfm?id=569673.569695`

8. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. In: Proceedings of the 14th Annual Conference on Computer Graphics and In-

teractive Techniques. pp. 163–169. SIGGRAPH ’87, ACM, New York, NY, USA (1987), DOI: 10.1145/37401.37422

9. Ma, K.L., Wang, C., Yu, H., Tikhonova, A.: In-situ processing and visualization for ultra-scale simulations. Journal of Physics: Conference Series 78(1), 012043 (2007)

10. Manten, S., Breuer, I., Olbrich, S.: Parallel isosurface extraction including polygon simplification via self adapting vertex clustering. In: The Ninth IASTED International Conference on Visualization, Imaging and Image Processing (VIIP-2009). Cambridge, UK (2009)

11. Manten, S., Vetter, M., Olbrich, S.: Evaluation of a scalable in-situ visualization system approach in a parallelized computational fluid dynamics application. In: Brunnett, G., Coquillart, S., Welch, G. (eds.) Virtual Realities: Dagstuhl Seminar 2008. pp. 225–238. Springer Vienna, Vienna (2011), DOI: 10.1007/978-3-211-99178-7_12

12. McLoughlin, T., Laramee, R.S., Peikert, R., Post, F.H., Chen, M.: Over Two Decades of Integration-Based, Geometric Flow Visualization. Computer Graphics Forum (2010)

13. Olbrich, S., Pralle, H., Raasch, S.: Using streaming and parallelization techniques for 3d visualization in a high performance computing and networking environment. In: Hertzberger, B., Hoekstra, A., Williams, R. (eds.) High-Performance Computing and Networking: 9th International Conference, HPCN Europe 2001 Amsterdam, The Netherlands, June 25–27, 2001 Proceedings. pp. 231–240. Springer Berlin Heidelberg, Berlin, Heidelberg (2001), DOI: 10.1007/3-540-48228-8_24

14. Pugmire, D., Peterka, T., Garth, C.: Parallel integral curves. In: Bethel, E.W., Childs, H., Hansen, C. (eds.) High-Performance Visualization - Enabling Extreme-Scale Scientific Insight, chap. 6. CRC Press (2012), DOI: 10.1201/b12985-9

15. Vetter, M., Manten, S., Olbrich, S.: Exploring unsteady flows by parallel extraction of property-enhanced pathlines and interactive post-filtering. In: Proceedings of 14th Eurographics Symposium on Virtual Environments (EGVE 2007), Posters. pp. 9–12 (2008)

16. Vetter, M., Olbrich, S.: Scalability issues of in-situ visualization in parallel simulation of unsteady flows. In: Bischof, C., Hegering, H.G., Nagel, W.E., Wittum, G. (eds.) Competence in High Performance Computing 2010: Proceedings of an International Conference on Competence in High Performance Computing, June 2010, Schloss Schwetzingen, Germany. pp. 177–190. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), DOI: 10.1007/978-3-642-24025-6_15

17. Weigle, C., Banks, D.C.: Complex-valued contour meshing. In: Proceedings of the 7th Conference on Visualization ’96. pp. 173–180. VIS ’96, IEEE Computer Society Press, Los Alamitos, CA, USA (1996), DOI: 10.1109/VISUAL.1996.568103

18. Whitlock, B., Favre, J.M., Meredith, J.S.: Parallel in situ coupling of simulation with a fully featured visualization system. In: Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization. pp. 101–109. EGPGV ’11, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2011), DOI: 10.2312/EGPGV/EGPGV11/101-109

19. Wong, P.C., Shen, H.W., Johnson, C.R., Chen, C., Ross, R.B.: The top 10 challenges in extreme-scale visual analytics. IEEE computer graphics and applications 32(4), 63–67 (2012)

20. Wood, J., Brodlie, K., Wright, H.: Visualization over the world wide web and its application to environmental data. In: Proceedings of the 7th conference on Visualization '96. pp. 81–86. VIS '96, IEEE Computer Society Press, Los Alamitos, CA, USA (1996), DOI: 10.1109/VISUAL.1996.567610

21. Zängl, G., Reinert, D., Rípodas, P., Baldauf, M.: The icon (icosahedral non-hydrostatic) modelling framework of dwd and mpi-m: Description of the non-hydrostatic dynamical core. Quarterly Journal of the Royal Meteorological Society 141(687), 563–579 (2015), DOI: 10.1002/qj.2378

DOI: 10.14529/jsfi170304

In Situ Visualization for 3D Agent-Based Vocal Fold Inflammation and Repair Simulation

Nuttiiya Seekhao[1], *Joseph JaJa*[1], *Luc Mongeau*[2], *Nicole Y.K. Li-Jessen*[2]

A fast and insightful visualization is essential in modeling biological system behaviors and understanding underlying inter-cellular mechanisms. High fidelity models produce billions of data points per time step, making *in situ* visualization techniques extremely desirable as they mitigate I/O bottlenecks and provide computational steering capability. In this work, we present a novel high-performance scheme to couple *in situ* visualization with the simulation of the vocal fold inflammation and repair using little to no extra cost in execution time or computing resources. The visualization component is first optimized with an adaptive sampling scheme to accelerate the rendering process while maintaining the precision of the displayed visual results. Our software employs VirtualGL to perform visualization *in situ*. The scheme overlaps visualization and simulation, resulting in the optimal utilization of computing resources. This results in an *in situ* system biology simulation suite capable of remote simulation of 17 million biological cells and 1.2 billion chemical data points, remote visualization of the results, and delivery of visualized frames with aggregated statistics to remote clients in real-time.

Keywords: in situ, visualization, vocal fold, systems biology simulation, agent based modeling, tissue inflammation and repair, computational steering.

Introduction

Agent-based modeling (ABM) is a powerful and widely used approach to simulate a system consisting of interacting components or *agent*s. This form of modeling expresses a system at the microscale, and attempts to explain the emergence of higher order properties of the overall system [29]. As opposed to the analytical, equation-based approaches, the agent-based approach offers the ability to add complex behaviors to individual components or*agents*, making modeling of composite networks uncomplicated [8, 15]. In ABM, *agent*s are used to represent a wide spectrum of entities such as animals in ecosystems [13, 18, 24, 25], consumers and markets in economic models [3, 10, 36–39], and cells and proteins in biological systems [9, 11, 12, 17, 20–23, 31, 32, 34, 41]. These entities interact among themselves and with their environment (ABM *world*) in discrete time steps following a set of stochastic and/or deterministic rules. The simulation area or ABM *world* is discretized into 3D cubes called *patches.* In ABM, *agent*s can be mobile and move from *patch* to *patch.* Each *patch* maintains its states, which affect the action decision of the residing and neighboring *agent*s.

In this paper, we use the ABM simulation approach to capture tissue injury and repair at the cellular level. More specifically, we focus on the vocal fold injuries. It is estimated that voice disorders afflict 1 in 13 adults [7], and nearly 1 in 12 children [1] in the the United State annually. During phonation, human vocal folds undergo continuous biomechanical stresses. Thus, voice overuse can lead to vocal fold mucosal tissue injury that triggers complex biological processes of inflammation and repair. Voice treatments are usually prescribed to patients with voice problems [16, 27]. However, the healing outcomes of the treatment depend on the patient's initial condition and biological profile [20], making the treatment-planning process restraining and difficult for physicians and therapists. Computational medicine is a promising approach to

[1]University of Maryland, College Park, USA
[2]McGill University, Montreal, Québec, Canada

addressing this problem as it incorporates the patient's biological profile and initial conditions as parameters in determining an appropriate treatment [19, 26]. Thus, ABM offers a gateway to capturing inflammation and repair behavior to predict the outcomes of specific treatments for an individual.

The vocal fold (VF) ABM requires a high-resolution 3D grid to capture cell-cell and cell-substrate interactions with sufficient details in order to accurately predict the temporal tissue response to a provocation. Since the size of the *world* grid reflects the spatial resolution of the simulation, a high-resolution 3D VF ABM involves generation and analysis of large amounts of data. Hence, a fast and low I/O load visualization is highly desirable, making *in situ* visualization an ideal candidate for understanding the model output.

In this work, we extend our previous work on a fast GPU implementation of VF ABM simulation to incorporate *in situ* visualization at no extra cost to the overall simulation. The proposed scheme combines a data reduction technique to gain optimal visualization performance with a CPU-GPU scheduling technique to overlap simulation and visualization while hiding the execution costs of the visualization component. More specifically, our main contributions can be stated as follows.

- Reducing the amount of data analyzed, while maintaining a high fidelity visual resolution using an adaptive sampling scheme;
- An *in-situ* bio-simulation suite capable of processing 17 million biological cells and 1.2 billion chemical data points (a scale biologically representative of human vocal fold at the cellular level) as well as collecting aggregated statistics, which:
  - Completely bypasses I/O; and
  - Analyzes and renders results without causing an increase in the overall simulation time.

The rest of this article is structured as follows: section 1 introduces brief descriptions of concepts fundamental to this work including agent-based modeling (ABM), the use of ABM for inflammation and healing modeling, *in situ* ABM, and information on software and hardware platforms used. Section 2 discusses approaches taken to enhance visualization resolution, while keeping visualization performance optimized to enable a complete masking of the visualization cost, using our CPU-GPU task scheduling scheme [32, 33]. Performance and resolution enhancement evaluation is discussed in section 3.

1. Background and Related Work

1.1. Agent-Based Modeling (ABM)

The basic components of ABMs are:

- ***Agents*** - Autonomous objects that perform actions and interact with other agents and the environment;
- ***Agent Rules*** - Behaviors of each type of agents; and
- ***World*** - The environment which all agents belong to.

Multiple types of agents can be modeled in a single ABM. Each type of agents behaves according to a set of predefined rules, which can be deterministic or stochastic. For example, a simulation related to tissue inflammation may have various biological cell types, such as neutrophils, macrophages and fibroblasts, as *agent*s. The predefined rules are determined using the best available knowledge in the literature of each component of the system. The autonomous agents are mobile and make decisions based on their states and their environment. *Patch* size is uniform across the world, and thus the resolution of the simulation environment is inversely proportional to the *patch* size. The temporal dimension of ABMs is discrete and the simulation progresses in a sequence of synchronous iterations (sometimes referred to as *ticks*).

1.2. Modeling Vocal Fold Inflammation and Repair with ABM

A Vocal Fold (VF) ABM simulating inflammation and repair was developed and partially verified against empirical data [20]. The biological cells were implemented as mobile ABM *agent*s. These *agent*s perform their functionality and make action decisions based on the states of their surrounding. Briefly, at the time of injury, the damaged mucosal tissue triggers platelet degranulation [21, 22]. Platelets secrete chemicals by modifying the states of their residing *patch*es. In the inflammatory phase, these chemical gradients stimulate vasodilation and attraction of inflammatory cells, namely, neutrophils and macrophages. These inflammatory cells then get activated once they reach the wound site. Activated cells then clean up cell debris and produce more chemicals to attract cells called fibroblasts, which are responsible for tissue structure maintenance. In the healing phase, fibroblasts, activated by tissue damage, synthesize and deposit extracellular matrix (ECM) proteins such as collagen, elastin, and hyaluronans to form building blocks in tissue repair [40]. These ECM proteins then form a scaffold for supporting fibroblasts and other cells migration and wound repair activities [5].

1.3. *In-Situ* Agent-Based Modeling

In this section, we summarize the most related works. To the best of our knowledge, there has not been much *in situ* visualization work involving agent-based modeling (ABM). A quadtree-based ABM is proposed in [17] to reduce the amount of irrelevant data analyzed in-situ, where the work in [35] attempts to accomplish the same goal with a bitmap-based approach. There are tools available on Paraview [14], a popular visualization framework, which can be used for ABMs. Paraview Catalyst [4, 6] was developed to process simulation output data in-situ according to the user's co-processing script. An image-based approach built on top of Paraview Catalyst was presented in [2] to efficiently manage rendered images created in-situ by Paraview Catalyst. As much as all these works [2, 4, 6, 17, 35] reduce I/O loads, none completely by-passes I/O or can be used to achieve anything close to the desired performance for our problem.

1.4. Platforms Used

The model was implemented using C++ as the main language for speed and portability. The tasks executed on CPUs and GPUs are parallelized using Open Multi-Processing (OpenMP) application programming interface (API) for shared-memory parallel programming and Compute Unified Device Architecture (CUDA) API [28], respectively. The visualization component was implemented using Open Graphics Library (OpenGL), a cross-platform API for 2D and 3D graphics rendering. The model outputs are visualized *in situ*, and the visualized frames are delivered to the remote user via VirtualGL, an open source software which allows any Unix or Linux remote display software to run OpenGL applications by using the server's powerful 3D accelerator to perform rendering calls and send only rendered images to the client [30].

The model was tested and benchmarked on a single compute node with 44-core Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz host and two attached accelerators, NVIDIA Tesla M40.

The host has 128 GB of main memory and the accelerator has 24 GB of memory per device. The program occupies the compute node fully while running and does not allow for sharing with other processes as the program requires most of the memory available on the node.

2. A Novel Approach

2.1. Data Reduction Techniques

The processing of large amounts of data is inevitable in high-fidelity simulations. Expressive visualization enables the human eye to easily extract insightful information about the simulated system. Our model consists of approximately 17 million agents and produces 1.7 billion bio-marker data points in each iteration. The visualization component includes cell migration, chemical diffusion, and damage tracking. The most time consuming component is chemical diffusion, which needs to access 154 million points of data during each iteration. To optimize the visualization of such large amounts of data, we employ sampling techniques and study their effects on the simulation visual output and corresponding performance enhancements.

2.1.1. Constant Sampling

The first sampling approach is simply constant sampling. The world environment is divided into tiles of size $grid_x \times grid_y \times grid_z$, where the corresponding grid point represents the values of the tile centered at the point and within the radius of $grid_x/2$, $grid_y/2$ and $grid_z/2$ in the x-, y-, and z-dimension respectively. This naive approach was used to improve the visualization speed in our earlier work [33]. At the perspective of the entire simulated volume, the appearances of sampled simulation were indistinguishable for up to $(6 \times 6 \times 6)$-segment sampling. However, the output became pixelated when the view is magnified and the camera focuses on a smaller area. In particular, the wound area was not rendered with reasonable fidelity.

2.1.2. Adaptive Sampling

Adaptive sampling is used to optimize the data access while enhancing the resolution of the visual output in important areas. This sampling scheme helps keep the execution time low, and yet the details in the output presented are not compromised.

For models aiming to capture injury undergoing inflammation and repair processes, the wound site is the most active area. Therefore, the highest importance index was assigned to the wound site volume. The margin around the wound site, and the rest of the tissue are then respectively assigned less and least importance. The adaptive sampling pipeline diagram in Fig. 1 illustrates data processing steps used to sample and send data to the visualization pipeline. The program, by default, divides the whole tissue volume into three sections by first inspecting the initial wound position and size, followed by adding a margin around the wound based on user inputs, and then labeling this volume as the most active (region 1). The program lets the user specify the volume ratio between medium- and low- activity area, and splits the rest of the volume into regions 2 and 3 accordingly. As our visualization intends to highlight wound activity, the re-sampling is performed only once, thus the memory footprint is not heavily affected by the re-sampling process.

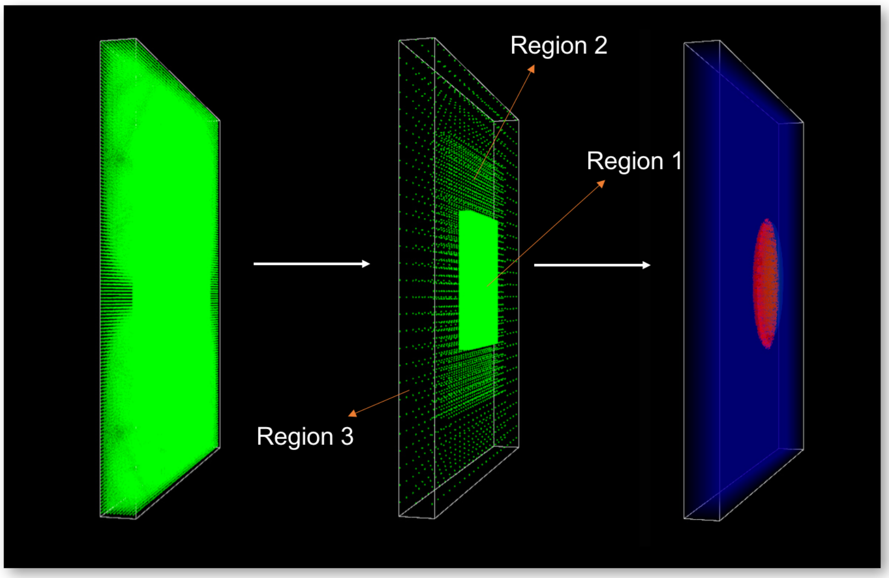


Figure 1. Adaptive sampling pipeline. From the whole set of data (left), the data are read with resolution conforming to the importance index (middle). The sampled data are then processed and sent to the visualization pipeline to be rendered (right)

2.2. Simulation and Visualization Scheduling

Given the amounts of data updated at each time step, our model requires a carefully planned scheduling mechanism for optimal performance. Accelerators such as GPUs need a CPU host, each of which has a number of cores that can be exploited using parallel programming techniques. However, whenever accelerated performance is the goal, the focus usually shifts towards GPUs due to their exceptional data parallel computation capabilities. Often, this tendency results in idle CPU cores, since their only job is to transfer data and launch GPU tasks, while the GPUs perform all the computing work. We proposed a host-device computation overlap technique in our earlier work [32], which results in state-of-the-art performance for multi-scale 2D bio-simulation ABMs. Since the 3D model is substantially more computationally demanding, we significantly extend the methods described there to achieve extremely high speed 3D simulation with *in situ* visualization.

Model operations are divided into sub-tasks and categorized as coarse or fine [32]. Since coarse-grain tasks (inflammatory cell and ECM function) are complex but less data-intensive, they are deemed CPU-suitable, where simple data-intensive fine-grain (chemical diffusion) and rendering tasks run most effectively on GPUs. The blue boxes in Fig. 2 illustrate the coarse-grain tasks that are executed in parallel on the CPU using OpenMP, where the green boxes denote the fine-grain tasks executed on the GPUs. The course-grain tasks execute on all CPU cores with the exception of $N_{GPU} + 1$ cores, where N_{GPU} denotes the number of GPUs. The rest of the N_{GPU} CPU cores are then used to dispatch fine-grain tasks and manage data to and from the GPUs, while the last core is spared to issue rendering calls to execute visualization on GPU. In [33], constant sampling was used, which was fast at the cost of lower resolution. Thus, placing the visualization execution in the GPU idle period was simple. However, in some circumstances, the user may want to focus on a smaller sections of the world, which means a sampling technique to enhance the resolution while keeping the visualization execution time smaller than GPU idle period is required. The adaptive sampling technique discussed in section 2.1.2 was introduced to solve this performance-resolution trade-offs problem.

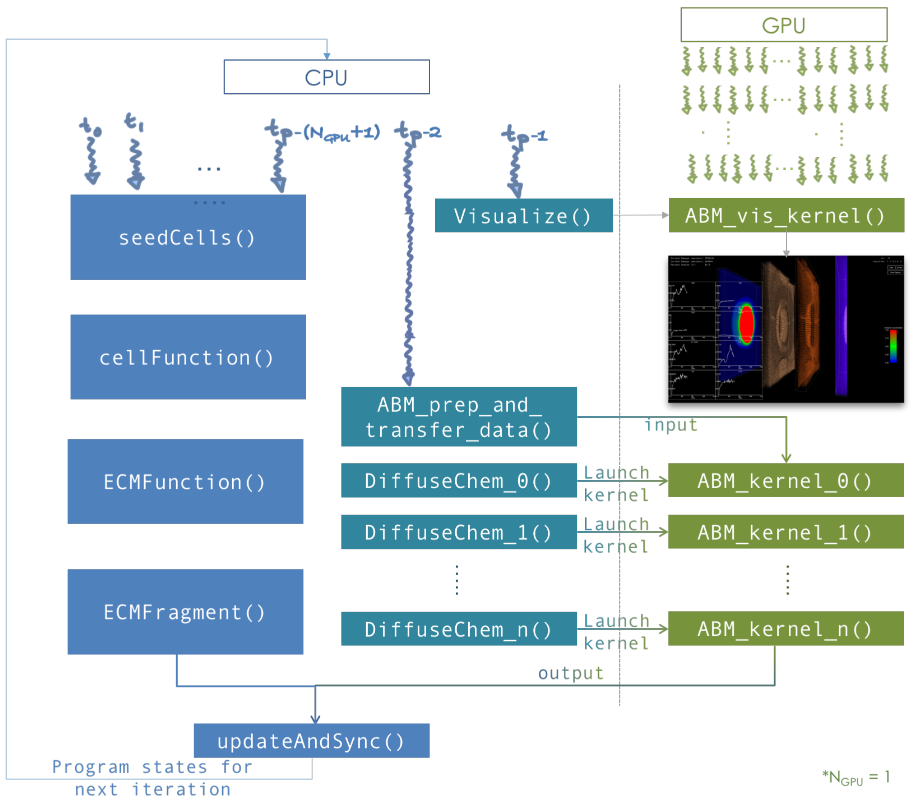


Figure 2. Diagram illustrating the scheduling and coordination of CPU-GPU computation and visualization overlap. For simplicity, this diagram is depicting a system with a single GPU. For multi-GPU, diffusion kernels launched are simply divided up and dispatched to multiple GPUs

3. Results

The results discussed below were benchmarked with 32 threads on a single compute node with 44-core Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz host and two attached accelerators, NVIDIA Tesla M40. The *in situ* simulation suite performance reported was measured with the remote user running the simulation off-site on a typical 50/50 Mbps WIFI connection. The size of the world grid is 110 x 1390 x 1006.

3.1. Visualization Component Performance

For the ABM simulation, CPU tasks (excluding updates) take about 4.7 seconds while GPU tasks only take 2.5 seconds for each iteration. Thus, there is an idle period on the GPUs waiting for the CPU to finish the coarse-grain tasks. Our goal is to make the visualization component fast enough so that the 2.2-second window gap would allow us to integrate visualization with computation on the GPUs without increasing the total execution time. As discussed earlier, our work in [33] used constant sampling, which was fast enough but did not achieve good enough resolution when zooming in areas of interest such as the wound site. It was observed that for visualization view of the entire simulated volume (Fig. 3), the appearance of sampled simulation was indistinguishable from $(1 \times 1 \times 1)$- up to $(6 \times 6 \times 6)$-segment sampling. Thus, for optimal performance, the least important areas in adaptive sampling are set to use the coarsest resolution of 6^3 sampling windows. The medium and most important areas are then sampled with finer 4^3 and 2^3 windows, respectively. With $2-4-6$-sampling resolution, as shown in Fig. 4, the resolution of the visualization improved significantly, while the execution time of the visualization component only increased to 1.9 seconds, which is still below our visualization time budget, hence no increase in the overall execution time.

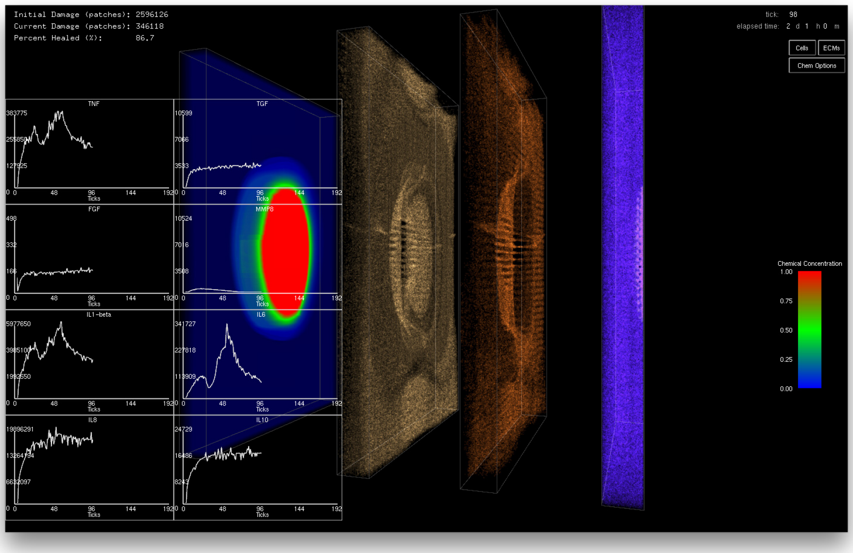


Figure 3. A screenshot of *in situ* visualization of the simulation captured at the client side. The 2D charts plot total concentration for each type of a chemical. The left most 3D volume displays the distribution of one of the eight chemicals specified by the user. The second and third volumes show macrophage (brown) and neutrophil (red) distributions respectively. The last volume on the right displays current damage (pink) and the distribution of fibroblasts (blue)

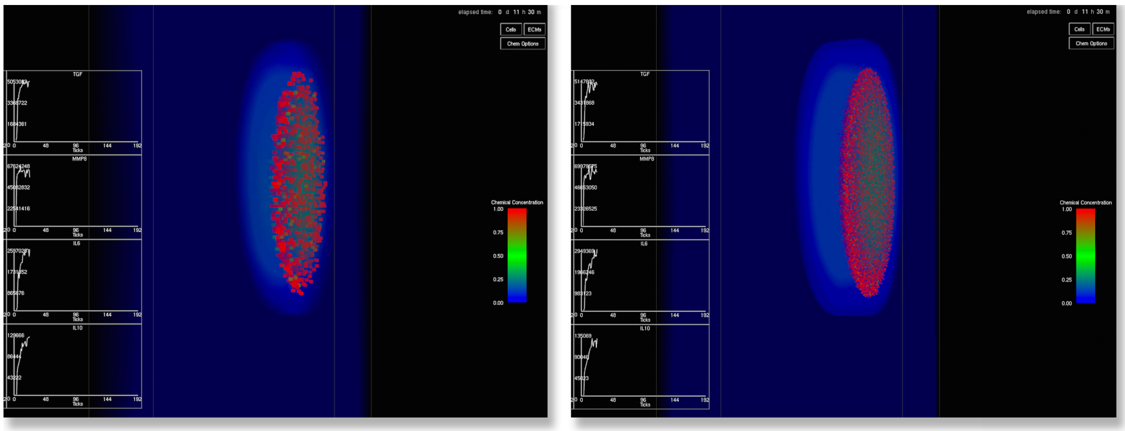


Figure 4. Screenshots comparison of 6^3 sampling windows (left) and $2-4-6$-sampling resolution (right) when zoomed in to high-activity area

3.2. Coupled Simulation and Visualization Performance

The performance of the simulation suite is shown in Fig. 5. Without sampling, visualization took 23 seconds to complete. With adaptive sampling, visualization of chemical diffusion decreased to 1.9 seconds. The visualization execution was performed during the idle period on one of the GPUs keeping the total execution time unchanged at 6.2 seconds per iteration on average. This results in the ability to run the simulation from the start of the iteration, remote computation, remote visualization to the moment the frame gets rendered on the client's machine in under 7 seconds/frame. To the best of our knowledge, this is by far the fastest known complex ABM simulation and visualization of a problem of that physiological scale.

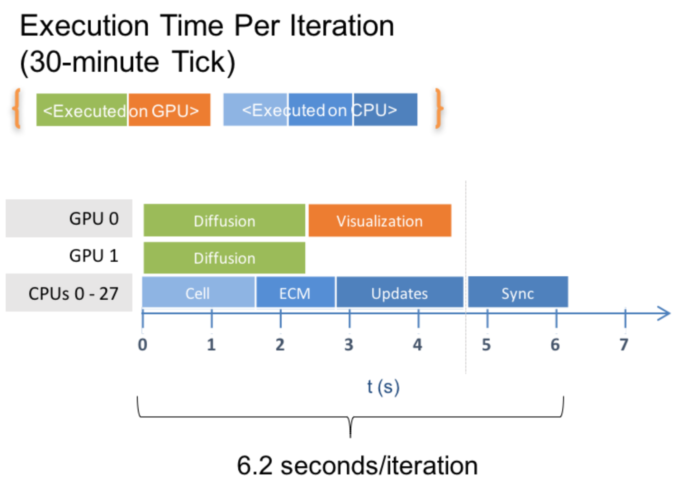


Figure 5. Simulation suite performance. Chart demonstrating overlapped visualization and computation executions on GPUs and CPUs

Conclusion

In this paper, we presented novel techniques to achieve *in situ* 3D ABM visualization at almost no cost to the overall simulation. As a result, we can simulate and render the VF inflammation and repair in real time. An effective task scheduling and management approach was used to orchestrate the execution of coarse-grain cellular functions, which are parallelized on the multi-core CPU, with the execution of fine-grain GPU tasks, including the overlapping with the in-situ visualization component. This results in optimal concurrent utilization of both multi-core CPU and GPU, including the fact that the execution time of the GPU visualization component is completely hidden behind the CPU tasks. We are able to simulate the total of 17 million inflammatory cells and 1.7 billion bio-marker data points, as well as analyze and render the same number of cells and 154 million bio-marker data points on the server and send result frames to the remote user in under 7 seconds per iteration. The model is currently being developed to incorporate more visualization functionality, which includes an integration of ECM proteins visualization and direct volume rendering into the simulation suite to give users the ability to extract meaningful information and explore the output data in different ways. The goal is to keep the overall performance of the simulation the same even with more comprehensive visualization components.

Acknowledgments

The authors would like to thank Caroline Shung, Yun (Yvonna) Li and Alireza Najafi Yazdi for their contributions to the development of the initial and base sequential model. Sujal Bista for guidance in developing the visualization component. And UMIACS staff for assistance with VirtualGL and TurboVNC configuration.

Research reported in this publication was supported by National Institute of Deafness and other Communication Disorders of the National Institutes of Health under award numbers R03DC012112 and R01DC005788. The authors gratefully acknowledge the support provided by NSF, Contract Number CNS-1429404 MRI project.

References

1. Voice, speech, and language quick statistics. `http://www.nidcd.nih.gov/health/statistics/vsl/Pages/stats.aspx` (2012), accessed: 2016-10-3

2. Ahrens, J., Jourdain, S., OLeary, P., Patchett, J., Rogers, D.H., Petersen, M.: An image-based approach to extreme scale in situ visualization and analysis. In: Damkroger, T., Dongarra, J. (eds.) SC14: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 424–434. IEEE (nov 2014), DOI: 10.1109/sc.2014.40

3. Arthur, W.B.: Chapter 32 out-of-equilibrium economics and agent-based modeling. In: Handbook of Computational Economics, pp. 1551–1564. Elsevier (2006), DOI: 10.1016/s1574-0021(05)02032-0

4. Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., Mauldin, J.: ParaView catalyst. In: Weber, G.H. (ed.) Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization - ISAV2015. pp. 25–29. ACM Press (2015), DOI: 10.1145/2828612.2828624

5. Bainbridge, P.: Wound healing and the role of fibroblasts. Journal of Wound Care 22(8), 407–412 (aug 2013), DOI: 10.12968/jowc.2013.22.8.407

6. Bauer, A.C., Geveci, B., Schroeder, W.: The paraview catalyst users guide (2013)

7. Bhattacharyya, N.: The prevalence of voice problems among adults in the united states. The Laryngoscope 124(10), 2359–2362 (may 2014), DOI: 10.1002/lary.24740

8. Bonabeau, E.: Agent-based modeling: Methods and techniques for simulating human systems. Proceedings of the National Academy of Sciences 99(Supplement 3), 7280–7287 (may 2002), DOI: 10.1073/pnas.082080899

9. Brown, B.N., Price, I.M., Toapanta, F.R., DeAlmeida, D.R., Wiley, C.A., Ross, T.M., Oury, T.D., Vodovotz, Y.: An agent-based model of inflammation and fibrosis following particulate exposure in the lung. Mathematical Biosciences 231(2), 186–196 (jun 2011), DOI: 10.1016/j.mbs.2011.03.005

10. Caiani, A., Russo, A., Palestrini, A., Gallegati, M.: Economics with Heterogeneous Interacting Agents: A Practical Guide to Agent-Based Modeling. Springer (2016)

11. Cilfone, N.A., Kirschner, D.E., Linderman, J.J.: Strategies for efficient numerical implementation of hybrid multi-scale agent-based models to describe biological systems. Cellular and Molecular Bioengineering 8(1), 119–136 (nov 2014), DOI: 10.1007/s12195-014-0363-6

12. D'Souza, R.M., Lysenko, M., Marino, S., Kirschner, D.: Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. In: Wainer, G.A., Shaffer, C.A., McGraw, R.M., Chinni, M.J. (eds.) Proceedings of the 2009 Spring Simulation Multiconference. p. 21. Society for Computer Simulation International, SCS/ACM (2009)

13. Gras, R., Devaurs, D., Wozniak, A., Aspinall, A.: An individual-based evolving predator-prey ecosystem simulation using a fuzzy cognitive map as the behavior model. Artificial Life 15(4), 423–463 (oct 2009), DOI: 10.1162/artl.2009.gras.012

14. Henderson, A., Ahrens, J., Law, C., et al.: The ParaView Guide. Kitware Clifton Park, NY (2004)

15. Janssen, M.A.: Agent-based modelling. Modelling in ecological economics pp. 155–172 (2005)

16. Johns, M.M.: Update on the etiology, diagnosis, and treatment of vocal fold nodules, polyps, and cysts. Current Opinion in Otolaryngology & Head and Neck Surgery 11(6), 456–461 (dec 2003), DOI: 10.1097/00020840-200312000-00009

17. Krekhov, A., Grninger, J., Schlnvoigt, R., Krger, J.: Towards in situ visualization of extreme-scale, agent-based, worldwide disease-spreading simulations. In: SIGGRAPH Asia 2015 Visualization in High Performance Computing on - SA '15. ACM Press (2015), DOI: 10.1145/2818517.2818543

18. Lampert, A., Hastings, A.: Stability and distribution of predator-prey systems: local and regional mechanisms and patterns. Ecology Letters 19(3), 279–288 (jan 2016), DOI: 10.1111/ele.12565

19. Li, N.Y.K., Abbott, K.V., Rosen, C., An, G., Hebda, P.A., Vodovotz, Y.: Translational systems biology and voice pathophysiology. The Laryngoscope 120(3), 511–515 (mar 2010), DOI: 10.1002/lary.20755

20. Li, N.Y.K., Verdolini, K., Clermont, G., Mi, Q., Rubinstein, E.N., Hebda, P.A., Vodovotz, Y.: A patient-specific in silico model of inflammation and healing tested in acute vocal fold injury. PLoS ONE 3(7), e2789 (jul 2008), DOI: 10.1371/journal.pone.0002789

21. Li, N.Y.K., Vodovotz, Y., Hebda, P.A., Abbott, K.V.: Biosimulation of inflammation and healing in surgically injured vocal folds. Annals of Otology, Rhinology & Laryngology 119(6), 412–423 (jun 2010), DOI: 10.1177/000348941011900609

22. Li, N.Y.K., Vodovotz, Y., Kim, K.H., Mi, Q., Hebda, P.A., Abbott, K.V.: Biosimulation of acute phonotrauma: An extended model. The Laryngoscope 121(11), 2418–2428 (oct 2011), DOI: 10.1002/lary.22226

23. Martin, K.S., Blemker, S.S., Peirce, S.M.: Agent-based computational model investigates muscle-specific responses to disuse-induced atrophy. Journal of Applied Physiology 118(10), 1299–1309 (feb 2015), DOI: 10.1152/japplphysiol.01150.2014

24. McLane, A.J., Semeniuk, C., McDermid, G.J., Marceau, D.J.: The role of agent-based models in wildlife ecology and management. Ecological Modelling 222(8), 1544–1556 (apr 2011), DOI: 10.1016/j.ecolmodel.2011.01.020

25. McLane, A.J., Semeniuk, C., McDermid, G.J., Tomback, D.F., Lorenz, T., Marceau, D.: Energetic behavioural-strategy prioritization of clark's nutcrackers in whitebark pine communities: An agent-based modeling approach. Ecological Modelling 354, 123–139 (jun 2017), DOI: 10.1016/j.ecolmodel.2017.03.019

26. Mi, Q., Li, N.Y.K., Ziraldo, C., Ghuma, A., Mikheev, M., Squires, R., Okonkwo, D.O., Verdolini-Abbott, K., Constantine, G., An, G., Vodovotz, Y.: Translational systems biology of inflammation: potential applications to personalized medicine. Personalized Medicine 7(5), 549–559 (sep 2010), DOI: 10.2217/pme.10.45

27. Misono, S., Marmor, S., Roy, N., Mau, T., Cohen, S.M.: Multi-institutional study of voice disorders and voice therapy referral. Otolaryngology-Head and Neck Surgery 155(1), 33–41 (jul 2016), DOI: 10.1177/0194599816639244

28. Nvidia, C.: Compute unified device architecture programming guide (2007)

29. Page, S.E.: Agent based models. The New Palgrave Dictionary of Economics. Palgrave MacMillan, New York (2005), DOI: 10.1057/9780230226203.0016

30. Project, T.V.: VirtualGL background. `http://www.virtualgl.org/About/Background` (2015)

31. Richmond, P., Walker, D., Coakley, S., Romano, D.: High performance cellular level agent-based simulation with FLAME for the GPU. Briefings in Bioinformatics 11(3), 334–347 (feb 2010), DOI: 10.1093/bib/bbp073

32. Seekhao, N., Shung, C., Jaja, J., Mongeau, L., Li-Jessen, N.Y.K.: Real-time agent-based modeling simulation with in-situ visualization of complex biological systems: A case study on vocal fold inflammation and healing. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE (may 2016), DOI: 10.1109/ipdpsw.2016.20

33. Seekhao, N., Shung, C., JaJa, J., Mongeau, L., Li-Jessen, N.Y.: High-resolution 3d vocal fold repair simulation using highly-parallelized agent-based modeling. submitted to PloS one (2017)

34. Shi, Z., Chapes, S.K., Ben-Arieh, D., Wu, C.H.: An agent-based model of a hepatic inflammatory response to salmonella: A computational study under a large set of experimental data. PLOS ONE 11(8), e0161131 (aug 2016), DOI: 10.1371/journal.pone.0161131

35. Su, Y., Wang, Y., Agrawal, G.: In-situ bitmaps generation and efficient data analysis based on bitmaps. In: Kielmann, T., Hildebrand, D., Taufer, M. (eds.) Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing - HPDC '15. pp. 61–72. ACM Press (2015), DOI: 10.1145/2749246.2749268

36. Tesfatsion, L.: Agent-based computational economics: modeling economies as complex adaptive systems. Information Sciences 149(4), 262–268 (feb 2003), DOI: 10.1016/s0020-0255(02)00280-3

37. Tesfatsion, L.: Agent-based computational economics: Growing economies from the bottom up. Artificial Life 8(1), 55–82 (jan 2002), DOI: 10.1162/106454602753694765

38. Tesfatsion, L.: Chapter 16 agent-based computational economics: A constructive approach to economic theory. In: Handbook of Computational Economics, pp. 831–880. Elsevier (2006), DOI: 10.1016/s1574-0021(05)02016-2

39. Tesfatsion, L., Judd, K.L.: Handbook of computational economics: agent-based computational economics, vol. 2. Elsevier (2006)

40. Velnar, T., Bailey, T., Smrkolj, V.: The wound healing process: An overview of the cellular and molecular mechanisms. Journal of International Medical Research 37(5), 1528–1542 (oct 2009), DOI: 10.1177/147323000903700531

41. Wang, Z., Butner, J.D., Kerketta, R., Cristini, V., Deisboeck, T.S.: Simulating cancer growth with multiscale agent-based modeling. Seminars in Cancer Biology 30, 70–78 (feb 2015), DOI: 10.1016/j.semcancer.2014.04.001

DOI: 10.14529/jsfi170305

Seismic Processing Performance Analysis on Different Hardware Environment

***E. O. Tyutlyaeva*[1]*, S. S. Konyukhov*[1]*, I. O. Odintsov*[1]*, A. A. Moskovsky*[1]**

In this research we have used computational-intensive software that implements 2D and 3D seismic migrations to study mini-application behavior for a set of the computational architectures. In addition to three architecture type comparative analyses, two CPU generation comparisons have been done.

The dynamic behavior of chosen mini-applications was studied using BSC performance analysis tools to identify their common features.

In summary, we observe the best performance of mini-applications on Intel Xeon E5-2698 CPU generation 4. Intel Xeon Phi 7250 peculiar architectural characteristics requires careful source code optimizations to help the compiler to effectively vectorize time-consuming loops and to improve the cache locality in order to achieve higher performance level. Elbrus-4S CPU is theoretically suitable for such kind of applications, but the currently observed performance is an order of magnitude smaller than on Xeon E5 family; we believe that the frequency and RAM bandwidth increase, as well as source code optimization work could improve its performance.

Keywords: Performance analysis, architecture comparison, seismic processing profiling, power-usage analysis, application behavior analysis.

Introduction

In this research, the suitable seismic processing mini-applications were selected with active collaborations with practitioners in seismic data analysis. These mini-applications can serve as a basis for detailed performance study of reverse time migration algorithms, which are actively used in reconstruction of under-surface Earth structure from the seismic sensor readings.

- Dynamic behavior of the chosen mini-apps is studied using the performance analysis tools to identify their common features;
- Analysis was performed for a set computational architectures, including both common architectures, such as x86, and non-standard one, VLIW; and
- Performed analysis demonstrates a scalability potential for the chosen mini-applications, and we expect more performance /speedup for these mini-applications if run on computational cluster in multi-threaded/multi-MPI processes way. That is planned for the future work.

The main emphasis was put on the computations, while mini-applications' I/O requirements, which play important role during data processing and affect total processing time, need to be investigated further.

1. Tested Applications

For performance analysis we have used the most typical seismic mini-applications, that implemented 2D and 3D seismic migrations, based on the algorithms used in practice. These applications have been chosen in cooperation with practicing researchers in this field. The source codes were provided by "GEOLAB" Company, [3].

[1]ZAO RSC Technologies, Moscow, Russian Federation

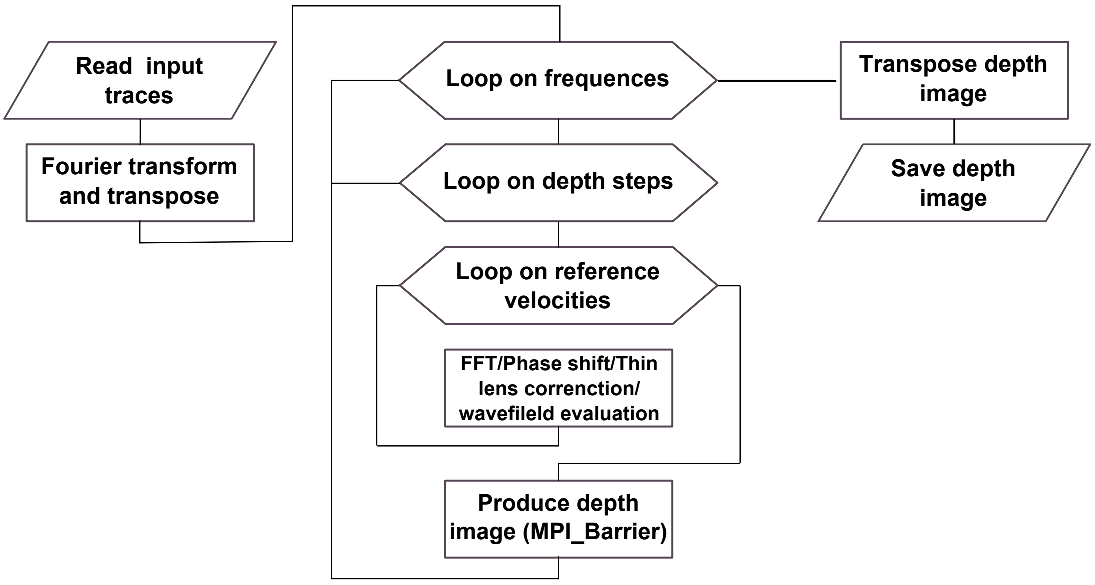

Read input traces
Fourier transform and transpose
Loop on frequences
Loop on depth steps
Loop on reference velocities
FFT/Phase shift/Thin lens correnction/ wavefileld evaluation
Produce depth image (MPI_Barrier)
Transpose depth image
Save depth image

Figure 1. Seismic migration basic flowgraph

At the same time, the applications are characterized by acceptable level of computation complexity, which allows to use different techniques for testing different computational platforms. The basic flowgraph of the seismic migration is represented on Fig. 1

2D-seismic migration application (`Wemig`, [7]) uses reverse-time wavefield continuation in frequency/space domains and depth imaging. The MPI parallel programming model has been implemented with basic auto-vectorization for Intel architectures. Input data amount for test: 206 MB.

3D seismic migration application (`Cazmig`, [2]) implements the Cazdag migration algorithm, based on 3D data migration. In this method all computations are performed in the frequency domain where the source and the receiver positions are aligned with the phase shift by the rotation operation of Fourier coefficients. The hybrid parallel programming model has been used (MPI+OMP) with basic auto-vectorization for Intel architectures. Input data amount for test: 13.4 GB.

2. Simulation Stage

At the first stage, the algorithm structure analysis has been studied using BSC Performance Analysis Tools [6], with early efforts focused on simulation of multi-node cluster.

The execution trace-files have been generated using Extrae [4], a tool for post-mortem analysis. Then a simulation tool Dimemas [1] has been used for first approximation of geological processing software and hardware interaction.

During the simulation stage, the workload intensity has been evaluated, as well as scalability limits and DRAM impact on computation performance.

As an example, the MIPS (Million Instructions Per Second) distribution during the 2D seismic migration execution on the simulated 1-node and 4-node configurations of Intel Xeon E5-2697 v3, 64 GB RAM is shown on Fig. 2. Each horizontal line represents the timelined view of each MPI rank, and the color intensity reflects the workload (darker color means more intensive computation workload, then lighter).

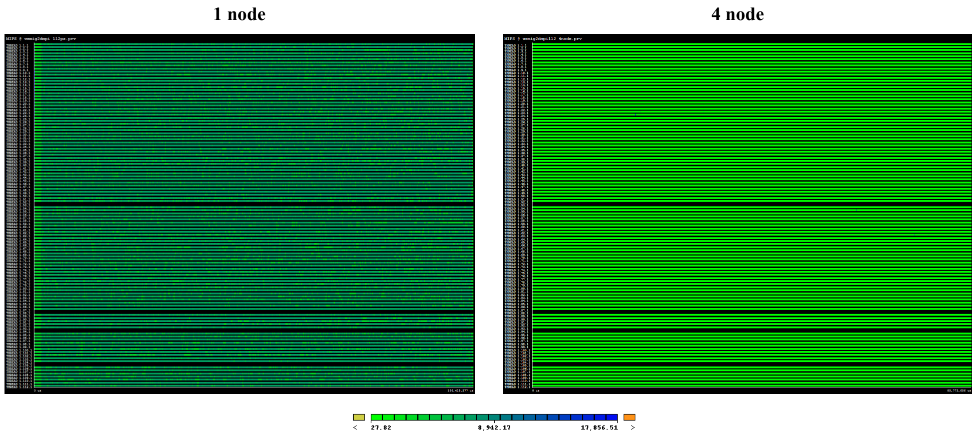


Figure 2. Tracing of MIPS during 2D seismic migration Mini-App simulation

According to the simulation result, the workload evenly distributed between all used MPI ranks during execution time. The simulated results for four similar nodes also show balanced workload and good application scalability, while computation intensity is decreased.

Further simulation and analysis with different amounts of cores and DRAM shows good scalability for 2D and 3D seismic migration and results and strong correlation between amount of DRAM and workload. Additional practical testing using different amount of RAM has been conducted, based on the simulation results.

3. Testbeds

The real cluster nodes prototypes have been chosen, based on the simulation results and the main trends in geological computations. The basic specifications of studied testbeds are listed in the Table 1.

4. Efficiency

Experiments with different numbers of used computation cores have shown the efficiency of the 2D seismic migration mini-app depending on the hardware (see Fig. 3).

The application efficiency depending of the amount of RAM is presented on Fig. 4. The dotted vertical line divides physical cores efficiency (left side of the line graph) from the hyperthreading technology efficiency, [5] (right side of the line graph, where number of threads exceed the number of physical cores).

It is worth noting, that the doubling of memory capacity leads to the significant performance increase on the Broadwell testbed, especially in the hyperthreading range, while on the Haswell testbed productivity gains are not substantial for the tested mini-apps. It seems the Broadwell cores with 64GB RAM configuration were stalled due to memory demands; the Haswell_64GB results are more balanced.

The absolute values of the execution time confirm these observations for 2D and 3D seismic migration cases (see Table 2). The provided results are the best from the tested range (for 2D seismic migration we have tested a number of MPI from range 1 .. N_cores x Hyperthreading; for 3D migration, implemented using hybrid parallelization scheme, we have tested all suitable MPI + OMP configurations in this range).

Table 1. Testbeds Specifications

Codename	CPU	# Cores	Memory	GB/Core
Haswell_64GB	Intel Xeon E5-2697 v3	2x 14	8x DRAM Micron 8GB DDR4/2133MHz	2.28
Haswell_128GB	Intel Xeon E5-2697 v3	2x 14	8x DRAM Samsung 16GB DDR4/2133MHz	4.57
Broadwell_64GB	Intel Xeon E5-2698 v4	2x 20	8x DRAM Micron 8GB DDR4/2133MHz	1.6
Broadwell_128GB	Intel Xeon E5-2698 v4	2x 20	8x DRAM Samsung 16GB DDR4/2133MHz	3.2
KNC	Intel Xeon Phi 7120D	61	SDRAM Intel 16GB GDDR5/2750MHz	0.26
KNL	Intel Xeon Phi 7250	68	MCDRAM Intel 16GB + 6x DRAM Micron 32GB DDR4/2133MHz	2.8
Elbrus	Elbrus-4C	4x 4	12x DRAM Micron 4GB DDR3/1600MHz	3

Table 2. Impact of the Amount of DRAM on the Execution Times of 2D and 3D Seismic Migration Mini-Apps

Testbed	2D Seismic Migration	3D Seismic Migration
Haswell_64GB	56 sec (56 MPI)	57 min 35 sec (14 MPI, 14 OMP)
Haswell_128GB	50 sec (56 MPI)	55 min 39 sec (14 MPI, 14 OMP)
Broadwell_64GB	55 sec (80 MPI)	56 min 13 sec (4 MPI, 16 OMP)
Broadwell_128GB	36 sec (80 MPI)	43 min 2 sec (16 MPI, 16 OMP)

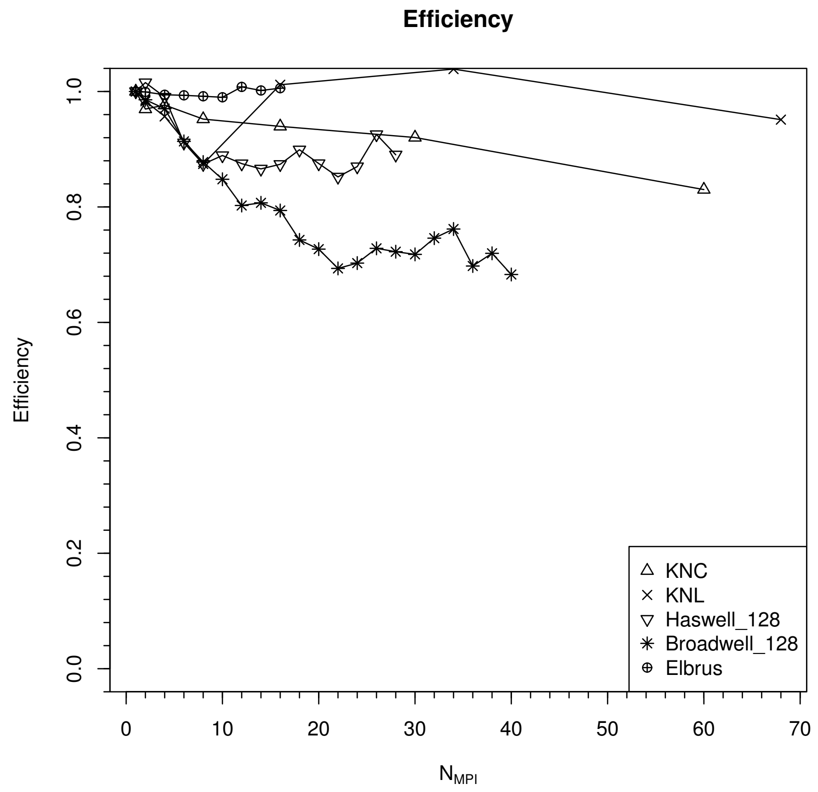
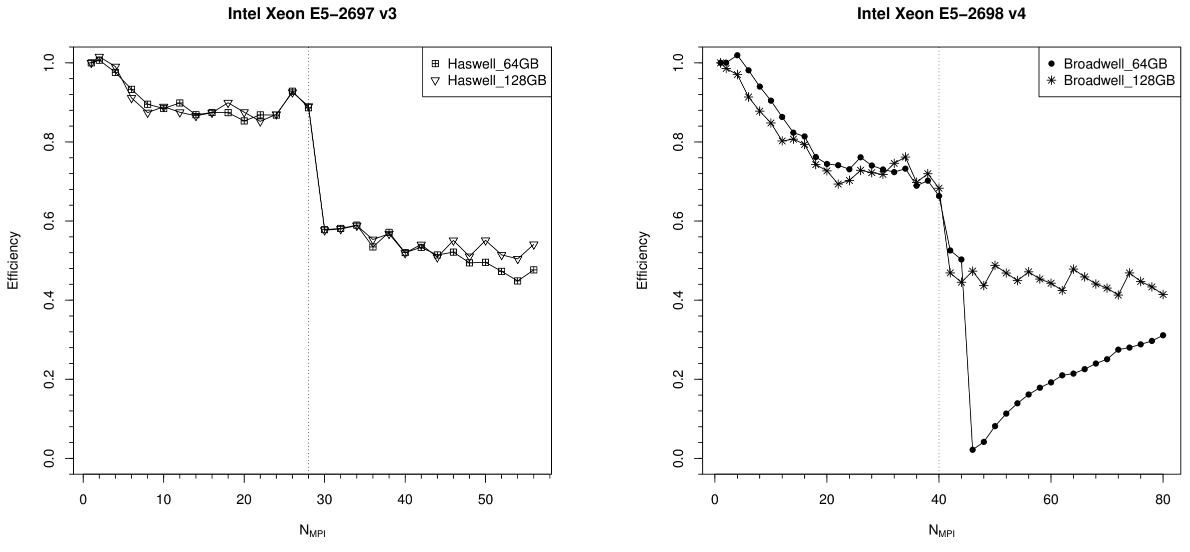


Figure 3. Efficiency of the 2D seismic migration depending on the hardware



a) b)

Figure 4. Efficiency of the 2D seismic migration depending of the amount of DRAM for a) Haswell, b) Broadwell testbeds.

However, the increase in number of MPI processes results in declining efficiency rate for all architectures despite enabled hyperthreading technology, that provides some performance increase in absolute values.

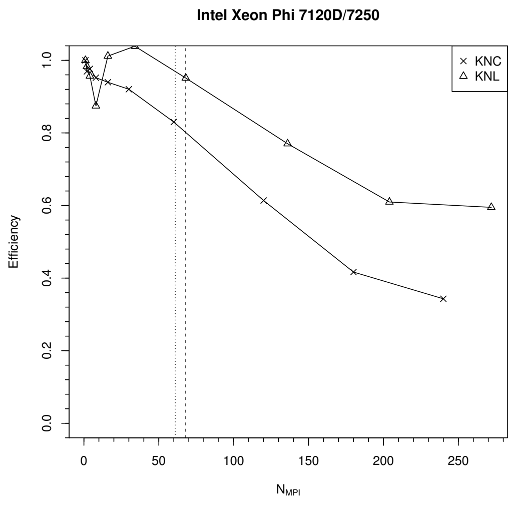


Figure 5. Efficiency of the 2D seismic migration on two generations of Intel Xeon Phi

The parallel computation efficiency for two generations of Intel Xeon PHI is shown on Fig. 5. Intel Xeon Phi architectures support hyperthreading technology for up to 4 virtual threads/core.

The resulting comparative curves show that the second generation architecture (KNL) has significant efficiency and scalability advantage over the first generation (KNC) for the tested mini-apps, especially in the hyperthreading range. The absolute values of execution times presented in Table 3 also support this observation. Numbers of MPI processes and MPI threads where carefully selected for each test run to achieve maximal performance.

Table 3. Execution Times of 2D Seismic Migration Mini-App on Two Generations of Intel Xeon PHI

Testbed	2D Seismic Migration
KNC	13 min (240 MPI)
KNL	3 min (272 MPI)

Finally, Elbrus-4S architecture demonstrates the almost perfect efficiency of up to 16 processes (i.e. up to one MPI process per computational core), because the amount of computations is high for this architecture. (Fig. 6.)

5. Tracing

The application tracing results are presented on Fig. 7 in the same way, as simulation traces, where each horizontal line represents one MPI rank. The server clients communication model is implemented, root computation rank read data and generates packages for clients to process. Client ranks are waiting and synchronizing.

The processors are highly loaded during the main computation stage, workload intensity balances for KNL and Haswell testbeds, while there is some performance swings on the Broadwell testbed due to high performance.

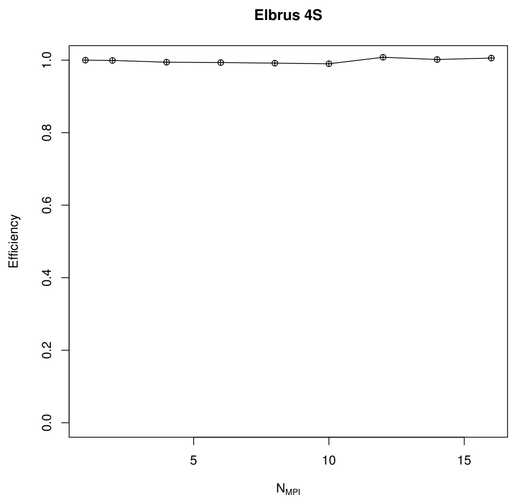
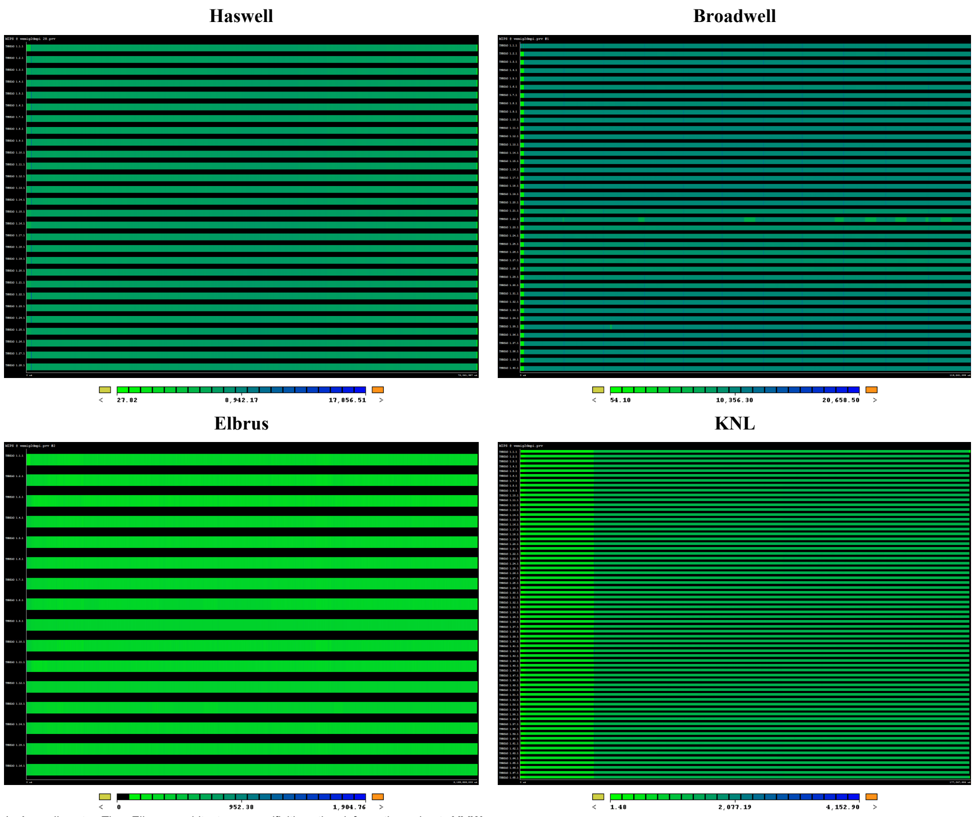


Figure 6. Efficiency of the 2D seismic migration on the Elbrus 4S

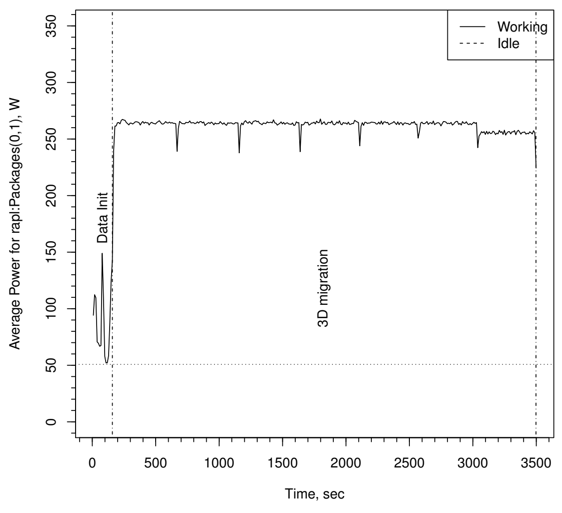


** According to The Elbrus architecture specificities the information about **VLIW instructions per cycle** were gathered instead of the standard IPC metric (Instructions per Cycle).*

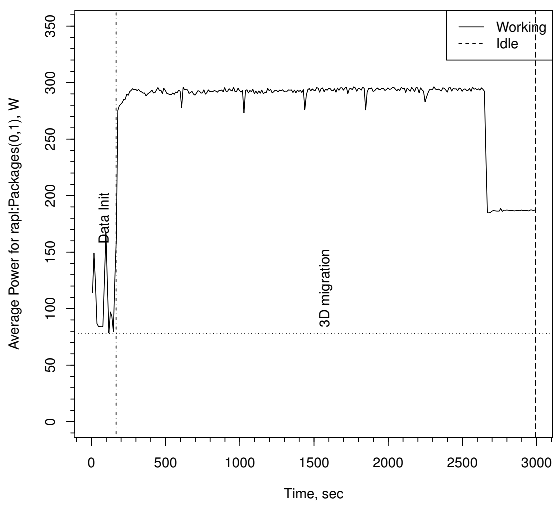
Figure 7. Efficiency of the 2D seismic migration depending on the hardware

6. Energy Consumption

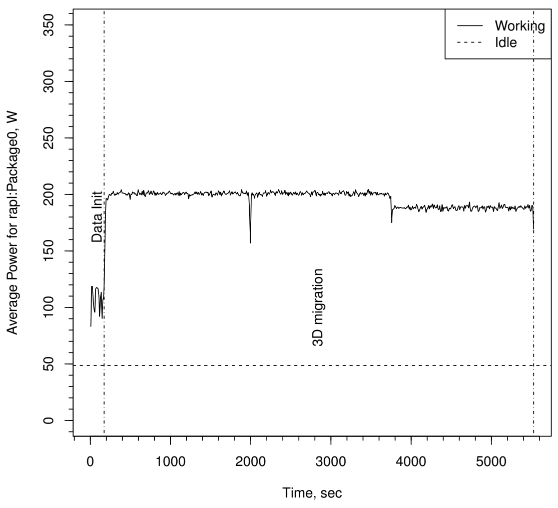
The energy consumption of 3D Seismic Migration was studied using Intel Running Average Power Limit (RAPL, [8]) counters. RAPL provides a way to measure power consumption on processor packages and DRAM. The power consumption tracing is shown on Fig. 8, where a curved line represents total power consumption, measured by RAPL*, and a dotted line represents measured average idle power consumption.



a)



b)



c)

Figure 8. Energy consumption for a) Haswell_128GB, b) Broadwell_128GB c) KNL testbeds during 3D seismic migration

Although during 3D-seismic migration test power consumption rate for Broadwell was higher than for KNL (270 W vs. 188W), total power consumption for Broadwell was lower (0.2245 kW*h vs. 0.2993 kW*h) because of substantially lesser runtime (3080 sec vs. 5741 sec). The Haswell power consumption results show intermediate values (see Table 4).

Table 4. Energy Consumption during 3D Seismic Migration Execution

	KNL	Haswell_128GB		Broadwell_128GB	
		Package0	Package1	Package0	Package1
Processor, J	967255	418726	424767	350858	92335
DRAM, J	113129	22204	25260	80799	90557
Time, sec	5741.172	3510.175		3080.837	
Total Energy, kWh	0,2993	0.2475		0.2245	

It may be noted, that the energy consumption of Haswell sockets is similar (see Table 5), while at the Broadwell testbed the first socket (processor) consumed more energy then the second one more then three times. There can be correlation with workload disbalance detected on the tracing stage. It seems, that the Broadwell_128GB testbed still has room for code optimization to achieve maximum possible performance; larger amount of computation data and more sophisticated processing models could also be used successfully.

Table 5. Energy Consumption during 3D Seismic Migration Execution

Testbed	Data	Input	Processing
KNL	Time. sec	169.62	5571.552 sec
	DRAM. J	1051	112078
	CPU. J	14980.58	952274.42
Haswell_128	Time. sec	157.32	3352.855 sec
	DRAM. J	819.86	46644.14
	CPU. J	12189.47	831303.53
Broadwell_128	Time. sec	165.29	2913.873 sec
	DRAM. J	4626.6	166729.4
	CPU, J	10389.07	339303

Conclusions

In this research we have used computational-intensive software that implements 2D (Wemig) and 3D (Cazmig) seismic migrations to study the application behavior for a set of the computational architectures. In addition to three architecture type comparative analyses, two CPU generation comparisons have been done.

For Haswell/Broadwell testbeds with similar architecture there has been a substantial (about 2x times) performance growth between generations; for the KNC/KNL testbeds the performance increase amounted up to 4x times. Moreover, there is portability issues with KNC architecture that are eliminated in KNL software stack. While the I/O overhead costs are non-essential (0.0% of overall runtime) for most studied architectures, for KNL it takes 0.73% of the runtime. KNC runtime results have worse scalability than the KNL due to lesser amount of RAM per core.

It is worth noting that the doubling of RAM memory capacity leads to the significant performance increase on the Broadwell testbed, while on the Haswell testbed productivity gains are not substantial. So the memory amount for seismic applications should be appropriate to avoid the CPU stalls. The Elbrus-4S CPUs show the best scalability while overall absolute values were lower than values for the Intel Xeons according to the theoretical performance value rates.

Average power consumption rate is the lowest for KNL and the largest for Broadwell; but total power consumption for 3D seismic migration run shows the best rates for Broadwell testbed.

In summary, it makes sense for seismic applications to use the Intel Xeon E5-2698 CPU (Broadwell) generation instead of E5-2697 (Haswell) only with large amount of RAM available; the Intel Xeon Phi (KNC/KNL) particular architectural characteristics requires careful source code optimizations to help the compiler to effectively vectorize time-consuming loops and to improve the cache locality for achieving higher performance level; The Elbrus-4S CPU is theoretically suitable for such kind of applications, but it requires the frequency and RAM bandwidth increasing, as well as sophisticated source code optimization work for achieving the best instruction-level parallelism.

Acknowledgments

This research was supported by the Russian Federal Science and Technology Program grant 14.607.21.0165 "Efficient co-design of massively parallel computer for multispectral night-time remote sensing". We thank our colleagues from GEOLAB, particularly Evgeny Kurin for the collaboration.

References

1. Dimemas: predict parallel performance using a single cpu machine ; `https://tools.bsc.es/dimemas`, accessed: 04.07.2017

2. Gazdag J. Wave Equation migration with equation migration with Phase Shift method / J. Gazdag // Geophysics - 1978 - V. 43 P. 1342–1351

3. The GEOLAB-IT Company; `http://sk.ru/net/1120173/`, accessed: 04.07.2017

4. Home: Extrae, Trace-generation package; `https://tools.bsc.es/extrae`, accessed: 04.07.2017

5. Intel Hyper-Threading Technology, `http://www.intel.ru/content/www/ru/ru/architecture-and-technology/hyper-threading/hyper-threading-technology.html`, (accessed: 29.11.2016)

6. Performance Analysis Tools: Details and Intelligence; `https://tools.bsc.es/`, (accessed: 04.07.2017)

7. Popovici A.M. Prestack migration by split-step DSR / A.M. Popovici // Geophysics - 1996 - V. 61 - P. 1412-1416;

8. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3B: System Programming Guide, Part 2, `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf, accessed:~04.07.2017`

DOI: 10.14529/jsfi170306

Exploring Scheduling Effects on Task Performance with TaskInsight

Germán Ceballos[1], *Andra Hugo*[1], *Erik Hagersten*[1], *David Black-Schaffer*[1]

Complex memory hierarchies of nowadays machines make it very difficult to estimate the execution time of tasks as depending on where the data is placed in memory, tasks of the same type may end up having different performances. Multiple scheduling heuristics have managed to improve performance by taking into account memory-related properties such as data locality and cache sharing. However, we may see tasks in certain applications or phases of applications that take little or no advantage of these optimizations. Without understanding when such optimizations are effective, we may trigger unnecessary overhead at the runtime level.

In previous work we introduced TaskInsight, a technique to characterize how the memory behavior of the application is affected by different task schedulers through the analysis of data reuse across tasks. We now use this tool to dynamically trace the scheduling decisions of multi-threaded applications through their execution and analyze how memory reuse can provide information on when and why locality-aware optimizations are effective and impact performance.

We demonstrate how we can detect particular scheduling decisions that produced a variation in performance, and the underlying reasons for applying TaskInsight to several of the Montblanc benchmarks. This flexible insight is a key for both the programmer and runtime to allow assigning the optimal scheduling policy to certain executions or phases.

Keywords: task-based scheduling, data reuse, data locality, cache model.

Introduction

Scheduling tasks in task-based applications have become significantly more difficult due to overall system complexity, particularly to the deep shared memory hierarchies. Typical approaches to optimizing scheduling algorithms consist of either providing an interactive visualization of the execution trace [1, 5] or simulating the tasks execution to evaluate the overall scheduling policy in a controlled environment [4, 9]. A developer then has to analyze the resulting profiling information and deduce if the scheduler behaves as expected, and *qualitatively* compare different schedulers.

Poor scheduling decisions can result in idle execution time due to load imbalance from the inability to prioritize tasks on the critical path or appropriate map tasks to processors. However, the scheduler decisions also impact data locality in the cache hierarchy by changing the order of tasks. The result of these decisions is performance variation across tasks that can only be understood by analyzing how the tasks share data and how the schedule affects that sharing.

Usually developers of a task-based application blame this performance degradation on data locality and attempt to characterize their workload based on data reuse without considering the dynamic interaction between the scheduler and the caches [3, 10]. This is simply because there has been no way to obtain precise information on how the data was reused through the execution of the application, such as how long it remained in the caches, and how the scheduling decisions influenced this reuse. Without an automatic tool capable of providing insight as to whether and where the scheduler misbehaved, the programmer must rely on intuition to understand and adjust the scheduler for improved performance.

In previous work we presented TaskInsight, a new methodology to characterize, in a *quantifiable* way, the scheduling process in the context of one of the most important performance-related

[1]Department of Information Technology, Uppsala University, Uppsala, Sweden

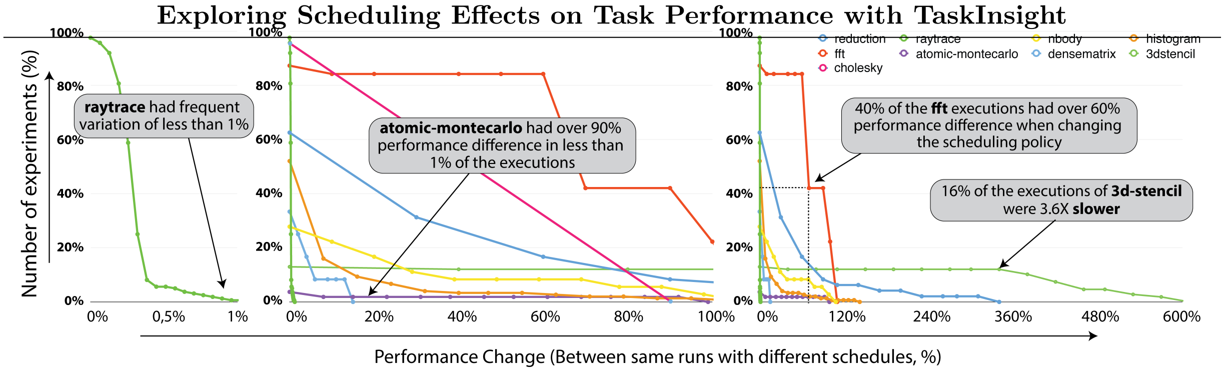


Figure 1. Distribution of the performance differences of all the experiments per benchmark. E.g. 80% of the runs of `fft` had over 20% performance difference when executed with a different schedule. All three graphs are the same but with different scales on the x-axis

characteristics: how the schedule affects data reuse between tasks through the cache hierarchy over time, which provides insight into performance of the scheduler.

In this work, we show how applying TaskInsight to the widely adopted Montblanc benchmarks reveals deep insight into why scheduling changed the memory behavior of applications, the key to understanding performance variation across different executions. The article is organized as follows. We firstly analyze all the applications with different scheduling policies and input sizes (Section 1). From all the executions, we select three specific applications, presenting case-studies to show how applying TaskInsight reveals what were the problematic issues (Section 2). Then, we cover related previous work (Section 3) to finally conclude with some remarks on how the TaskInsight analysis enables us to understand other behaviors across the benchmarks and schedulers (Conclusion).

1. Motivation

It is well-known that cache optimization is crucial for performance, but real-world applications expose different sensitivities to changes in memory behavior. Task-based applications can vary wildly in their behavior based on several factors such as the size of the input problem (total data), the number of tasks they spawn, how they distribute work among those tasks (parallelism), and how many tasks they can run in parallel (dependencies). Each of these factors can typically be controlled, either by configuring the application or by the runtime. Nevertheless, it is often difficult to know which one is the best and how would the scheduler behave with these new configurations.

Before diving into understanding of how scheduling changes the internal application's memory behavior, it is worth studying how significant the effects of scheduling can be. To do that, we begin by looking at OmpSs [6] implementations of the Montblanc Benchmarks [4]. These benchmarks were designed to cover a diverse range of task-based applications and behaviors. We selected 9 benchmarks from the suite, and executed them with over 100 different combinations of input parameters (different input datasets, number of tasks, steps, dimensions, simulations, particles, etc.), each of which we call *configuration*. Every configuration was executed with two different scheduling policies, one of which attempts to optimize for locality (`smart`, which follows the path of the spawned child tasks first) and one of which does not (`naive`, which follows a regular Breadth-First search on the spawned tasks), and we refer to them as *experiments*. These two scheduling policies are default options provided by the OmpSs framework. In order to understand how sensitive the benchmarks are to scheduling, we show how much of an impact the two schedulers have on the tasks' performance in Figure 1. Note that this metric considers the task execution time to reveal memory system effects, and therefore excludes the scheduler or the runtime overhead.

This gives us a distribution of the performance differences on a per-benchmark basis. We can see which benchmarks exposed the *largest* sensitivity to scheduling across different configurations, and how *often* these performance differences occur. The graph shows the percentage of the population (of the 100 configurations per benchmark, y-axis) as a function of slowdown (percentage between the two schedulers, x-axis). In essence, this summarizes how many of the *experiments* have a slowdown larger than X% (i.e. when changing the scheduling policy). Benchmarks such as `fft`, `cholesky`, `reduction` and `n-body` have a high variation in performance across a significant number of their configurations: 40% of the executions of `fft` have more than 60% performance difference when changing the scheduling policy; for `reduction` 30% of the executions have over 30% performance difference; and for `cholesky`, 40% have differences of over 30%.

On the other hand, `raytrace`, `dense-matrix` and `atomic-montecarlo` show negligible performance changes between the schedulers across all configurations. Interestingly, being insensitive to scheduler changes appears in two forms depending on whether the performance impact is small or the number of configurations that experience it is small. `raytrace` (green line) shows that there is a variation when changing the scheduling policy in almost every execution, however, all those differences are less than 1%, and a similar (but more noticeable) effect is observed in `dense-matrix`, with differences under 13%. On the flip side `atomic-montecarlo` shows that variation is very unlikely to occur (1 in 100 configurations), but when it does, the performance difference is significant (over 90%). These are two opposite and very extreme behaviors, but in the context of this work they have the same implication that there is little to be gained from optimizing the scheduler. For simplicity, we will classify both types as benchmarks that are performance insensitive to scheduling changes.

The results of comparing the schedulers in Figure 1 shows significant variation in how applications respond to changes in scheduling, and a substantial potential for improving performance through better scheduling. However, today's tools and techniques do not enable us to analyze and understand how they are related to scheduling, why they occur, and how to avoid them.

TaskInsight, originally presented in [2], can characterize performance differences caused by changes in memory behavior of the application *due to scheduling.* In the following section, we will select three applications that exposed performance variation (reduction, histogram and fft), and show how TaskInsight enables us to understand the reasons behind it.

2. Analyzing Performance Variation Due To Scheduling

From the benchmarks shown in Figure 1, we have selected three of the scheduling-sensitive benchmarks: `fft` (solving a Fast Fourier Transform), `reduction` (non-trivial computation over vectors and reduction of the results) and `histogram` (computing the histogram of a sample population). We will first observe the performance variation over time to identify the main differences between the two schedulers (`naive` and `smart`). We will then look at the profile data to see whether the source of the performance difference is memory, such as more L2 or L3 cache misses. Finally, we will correlate this data with TaskInsight's reuse analysis to understand how this memory behavior comes from different data reuse patterns in different schedules. Each execution took less than 10 minutes.

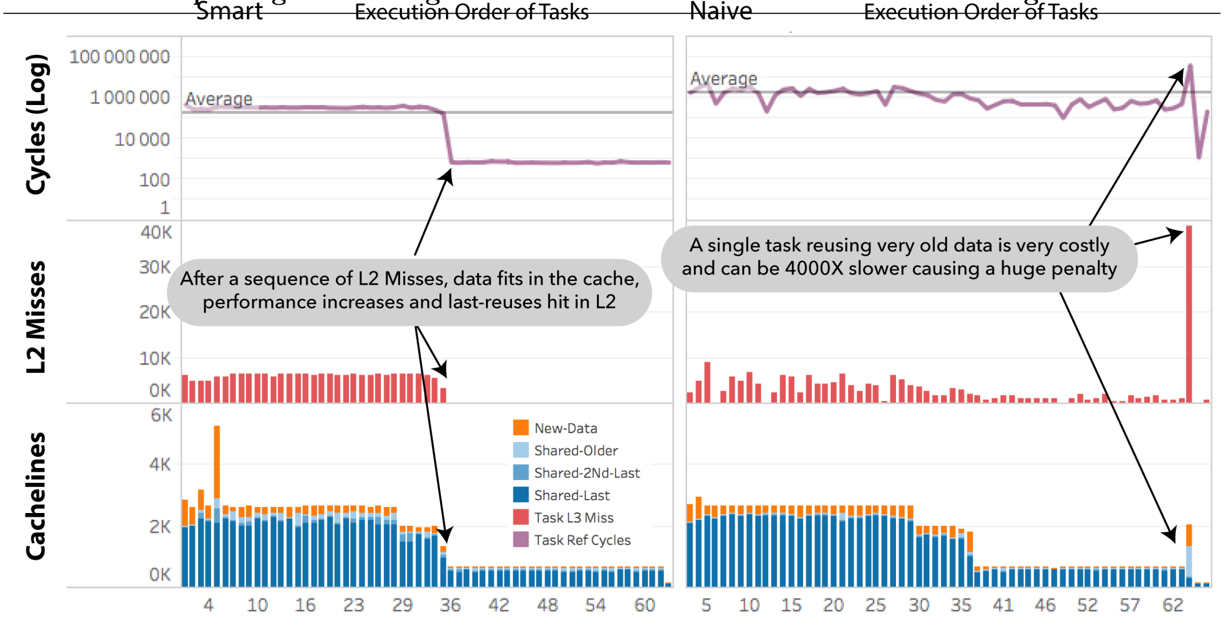


Figure 2. If data is used in a tidy pattern (`smart`, left), it fits in the cache exposing perfect reuse and incurring fewer cache misses. However, a more spread pattern (`naive`, right) has constant cache misses across the entire execution with a performance penalty. *Cacheline* refers to the number of unique cachelines touched by the task, which can be directly correlated to cache misses

2.1. Single Task with Negative Impact if Data is not in the Cache (Reduction)

An interesting case study is `reduction` (with inputs: `nelem`=8M, `nt=128`), where most of the performance differences come from missing in the L2 cache. Data for L3 is not displayed because there is no relevant change across schedules. From Figure 2 we can see that in `smart`, there is a cycle drop after executing the first 36 tasks, and it corresponds to a drop in L2 misses. From TaskInsights reuse analysis we see that data is still being reused, and has been reused very recently (many `last-reuses`), meaning that the data is present in L2. On the other hand, `naive` has a lower number of L2 misses but they are spread across the entire execution, causing a more constant average cycle count. From the TaskInsight reuse analysis we see that data has also been reused recently, but in a different pattern.

The interesting singularity that we observe is the task number 63 which brings a significant amount of new data. In `smart`, this task was scheduled in the very beginning (5th), and it is both touching a large set of new data and reusing very recent data. In `naive` this task, because of dependency flexibility, was scheduled much later (63rd), and by that time the data it has to reuse is much older as we see from TaskInsight's classification. As its reuse is not much further away, the task generates roughly 10x more L2 misses (also 10x more L3 misses) and its execution now is 400X slower than in `smart` (note the Log scale). We are able to draw these type of conclusions because OmpSs assigns unique tasks IDs to the tasks at initialization time. Thus, we can identify how the same tasks were scheduled differently.

This is an interesting example of an application that exposes two fascinating effects. First, if data is used in a *tidy* pattern, it might fit in the cache, delivering good reuse and incurring many fewer cache misses in contrast to a more spread-out pattern that has constant misses across the entire execution. Second, a single task can suffer a significant performance loss by being scheduled further away from tasks with which it shares data, thereby creating a significant penalty to the

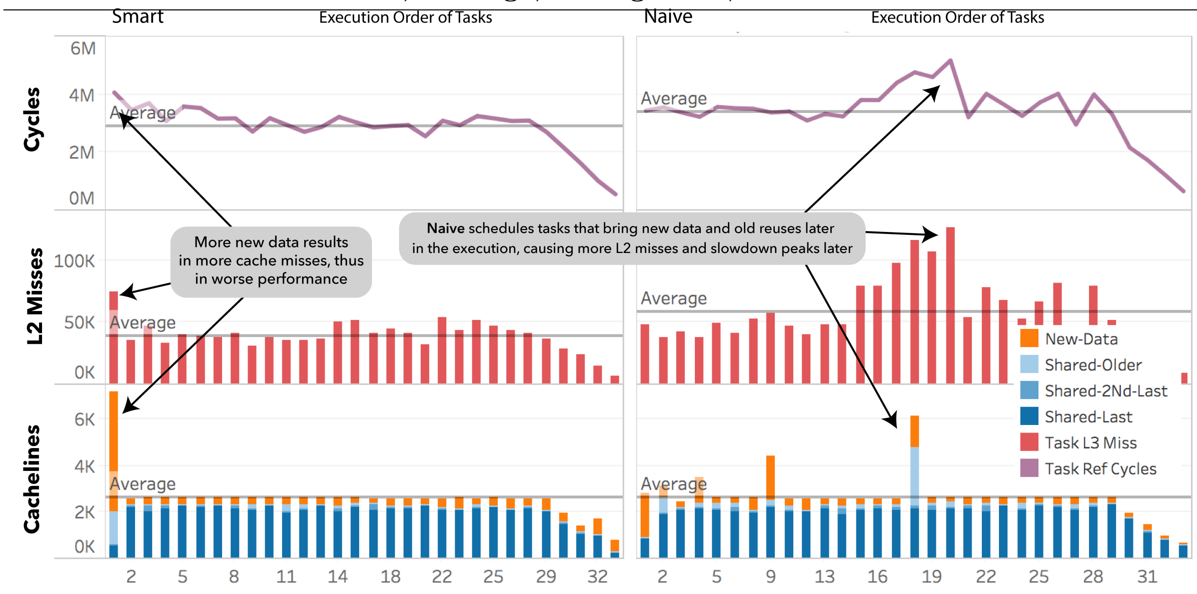


Figure 3. Scheduling bigger tasks sooner (`smart`, left) will bring a larger amount of `new-data` sooner, exposing more `last-reuses` between subsequent tasks

applications overall performance. In order to understand these issues we need TaskInsight's data usage classification combined with the performance profile.

2.2. Differences in Bringing New Data Sooner or Later (Histogram)

Another example where temporal locality of data at the private caches matters significantly is the benchmark `histogram` (with inputs: `nelem`=4M, `nbins=256`, `nt=128`), Figure 3. The largest slowdown we observe in a spike in the cycle count for `naive`, due to an increase in L2 misses (creating a difference of over 20% across the two schedules).

TaskInsight shows *what* the problem is (the reuse of old data and use of new data) and *where* it comes from (the 18th task). Task 18 was executed as number 1 in `smart`, and has a particularly large data set. By bringing its data in first, it enables that data to be installed in the cache and we see reuse by subsequent tasks. On the other hand, if this task is scheduled later, as in `naive`, it will reuse many smaller portions of already executed tasks (with smaller data sets) which were very likely evicted from the cache. This causes a domino-effect that affects several tasks creating a substantial difference in the overall cycle count.

TaskInsight not only shows the importance of scheduling a task sooner or later because of its consequences, but it also allows us to detect which specific tasks have this type of behavior and which other tasks are affected, enabling new insight into the scheduler's behavior.

2.3. Reusing Old Data Spikes Last Level Cache Misses (FFT)

In `fft` (with inputs: `nelem`=1M, `bs-tr=64`, `bs-fft=128`) there is an interesting effect worth studying at the L3 cache level. We use the multithreaded capability of TaskInsights to analyze this. As we see from Figure 4, both schedulers show two spikes with a sudden slowdown in cycles at co-running sets 35 and 73, which are correlated with spikes in L3 misses. The number of L2 misses is not displayed in this case, for simplicity, but the trend is similar to L3.

The two spikes in the total cycles (performance, top) align with an increase in L3 misses (middle). By looking at TaskInsight's data classification (bottom), we can see that at those places where the application is missing more, it is also reusing older data, which is likely not

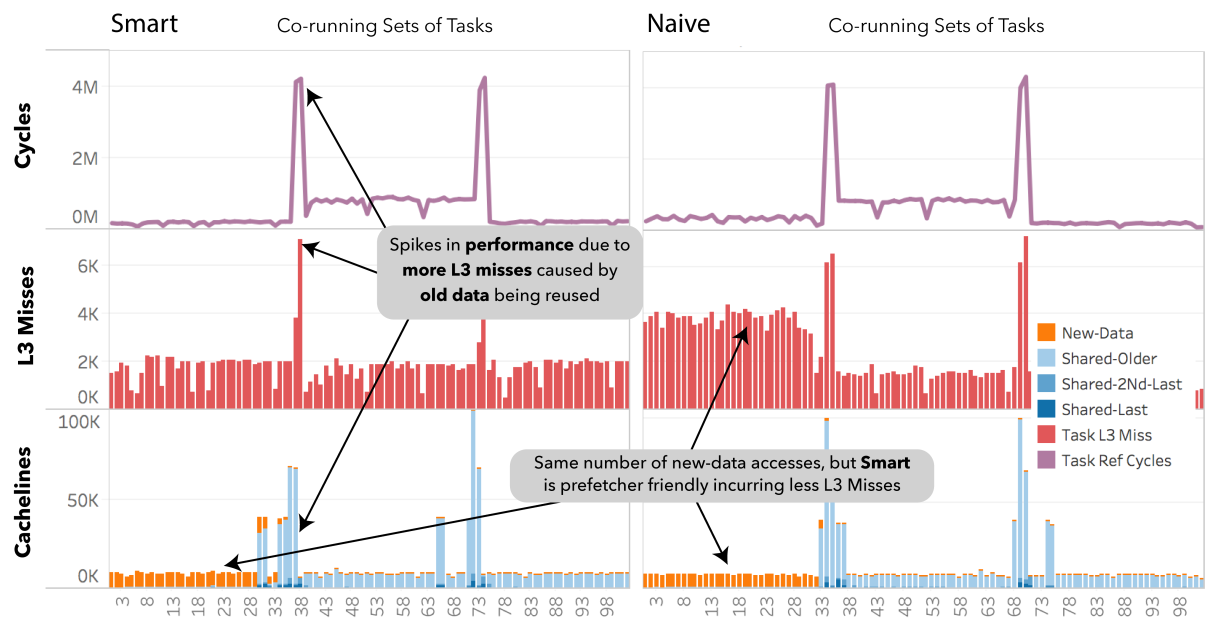


Figure 4. TaskInsight analysis of `fft`, where reuse of older data incurred in more L3 cache misses, causing two spikes in the tasks cycle count

present in the cache. By correlating this data over time, we can see that this is caused by the fact that tasks executed together at these times are not scheduled to reuse data effectively.

By comparing the two schedules, we see that `naive` has over 10% worse performance. When looking at the cycle count over time and L3 misses, we see that during co-running sets 0 to 33, it incurred in twice as many L3 cache misses, explaining where the overall performance difference is coming from.

Even though we can observe this performance and memory behavior directly with performance counters, is is only by adding TaskInsights data classification that we can understand that the extra cycles seen in `naive` is *only* from `new-data` accesses which should be coming from the main memory. This might seem counter-intuitive, as the reuse analysis is almost identical in both schedulers, but other memory factor (not yet included in this model) like prefetching fit as a very good candidate responsible for the performance difference.

3. Related Work

Multiple visualization tools [1, 5, 7] propose diagnosing scheduling anomalies by summarizing and averaging information provided by both the runtime and the hardware performance counters. However, when certain tasks end up being executed in a certain order and with different performance, it is up to the programmer to reverse-engineer the scheduler's decisions, the reasons behind them, and the moment in time where these decisions happen. With TaskInsight we analyze the memory reuse and show the reasons and the exact points in time that trigger performance variation.

Stanisic et al. [9] simulate tasks' execution in order to isolate the scheduler's effect on performance from tasks' unpredictable behavior. In our work we execute the entire application on real hardware and we characterize the interaction between the scheduler and the tasks, which triggers online scheduling decisions and performance variation.

The *memory reuse* metric has been previously used [3, 10] to analyze spatial and temporal locality of the application independent of the architecture. We propose correlating the tasks efficient use of the memory with performance variation by detecting when and why an online scheduling decision is taken. Unlike [8] we look at the reused data throughout the execution and provide an analysis over time.

Conclusion

In this work we have applied TaskInsight, a methodology that provides high-level, quantifiable information that ties task scheduling decisions to how tasks reuse data and the resulting task performance, to diagnose the reasons behind performance differences across different runs of the Montblanc applications. By combining schedule independent memory access profiling (to classify how data is reused between tasks) and schedule specific hardware performance counter data (to determine performance on a given system) we are able to identify *which* scheduling decisions impact performance, *when* they happen, and *why* they cause a problem.

TaskInsight not only shows how a scheduler can be improved, but also gives explanations for why tasks of the same type can demonstrate drastic variation in performance (up to 60% in our examples). With this, programmers can now quantitatively analyze the behavior of the scheduling algorithm and the runtime can use this information to dynamically make better decisions, for example, by using a saved profile of the memory behavior of each task to determine what to schedule next.

Our analysis exposed scheduler-induced performance differences of above 10% due to 20% changes in data reuse through the private caches and up to 80% difference data reuse through the shared last level cache. By providing this insight into the coupling between the schedule's behavior, data reuse through the cache hierarchy, and the resulting performance, we lay the groundwork for improving scheduling policies.

Acknowledgments

This work was supported by the Swedish Research Council, the Swedish Foundation for Strategic Research [project FFL12-0051] and carried out within the Linnaeus Centre of Excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

References

1. Bell, R., Malony, A.D., Shende, S.: Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In: Proceedings of the 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003 (2003), DOI:10.1007/978-3-540-45209-6_7

2. Ceballos, G., Grass, T., Hugo, A., Black-Schaffer, D.: Taskinsight: Understanding task schedules effects on memory and performance. In: Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores. pp. 11–20. PMAM'17, ACM, New York, NY, USA (2017), DOI:10.1145/3026937.3026943

3. Cheveresan, R., Ramsay, M., Feucht, C., Sharapov, I.: Characteristics of workloads used in

high performance and technical computing. In: Proceedings of the 21th Annual International Conference on Supercomputing, ICS 2007, Seattle, Washington, USA, June 17-21, 2007 (2007), DOI:10.1145/1274971.1274984

4. Chronaki, K., Rico, A., Badia, R.M., Ayguadé, E., Labarta, J., Valero, M.: Criticality-aware dynamic task scheduling for heterogeneous architectures. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015 (2015), DOI:10.1145/2751205.2751235

5. Drebes, A., Pop, A., Heydemann, K., Cohen, A.: Interactive visualization of cross-layer performance anomalies in dynamic task-parallel applications and systems. In: 2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016. IEEE Computer Society (2016), DOI:10.1109/ISPASS.2016.7482102

6. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters 21(2) (2011), DOI:10.1142/S0129626411000151

7. Müller, M.S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E.: Developing scalable applications with vampir, vampirserver and vampirtrace. In: Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany, 4-7 September 2007 (2007)

8. Pericàs, M., Amer, A., Taura, K., Matsuoka, S.: Analysis of data reuse in task-parallel runtimes. In: High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation - 4th International Workshop, PMBS 2013, Denver, CO, USA, November 18, 2013. Revised Selected Papers (2013), DOI:10.1007/978-3-319-10214-6_4

9. Stanisic, L., Thibault, S., Legrand, A., Videau, B., Méhaut, J.: Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. Concurrency and Computation: Practice and Experience 27(16) (2015), DOI:10.1002/cpe.3555

10. Weinberg, J., McCracken, M.O., Strohmaier, E., Snavely, A.: Quantifying locality in the memory access patterns of hpc applications. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing. SC '05, IEEE Computer Society, Washington, DC, USA (2005), DOI:10.1109/SC.2005.59

DOI: 10.14529/jsfi170307

From Processing-in-Memory to Processing-in-Storage

***Roman Kaplan*[1]*, Leonid Yavits*[1]*, Ran Ginosar*[1]**

Near-data in-memory processing research has been gaining momentum in recent years. Typical processing-in-memory architecture places a single or several processing elements next to a volatile memory, enabling processing without transferring data to the host CPU. The increased bandwidth to and from volatile memory leads to performance gain. However processing-in-memory does not alleviate von Neumann bottleneck for big data problems, where datasets are too large to fit in main memory.

We present a novel processing-in-storage system based on Resistive Content Addressable Memory (ReCAM). It functions simultaneously as a mass storage and as a massively parallel associative processor. ReCAM processing-in-storage resolves the bandwidth wall by keeping computation inside the storage arrays, without transferring it up the memory hierarchy.

We show that ReCAM based processing-in-storage architecture may outperform existing processing-in-memory and accelerator based designs. ReCAM processing-in-storage implementation of Smith-Waterman DNA sequence alignment reaches a speedup of almost five over a GPU cluster. An implementation of in-storage inline data deduplication is presented and shown to achieve orders of magnitude higher throughput than traditional CPU and DRAM based systems.

Keywords: Content Addressable Memory, Associative Processing, In-Storage Processing, Memristors.

Introduction

Until the breakdown of Dennard scaling designers focused on improving performance of a single core by increasing instruction level parallelism. In recent years, as Dennard scaling slowed down but Moore's law endured, the focus has shifted to improving parallelism by increasing the number of cores in multicore processors [16]. However, memory bandwidth does not improve at the same rate, making von Neumann bottleneck one of the main performance limiting factors.

Data is typically fetched to CPU's main memory from a non-volatile storage such as hard disks or Flash SSDs. Consequently, storage bandwidth and access time pose a major constraint to performance improvement. The problem worsens in datacenter cloud environment, where datasets are distributed among multiple nodes across the datacenter. In such case, data transfer adds latency and reduces bandwidth even further, lowering the performance upper bound.

This challenge has motivated renewed interest in Near-Data Processing (NDP) [7]. The main premise of NDP is shifting computing closer to data. NDP seeks to minimize data movement by computing at the most appropriate location in the memory hierarchy, which can be cache, main memory or persistent storage. With NDP, less data needs to be transferred through levels of hierarchy, thus alleviating the limited bandwidth problem. Placing computing resources at the cache level or in main memory (also known as Processing-in-Memory or PiM) does not address the emerging big data problems, where datasets are too large to fit in main memory.

Resistive CAM (ReCAM), a storage device based on emerging resistive materials in the bitcell with a novel non-von Neumann Processing-in-Storage (PRinS) compute paradigm, is proposed in order to mitigate the storage bandwidth bottleneck of big data processing. Section 1 provides background on the basic concepts of ReCAM and PRinS and covers related work. Section 2 presents the ReCAM architecture, explains how processing is performed within ReCAM and

[1]Israel Institute of Technology, Haifa, Israel

establishes its scalability. PRinS implementations of two algorithms are presented in Sections 3 and 4 and compared to other approaches: Smith-Waterman DNA sequence alignment and in-storage data deduplication.

1. Background and Related Work

Three basic concepts underline the proposed ReCAM: content addressable memories, associative processing and resistive materials. The following two subsections introduce each concept. The third subsection covers related work on NDP and highlights their limitations at addressing the storage bandwidth challenge of big data processing.

1.1. Content Addressable Memory and Associative Processing

Content addressable memory (CAM), also called associative memory, allows the comparison of all data words to a key in parallel, tagging the matching words, and possibly reading some or all of the tagged words, one by one. Standard memory read and write operations of a single word at a time can also take place. In addition to storing information, a CAM array can be modified to function as an associative processor [18] [47]. In associative processing, the parallel compare and parallel write operations supported by CAM are used to implement an "if condition, then value" expression. Thus, complex Boolean expressions are evaluated in parallel on all data words (CAM rows) by sequential execution of truth table if-then lines. Each (multi-bit) argument of a truth table line is matched with the contents of the appropriate field in the entire CAM array: the matching rows are tagged, and the corresponding result values from the truth table line are written into the designated fields of all tagged rows. For an m-bit argument x, any Boolean function $f(x)$ has 2^m possible values, therefore, the associative computing operation should incur $O(2^m)$ cycles, regardless of the dataset size. More efficiently, arithmetic operations can be performed on ReCAM in a word-parallel, bit-serial manner, reducing compute time from $O(2^m)$ to $O(m)$. The massive parallelism of each operation compensates in performance for the relatively large number of (parallel execution) cycles of each arithmetic operation. More complex computations (more than Boolean functions) are decomposed into series of Boolean expressions [18] [47].

1.2. Resistive Memories

Resistive memories store information by modulating the resistance of nanoscale storage elements (memristors). Memristors are two-terminal devices, where the resistance of the device is changed by the electrical current or voltage. The resistance of the memristor is bounded by a minimum resistance R_{on} (low resistive state, logic "1") and a R_{off} maximum resistance (high resistive state, logic "0").

Resistive memories are non-volatile, free of leakage power, and emerge as long-term potential alternatives to charge-based memories, including NAND flash. The metal-oxide resistive random access memory (ReRAM), employing one resistive device and possibly also one transistor (1R1T) per bit-cell, is considered a potential technology to replace next-generation nonvolatile memories. Its main features are high reliability and fast access speed. A test-chip of 32GB device with two ReRAM-based memory layers and a CMOS logic layer underneath has been developed [32], demonstrating design techniques to achieve a high density functional chip.

1.3. Related Work

While processing-in-storage research is relatively young, the wider concept of near-data processing, focusing mainly on processing in memory (PIM) has been thoroughly researched. The concept of mixing memory and logic has been around since 1960s. The DAPP, STARAN, CM-2, and GAPP computer architectures [39] used large number of processing units positioned in proximity to memory arrays to implement a massively parallel SIMD computer. Gokhale et al. [21] designed TeraSys, a computer architecture comprising a conventional host processor where at least part of its memory was replaced by a PIM array, integrating memory and ALUs in close proximity. Hall et al. [24] developed DIVA, the Data-Intensive Architecture, combining PIM memories with external host processors and performing selected computations in processing elements near memory and reducing the volume of data transferred across the long and slow processor-memory interface. Kogge et al. [30] developed HTMT, a parallel multilevel memory architecture, where each RAM level is a PIM memory (memory blocks interconnected with ALUs). Suh et al. [43] introduced a SLIIC QL computer featuring a processor integrated on the same die as DRAM. Lipovski et al. [31] developed a dynamic associative access memory architecture that combined DRAM and a single-bit processing element capable of associative and conventional arithmetic processing, placed in the sense amplifier area of a DRAM. Yavits et al. [48] suggested replacing the last level cache and the vector co-processor of a conventional high-performance CPU by an associative processor, which is a PIM accelerator, combining data storage and massively parallel SIMD processing capabilities. Nitin et al. [35] introduced RowCore, a near-memory processing architecture for Big Data Machine Learning. Gao et al. [19] proposed Heterogeneous Reconfigurable Logic, a reconfigurable array for near-data processing systems.

1.3.1. 3D Processing-in-Memory Architectures

While embedding processing with conventional 2D DRAM chips is less practical, recent advancement in 3D memory and logic stacking technology may remove this obstacle. Citing severe bandwidth limitations in conventional computer architecture as datasets continue to grow, Ahn et al. [1] introduced Tesseract, a 3D Processing-in-Memory accelerator for large-scale graph processing. In another work, Ahn et al. [2] developed a hybrid-memory-cube based framework that automatically decides whether to execute PIM operations in memory or processors depending on the locality of data. Nair, Sura et al. [44] [34] introduced the Active Memory Cube, a heterogeneous computing system including general-purpose host processors and specially designed in-memory processors that would be integrated in a logic layer within 3D DRAM memory. In another work, Gao et al. [20] developed hardware and software of a 3D stack memory and near-data processing architecture for in-memory analytics frameworks, including MapReduce, graph processing, and deep neural networks. Azarkhish et al. [5] developed Smart Memory Cube and designed a high bandwidth interconnect to serve the bandwidth demand of PIM architecture. Zhang et al. [49] explored PIM implemented via 3D die stacking. Akin et al. [3] addressed the issue of data reorganization in 3D stacked near-data processing architecture, introducing HAMLeT, a mechanism for host interference, bandwidth allocation, and in-memory coherence. Farmahini-Farahani et al. [17] proposed NDA, a near-DRAM acceleration architecture that processes data using accelerators 3D-stacked on DRAM devices.

1.3.2. Processing-in-Memory with Resistive Materials

Recently, emerging memory technologies such as resistive memory have become a focus of PIM research. Paul et al. [38] developed MBARC, a resistive crossbar in-memory LUT-based processing architecture. Chi et al. [10] introduced PRIME, a PIM accelerator of neural network applications in RRAM-based main memory. Yavits el al. [47] introduced a resistive CAM-based massively parallel accelerator. Shafiee et al. [40] developed an in-situ processing architecture, where memristor crossbar arrays are used to perform dot-product operations in an analog manner.

1.3.3. Near-Data Processing-in-Storage

Flash-based SSD allowed for increased in-storage bandwidth, enabling data port from each chip thus achieving higher data throughput. Typical processing-in-storage architecture places a single or several processing cores inside the storage and allows data processing without transferring it to the host processor. The concept of near-data processing-in-storage is illustrated in Fig. 1a.

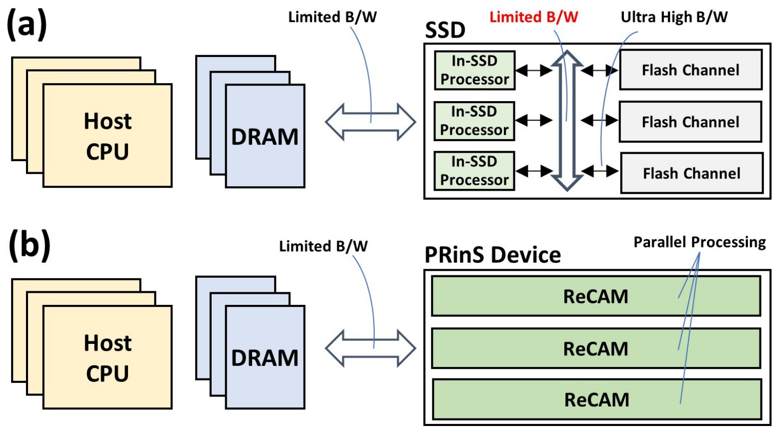


Figure 1. Comparison of (a) **near-data** processing-in-storage (b) and **in-data** processing-in-storage based on ReCAM

Boboila et al. [8] proposed Active Flash, a processing in solid-state storage that expedites data analysis by migrating the data to the flash device. The authors explored energy and performance trade-offs of their processing-in-storage architecture. Bae et al. [6] introduced the notion of Intelligent SSDs, exploring the design considerations and examining their potential benefits in data mining applications. Continuing the work on Intelligent SSD, Jo et al. [25] studied optimal ways of combining CPU, GPU and SSD for efficient processing of data-intensive algorithms. Cho et al. [11] cited the lack of parallel processing abilities in earlier in-SSD processing architectures and proposed integrating a GPU, providing API sets based on the MapReduce framework. Kang et al. [27] introduced the Smart SSD model, which combines in-SSD processing with a powerful host system, and constructed a Smart SSD prototype. De et al. [14] introduced the FPGA-based Minerva, which executed application-specific operations in the NVM controller. Jun et al. [26] introduced and constructed BlueDBM, combining a flash based storage with in-store processing capability and a low latency high-throughput inter-controller network, and explored its performance benefits. Cho et al. [12] explored some of the questions which are also addressed in this paper. The authors made a case for Intelligent SSD by discussing the bandwidth trends and quantifying the potential benefits of processing-in-storage across a range of applications.

2. Processing-in-Storage with ReCAM

The approach of this work is to design a device for storing big datasets and processing them efficiently. The key properties of this design are scalability and massively parallel processing, possible due to the non-von Neumann architecture. Parallelism is achieved by in-situ processing of the data, in contrast with NDP approaches. In this work, Resistive CAM (ReCAM), a non-volatile and scalable storage device with resistive bitcells and a novel Processing-in-Storage (PRinS) paradigm is presented. The concept is demonstrated in Fig. 1b.

2.1. ReCAM Crossbar Array

While ReRAM may employ one transistor and one memristor (1T1R) cells, ReCAM uses 2T2R cells, following [4]. Fig. 2 shows the resistive CAM crossbar. A bitcell, shown in Fig. 2a, consists of two transistors and two resistive elements (2T2R). The KEY register contains a data word to be written or compared against. The MASK register defines the active columns for write and read operations, enabling bit selectivity. The TAG register (Fig. 2b) marks the rows that are matched by the compare operation and may be affected by a parallel write. The TAG register enables chaining multiple ReCAM ICs. In a conventional CAM, compare operation is typically followed by a read of the matched data word. When in-storage processing involves arithmetic operations, a compare is usually followed by a parallel write into the unmasked bits of all tagged rows, and additional capabilities, such as read and reduction operations, are included [47].

Table 1. Operations included in ReCAM

Integer Instruction (32bit)	Cycles
B ← A+B	256
C ← A+B	512
Shift down by one row	96
Row-wise Max (A,B)	64
Max Scalar (A)	64

Any computational expression can be efficiently implemented in ReCAM storage using line-by-line execution of the truth table of the expression [7]. Arithmetic operations are typically performed bit-serially. Table 1 lists several operations supported by ReCAM and the number of cycles required by each operation. Shifting down a consecutive block of rows by one row position requires three cycles per bit. First, compare-to-'1' copies the source bit-column of all rows into the TAG. Second, shift moves the TAG vector down by setting the shift-select line (Fig. 2b). Third, write-'1' copies the shifted TAG to the same bit-column. Shifting 32-bit numbers thus requires 96 cycles. Addition (in-place or not) is performed in a bit-serial manner using a truth table approach [7] (32 bits times 8 truth-table rows times 2 for compare and write amount to 512 cycles). Row-wise maximum compares in parallel two 32-bit numbers in each row. Max Scalar tags all rows that contain the maximal value in the selected element. Additional operations, such as parallel and reduction arithmetic, may be required for other algorithms.

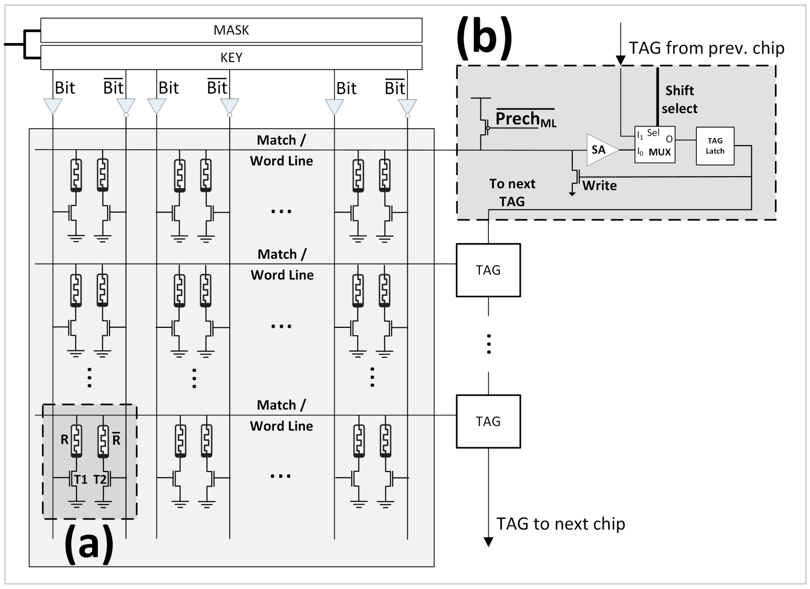


Figure 2. Single ReCAM crossbar integrated circuit. (a) 2T2R ReCAM bitcell. (b) TAG logic

2.2. System Architecture

Conceptually, the ReCAM comprises hundreds of millions of rows, each serving as a computational unit. Due to power die restrictions, the entire array may be divided into multiple smaller ICs, as in Fig. 3a. A row is fully contained within an IC. All ICs are daisy-chained for Shift and Max Scalar operations. The ReCAM storage system uses a microcontroller (Fig. 3b) similar to [23]. It issues instructions, sets the KEY and MASK registers, handles control sequences and executes read requests. In addition, the microcontroller may also perform some baseline processing, such as normalization of the reduction tree results. ReCAM-based storage is scalable due to its inherent parallelism. It allows for scalability by adding more ICs and increasing storage capacity at no performance cost since compute capability is linearly scalable in the number of ICs. Therefore, processing in-storage of large data sets does not require ReCAM for external communication, in contrast to datacenter-scale storage.

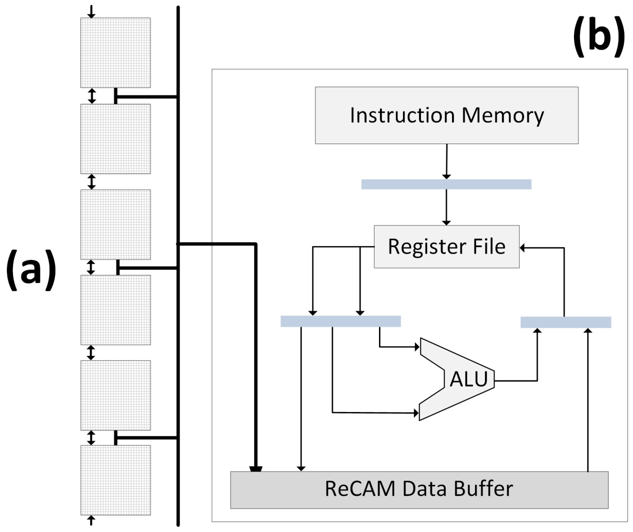


Figure 3. Complete ReCAM-based Storage system, composed of (a) separate multiple daisy-chained ICs and (b) Microcontroller. Connected to the multiple ICs with a reduction tree network

3. PRinS Application: Smith-Waterman DNA Sequence Alignment

Searching for similarities in pairs of protein and DNA sequences (also called Pairwise Alignment) has become a routine procedure in Molecular Biology and it is a crucial operation in many bioinformatic tools. The Smith-Waterman algorithm (S-W) [42] provides an optimal solution for the pairwise sequence alignment problem, but requires a number of operations proportional to the product of the two sequences. S-W identifies the optimal alignment of two sequences by computing a two-dimensional scoring matrix. Matchings base-pairs score positively, while mismatching results in a negative score. The optimal alignment score of two sequences is the highest score in the matrix. The S-W has two steps: scoring (to find the maximal alignment score) and trace-back to construct the alignment. The first step is the most computationally demanding and is the focus of this work.

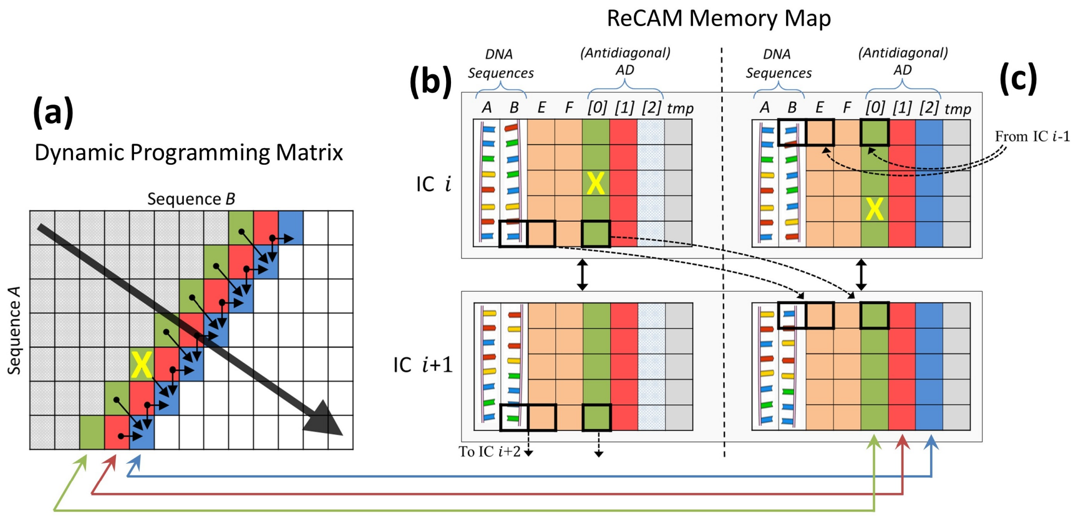


Figure 4. Mapping the dynamic programming matrix on ReCAM. (a) A snapshot of the dynamic programming matrix, shows the direction of progress for the parallel algorithm. (b) and (c) show an example of organization of data in the ReCAM crossbar array at the beginning (b) and end (c) of an iteration. AD[2] contents in (b) are being replaced with the new result (c). Bottom rows in a crossbar IC are daisy-chained to the next IC in a shift instruction. The cell marked with X contains the global maxmium score

Fig. 4a shows a snapshot of the scoring matrix during the algorithm execution. In a parallel implementation, the matrix is filled along the main diagonal, and the entire anti-diagonal scores are calculated in parallel. Fig. 4b shows the ReCAM memory map of two consecutive ICs at the beginning of an iteration. A and B contain the sequences, where each base-pair takes 2-bit and resides in a separate row. E and F are partial score results of the affine gap model [22]. AD[0], AD[1] and AD[2] contain scoring matrix anti-diagonals. Scores are represented by 32-bit integers. Shift operations in the PRinS implementation move data between rows inside an IC and between daisy-chained ICs. Fig. 4c shows the ReCAM memory map at the end of an iteration and the mapping between ReCAM and the scoring matrix. A complete description of the S-W PRinS implementation appears in [28].

3.1. Simulation and Comparison to State-of-the-art

A cycle-accurate simulator of the ReCAM storage was constructed. Assumed operational frequency is 1GHz. An in-house power simulator was used to evaluate the power consumption of ReCAM. The latency and energy figures used by both the timing and power simulations are obtained using SPICE simulation and are detailed in [47].

Table 2. Simulated ReCAM parameters

ReCAM Parameter	Value
Active storage size	8GB
Frequency	1Ghz
Power per integrated circuit	200W
Number of integrated circuit	32

We simulate the ReCAM with the cycle-accurate simulator. Assumed ReCAM parameters are listed in Table 2. The CUPS metric (Cell Updates per Second) is used to measure S-W performance. Results are compared to other works in Table 3. The in-storage implementation is compared to other implementations in different platforms: a 384-GPU cluster [37], the 128-FPGA RIVYERA platform [45] and a four Xeon Phi implementation [33]. On ReCAM with a total of 8GB in 32 separate ICs, each 256MB and 8M rows, 53 TCUPS are demonstrated, computing a total of 57×10^{12} scores. 4.7× faster than the best implementation. The table also shows computed GCUPS/Watt ratios; ReCAM is close to twice better than the FPGA solution and 80× better than the GPU system.

Table 3. Summary of state-of-the-art performance for S-W scoring step in previous works and in ReCAM

Accelerator	Xeon Phi	FPGA	GPU	ReCAM
Performance (TCUPS)	0.23	6.0	11.1	53
Number of ICs	4	128	384	32
Power (kWatt)	0.8	1.3	100	6.6
GCUPS/Watt	0.3	4.7	0.1	8.0
Reference	[33]	[45]	[37]	

4. PRinS Application: In-Storage Data Deduplication

Deduplication is a data compression technique for eliminating redundant copies of repeated data, designed to improve storage utilization. Files are split into multiple data blocks. Only unique blocks are meant to be stored. With every new write, a data block is compared against all blocks in the storage. If a match occurs, a pointer to the previously stored block is saved in lieu of the data block. Given that the same data block may occur multiple times (match frequency is also dependent on the block size), storage efficiency can be greatly improved [50].

Deduplication operates on the physical layer of the storage, managing a set of data structures to expose a consecutive logical layer of storage. Each data block has two addresses, physical (PA) and logical (LBA). Only the LBAs are exposed to the outside world, while physical addresses are used internally by the deduplication mechanism.

4.1. Related Work: Conventional Deduplication

In a typical inline storage deduplication system (comprising disk / SSD storage, CPU and DRAM for holding indices and tables), the basic deduplication data unit is termed a chunk. Upon writing a new data chunk to storage, comparing the chunk contents (typically 4-8 KByte) to the entire storage is infeasible. Instead, a much shorter representation, called a fingerprint or hash (e.g., 20-byte SHA-1 hash) is calculated for each chunk, and the fingerprint is looked up in a chunk index. If no entry is found, the chunk is stored, and a new entry is added to the chunk index. In addition to the fingerprint, the index entry also holds at least the chunk's PA and the number of references to it (Fig. 5). If the fingerprint of the new chunk is found, its number of references is incremented. An additional address translation table holds both the LBA and the PA of each chunk.

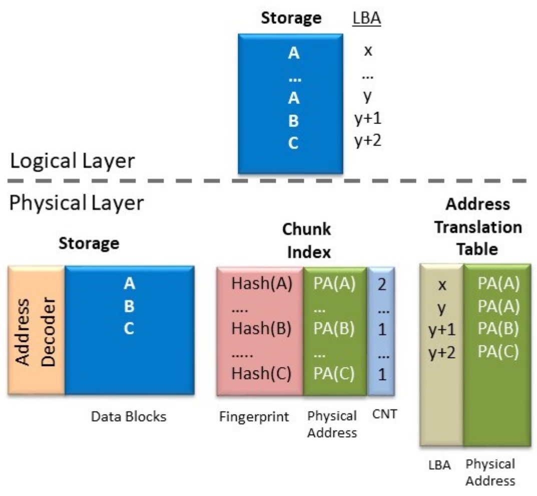


Figure 5. Conventional deduplication scheme after writing the following sequence of (data block, LBA): (A, x), (A, y), (B, y+1), (C, y+2). The storage, chunk index and address translation table reside in the physical layer

Conventional implementations of deduplication require a dedicated computer within the storage appliance. For example, a disk-based deduplication system [50] with usable capacity of 6TB employs 15 SATA drives (connected in RAID6), 500GB each, and two dual-core CPUs with 8GB of DRAM. It reaches 90% CPU utilization at peak I/O performance. All chunk metadata is stored on disk, while the DRAM serves as a cache for chunk metadata, to reduce non-I/O storage access. An expansion of that system [15] includes a flash-based SSD serving as fast storage for the entire chunk metadata. The configuration is similar to [50], although smaller, with a RAID4 storage comprising five hard drives, 500GB each, a dual-core CPU and 4GB of DRAM. As in the previous work, DRAM serves as a small cache for chunk metadata. Xtremio's X-brick [46] is an example of an all-flash high-end large-scale contemporary storage appliance. Each of its units contains either 13 or 25 SSDs with an effective capacity of 3.2 or 7.2 TB, respectively. The appliance supports up to 8 units and uses a quad-core processor with 256GB of DRAM.

At the other end of the spectrum, [9] shows an example of an in-SSD deduplication with the purpose of enhancing the device endurance. The authors suggest using the device controller and memory buffer to calculate the chunk fingerprint. Deduplication is implemented with an additional indirection in the flash translation layer and uses the buffer as a small cache (similar to the DRAM in [15] and [50]).

4.2. In-Storage ReCAM-Based Deduplication

The proposed ReCAM based inline deduplication requires neither external CPU, nor DRAM. The deduplication is accomplished entirely within the ReCAM, using its in-storage processing capabilities. ReCAM based deduplication is illustrated in Fig. 6. Each data block in ReCAM storage is divided into $S = (block\_size)/(ReCAM\_width)$ row-segments of $ReCAM\_width$ size. For example, for 256-bit wide ReCAM and 4KB blocks, the number of segments is $S = 128$. Data blocks are stored in ReCAM in segment by segment fashion, in S consecutive ReCAM rows. The first segment of each data block is marked by "1" in the block_start bit column. The values of *block_start* in all other ReCAM rows of the data block are zero.

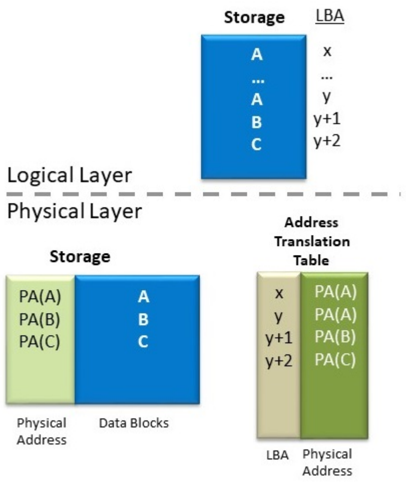


Figure 6. ReCAM based deduplication scheme, following the same sequence of writes as Fig. 5

Otherwise, the new block is unique. In that case it is written into the ReCAM along with its (arbitrarily assigned, unique) PA. As described above, the block is written segment by segment into S consecutive rows, and the first segment is marked "1" in the start_block bit column.

4.2.1. ReCAM-Based Deduplication Algorithm

Fig. 7 shows the pseudo code for the ReCAM implementation of the three main deduplication functions: write, read and delete.

During write, a new data block is compared (in parallel) against all data blocks stored in the ReCAM. This is achieved by a sequence of one single compare followed by $S - 1$ continuous compare operations. During the single compare, the *start_block* bit column is masked-on, to enable comparison of only the first segment of each data block in the storage. During the following $S - 1$ continuous compare, the result (TAG) of every consecutive compare is ANDed with the result of the previous compare. Thus, in each compare, only the rows matched in the previous compare are active, and the number of active rows drops progressively, significantly reducing the compare energy. In both cases (unique and duplicate), the LBA of the data block is placed together with its PA in an associative address translation table, which can be stored in a separate module of the

ReCAM storage. The translation table mapping can be optimized to eliminate storing multiple copies of the same PA (of duplicated blocks). Overall, write takes $O(S)$ cycles.

Read is done in two steps. First, the LBA of the data block is searched in the associative address translation table. The corresponding PA is retrieved from the table. Second, the PA is searched in the ReCAM storage (by compare), followed by read of the data block from the matched ReCAM rows. It is accomplished by a series of S read operations, starting with the row marked by "1" in the *start_block* bit column. Overall, read operation takes $O(S)$ cycles.

Deletion of a data block is performed in three steps. In the first step, the LBA is searched in the address translation table; its PA is retrieved (to be used at the second step), and the entry at the address translation table is deleted. In the second step, the PA (retrieved at the first step) is searched again in the address translation table; if matched, it means that the deleted block has no duplicates. In this case, it is deleted from the ReCAM storage, in $O(S)$ cycles.

A complete description of the in-storage deduplication appears in [29].

4.3. In-Storage Deduplication Evaluations

We simulate the ReCAM based deduplication using the cycle-accurate CAM simulator introduced in [47], employing ReCAM performance and power figures obtained by SPICE simulations. During ReCAM execution we record and count all operations (compare, write and delete). The simulated ReCAM size is 256GB, running at 1GHz. External data throughput is assumed non-limiting (contemporary interconnect such as multi-lane PCIe is capable of supporting in excess of 2.2M IOPS).

We compare our ReCAM deduplication implementation with opendedup [41], which supports inline deduplication and runs on top of the local filesystem. It allows for either variable or fixed-size blocks and does not limit the amount of stored data. In our analysis, block sizes of 1KB, 2KB, 4KB and 8KB were used. In addition, we used opendedup on a server with four octa-core Intel Xeon E5-4650 CPUs with 64GB of RAM and 800GB Intel SSD DC P3700 drive.

To evaluate the performance and energy consumption of opendedup,the file system benchmark IOzone was used [36]. IOzone allows writing data chunks with fixed number of duplicate parts, to control the degree of deduplication. All runs include writing of 50GB of data, with varying percentage of duplicate blocks. Each test was repeated with inline deduplication on and off, to isolate the CPU and DRAM energy consumptions during deduplication. Intel performance counter monitor [13] was used for measurements.

As demonstrated by [50], real-world workloads have high variability in the percentage of duplicate data. Our goal is to exhaustively examine ReCAM performance and energy consumption. Therefore we use a suite of artificial workloads with a varying degree of duplication ratio. It allows us to control both the workload and the mainline system parameters. Both opendedup and ReCAM deduplicate all duplicate blocks.

The simulated write throughput as a function of percentage of deduplicated blocks is presented in Fig. 8. The measured throughput of opendedup is also presented in Fig. 8 for comparison. The ReCAM throughput increases with the percentage of duplicate blocks, as the number of writes drops. For 8KB data blocks, ReCAM storage reaches 2.2M IOPS for 30% duplicate blocks. For comparison, high-end all-flash X-brick storage appliance reaches 150K IOPS in 30% write, 70% read operation [46], similar to the simulated performance of opendedup.

The simulated energy consumption of ReCAM-based deduplication as a function of percentage of deduplicated blocks is presented in Fig. 9. The measured energy consumption of opendedup

```
1: function DEDUP_WRITE(logical_address, data_block)
2:     Compare (data_block)                ▷ in parallel for all data block stored in ReCAM
3:     if NNZ_TAG then                                              ▷ no match for data block
4:         Compare(empty_bit)
5:         First_tag()
6:         Write (address, data, empty_bit)                              ▷ empty_bit ← 0
7:     else                                                      ▷ data_block is duplicated
8:         Read(logical_address)  ▷ reads the address from matching data block in ReCAM
9:         Save (new block address)      ▷ Save pointer to an existing block in an associative
   address conversion table. Could be stored in a Se
10:    end if
11: end function

1: function DEDUP_READ(logical_address)
2:     Compare (logical_address)                   ▷ find address in address conversion table
3:     if NNZ_TAG == 0 then ▷ if found, block is deduplicated. Need to fetch its physical
   address.
4:         Read (physical_address)    ▷ read physical_address field from address conversion
   table
5:         Compare (pysical_address)                         ▷ find physical_address in ReCAM
6:     else                                  ▷ block is unique, address is its physical address
7:         Compare (logical_address)
8:     end if
9:     Read (data_block)                                                  ▷ read a data_block
10: end function

1: function DEDUP_DELETE(logical_address)
2:     Compare (logical_address)                   ▷ find address in address conversion table
3:     if NNZ_TAG == 0 then                                              ▷ match for address.
4:         Remove (logical_address)        ▷ if deduplicated, remove pointer from associative
   address conversion table
5:     else                                                           ▷ data block is unique
6:         Delete (logical_address)                  ▷ delete data block from ReCAM storage
7:     end if
8: end function
```

Figure 7. Associative Write, Read and Delete in ReCAM-based data deduplication

(including the SSD energy consumption) is also presented in Fig. 9 for comparison. The energy consumption of ReCAM-based deduplication is in the same range (slightly higher for smaller blocks, lower for larger blocks).

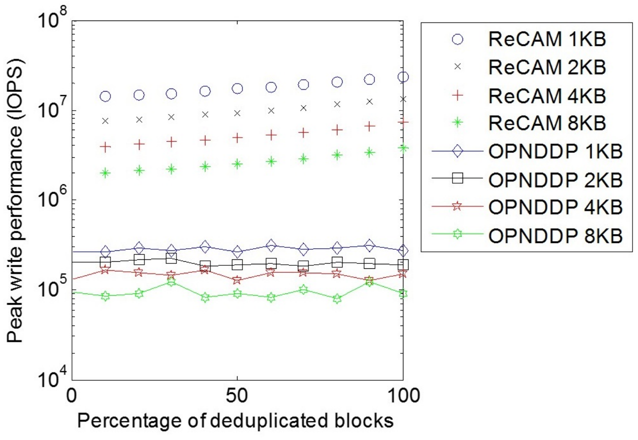


Figure 8. Write performance for different block sizes vs. percentage of deduplicated blocks, for data blocks of 1KB, 2KB, 4KB and 8KB (OPNDDP = Opendedup)

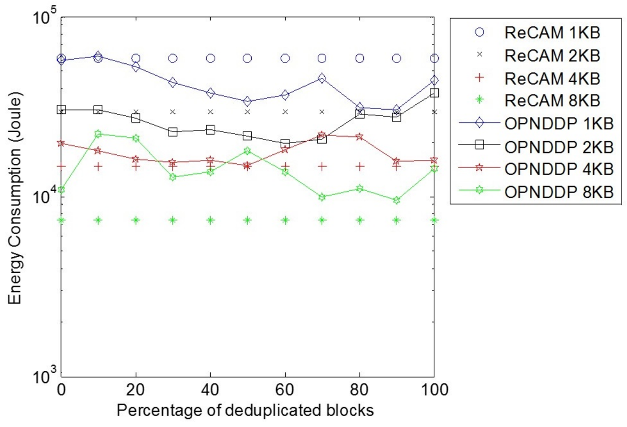


Figure 9. Deduplication energy for different block sizes vs. percentage of deduplicated blocks, for data blocks of 1KB, 2KB, 4KB and 8KB while writing 50GByte of data

Conclusions

Processing-in-memory does not address the bandwidth bottleneck problem when solving big data workloads. We propose a novel in-data processing-in-storage architecture based on Resistive Content Addressable Memory (ReCAM). It enables mass storage with in-data associative processing capabilities. ReCAM storage contains billions of data rows, each row serving as an associative processing unit. ReCAM requires no in-storage processing cores external to the storage arrays. There is no data transfer outside the storage arrays. Therefore, the internal bandwidth of the resistive memory based storage can be utilized to its fullest extent, considerably improving computation throughput of processing-in-storage system.

The ReCAM architecture, capable of general purpose associative processing, has been applied to challenging big data problems, such as the Smith-Waterman bioinformatics algorithm and inline data deduplication. The paper also compares ReCAM to other implementations and shows a significant improvement in performance and energy efficiency.

References

1. Ahn, J., Hong, S., Yoo, S., Mutlu, O., Choi, K.: A scalable processing-in-memory accelerator for parallel graph processing. In: 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). pp. 105–117 (June 2015), DOI: 10.1145/2749469.2750386

2. Ahn, J., Yoo, S., Mutlu, O., Choi, K.: Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In: 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). pp. 336–348 (June 2015), DOI: 10.1145/2749469.2750385

3. Akin, B., Franchetti, F., Hoe, J.C.: Hamlet architecture for parallel data reorganization in memory. IEEE Micro 36(1), 14–23 (Jan 2016), DOI: 10.1109/MM.2015.129

4. Akinaga, H., Shima, H.: Resistive random access memory (reram) based on metal oxides. Proceedings of the IEEE 98(12), 2237–2251 (Dec 2010), DOI: 10.1109/JPROC.2010.2070830

5. Azarkhish, E., Pfister, C., Rossi, D., Loi, I., Benini, L.: Logic-base interconnect design for near memory computing in the smart memory cube. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25(1), 210–223 (2017)

6. Bae, D.H., Kim, J.H., Kim, S.W., Oh, H., Park, C.: Intelligent ssd: A turbo for big data mining. In: Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management. pp. 1573–1576. CIKM '13, ACM, New York, NY, USA (2013), DOI: 10.1145/2505515.2507847

7. Balasubramonian, R., Chang, J., Manning, T., Moreno, J.H., Murphy, R., Nair, R., Swanson, S.: Near-data processing: Insights from a micro-46 workshop. IEEE Micro 34(4), 36–42 (2014)

8. Boboila, S., Kim, Y., Vazhkudai, S.S., Desnoyers, P., Shipman, G.M.: Active flash: Out-of-core data analytics on flash storage. In: 012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–12 (April 2012), DOI: 10.1109/MSST.2012.6232366

9. Chen, F., Luo, T., Zhang, X.: Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In: Proceedings of the 9th USENIX Conference on File and Stroage Technologies. pp. 6–6. FAST'11, USENIX Association, Berkeley, CA, USA (2011), `http://dl.acm.org/citation.cfm?id=1960475.1960481`

10. Chi, P., Li, S., Xu, C., Zhang, T., Zhao, J., Liu, Y., Wang, Y., Xie, Y.: Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). pp. 27–39 (June 2016), DOI: 10.1109/ISCA.2016.13

11. Cho, B.Y., Jeong, W.S., Oh, D., Ro, W.W.: Xsd: Accelerating mapreduce by harnessing the gpu inside an ssd. In: Proceedings of the 1st Workshop on Near-Data Processing (2013)

12. Cho, S., Park, C., Oh, H., Kim, S., Yi, Y., Ganger, G.R.: Active disk meets flash: A case for intelligent ssds. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. pp. 91–102. ICS '13, ACM, New York, NY, USA (2013), DOI: 10.1145/2464996.2465003

13. Corporation, I.: Intel performance counter moniter. www.intel.com/software/pcm (2017), accessed: 2017-07-15

14. De, A., Gokhale, M., Gupta, R., Swanson, S.: Minerva: Accelerating data analysis in next-generation ssds. In: 21st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2013, Seattle, WA, USA, April 28-30, 2013. pp. 9–16. IEEE Computer Society (2013), http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6545868

15. Debnath, B., Sengupta, S., Li, J.: Chunkstash: Speeding up inline storage deduplication using flash memory. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference. pp. 16–16. USENIXATC'10, USENIX Association, Berkeley, CA, USA (2010), http://dl.acm.org/citation.cfm?id=1855840.1855856

16. Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: Proceedings of the 38th Annual International Symposium on Computer Architecture. pp. 365–376. ISCA '11, ACM, New York, NY, USA (2011), DOI: 10.1145/2000064.2000108

17. Farmahini-Farahani, A., Ahn, J.H., Morrow, K., Kim, N.S.: Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). pp. 283–295 (Feb 2015), DOI: 10.1109/HPCA.2015.7056040

18. Foster, C.C.: Content Addressable Parallel Processors. John Wiley & Sons, Inc., New York, NY, USA (1976)

19. Gao, M., Kozyrakis, C.: Hrl: Efficient and flexible reconfigurable logic for near-data processing. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 126–137 (March 2016), DOI: 10.1109/HPCA.2016.7446059

20. Gao, M., Ayers, G., Kozyrakis, C.: Practical near-data processing for in-memory analytics frameworks. In: Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT). pp. 113–124. PACT '15, IEEE Computer Society, Washington, DC, USA (2015), DOI: 10.1109/PACT.2015.22

21. Gokhale, M., Holmes, B., Iobst, K.: Processing in memory: the terasys massively parallel pim array. Computer 28(4), 23–31 (Apr 1995), DOI: 10.1109/2.375174

22. Gotoh, O.: An improved algorithm for matching biological sequences. Journal of Molecular Biology 162(3), 705 – 708 (1982)

23. Guo, Q., Guo, X., Patel, R., Ipek, E., Friedman, E.G.: Ac-dimm: Associative computing with stt-mram. SIGARCH Comput. Archit. News 41(3), 189–200 (Jun 2013), DOI: 10.1145/2508148.2485939

24. Hall, M., Kogge, P., Koller, J., Diniz, P., Chame, J., Draper, J., LaCoss, J., Granacki, J., Brockman, J., Srivastava, A., Athas, W., Freeh, V., Shin, J., Park, J.: Mapping irregular applications to diva, a pim-based data-intensive architecture. In: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing. SC '99, ACM, New York, NY, USA (1999), DOI: 10.1145/331532.331589

25. Jo, Y.Y., Cho, S., Kim, S.W., Oh, H.: Collaborative processing of data-intensive algorithms with cpu, intelligent ssd, and gpu. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. pp. 1865–1870. SAC '16, ACM, New York, NY, USA (2016), DOI: 10.1145/2851613.2851741

26. Jun, S.W., Liu, M., Lee, S., Hicks, J., Ankcorn, J., King, M., Xu, S., Arvind: Bluedbm: An appliance for big data analytics. In: Proceedings of the 42Nd Annual International Symposium on Computer Architecture. pp. 1–13. ISCA '15, ACM, New York, NY, USA (2015), DOI: 10.1145/2749469.2750412

27. Kang, Y., s. Kee, Y., Miller, E.L., Park, C.: Enabling cost-effective data processing with smart ssd. In: 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–12 (May 2013), DOI: 10.1109/MSST.2013.6558444

28. Kaplan, R., Yavits, L., Ginosar, R., Weiser, U.: A resistive cam processing-in-storage architecture for dna sequence alignment. IEEE Micro 37(4), 20–28 (2017), DOI: 10.1109/MM.2017.3211121

29. Kaplan, R., Yavits, L., Morad, A., Ginosar, R.: Deduplication in resistive content addressable memory based solid state drive. In: 2016 26th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS). pp. 100–106 (Sept 2016), DOI: 10.1109/PATMOS.2016.7833432

30. Kogge, P.M., b. Brockman, J., Freeh, V.W.: Pim architectures to support petaflops level computation in the htmt machine. In: Innovative Architecture for Future Generation High-Performance Processors and Systems (Cat. No.PR00650). pp. 35–44 (Dec 1999), DOI: 10.1109/IWIA.1999.898841

31. Lipovski, G.J., Yu, C.: The dynamic associative access memory chip and its application to simd processing and full-text database retrieval. In: Records of the 1999 IEEE International Workshop on Memory Technology, Design and Testing. pp. 24–31 (1999), DOI: 10.1109/MTDT.1999.782680

32. y. Liu, T., Yan, T.H., Scheuerlein, R., Chen, Y., Lee, J.K., Balakrishnan, G., Yee, G., Zhang, H., Yap, A., Ouyang, J., Sasaki, T., Al-Shamma, A., Chen, C., Gupta, M., Hilton, G., Kathuria, A., Lai, V., Matsumoto, M., Nigam, A., Pai, A., Pakhale, J., Siau, C.H., Wu, X., Yin, Y., Nagel, N., Tanaka, Y., Higashitani, M., Minvielle, T., Gorla, C., Tsukamoto, T., Yamaguchi, T., Okajima, M., Okamura, T., Takase, S., Inoue, H., Fasoli, L.: A 130.7-$hboxmm^2$ 2-layer 32-gb reram memory device in 24-nm technology. IEEE Journal of Solid-State Circuits 49(1), 140–153 (Jan 2014), DOI: 10.1109/JSSC.2013.2280296

33. Liu, Y., Schmidt, B.: Swaphi: Smith-waterman protein database search on xeon phi coprocessors. In: 2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors. pp. 184–185 (June 2014), DOI: 10.1109/ASAP.2014.6868657

34. Nair, R., Antao, S.F., Bertolli, C., Bose, P., Brunheroto, J.R., Chen, T., Cher, C.Y., Costa, C.H.A., Doi, J., Evangelinos, C., Fleischer, B.M., Fox, T.W., Gallo, D.S., Grinberg, L., Gunnels, J.A., Jacob, A.C., Jacob, P., Jacobson, H.M., Karkhanis, T., Kim, C., Moreno, J.H., O'Brien, J.K., Ohmacht, M., Park, Y., Prener, D.A., Rosenburg, B.S., Ryu, K.D., Sallenave, O., Serrano, M.J., Siegl, P.D.M., Sugavanam, K., Sura, Z.: Active memory cube: A processing-in-memory architecture for exascale systems. IBM Journal of Research and Development 59(2/3), 17:1–17:14 (March 2015), DOI: 10.1147/JRD.2015.2409732

35. Nitin, Thottethodi, M., Vijaykumar, T., et al.: Rowcore: A processing-near-memory architecture for big data machine learning. Purdue ECE Technical Report 473 (2016)

36. Norcott, W.D., Capps, D.: Iozone filesystem benchmark. `http://www.iozone.org/` (2003), accessed: 2017-07-15

37. d. O. Sandes, E.F., Miranda, G., Martorell, X., Ayguade, E., Teodoro, G., Melo, A.C.M.: Cudalign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in gpu clusters. IEEE Transactions on Parallel and Distributed Systems 27(10), 2838–2850 (Oct 2016), DOI: 10.1109/TPDS.2016.2515597

38. Paul, S., Bhunia, S.: A scalable memory-based reconfigurable computing framework for nanoscale crossbar. IEEE Transactions on Nanotechnology 11(3), 451–462 (May 2012), DOI:10.1109/TNANO.2010.2041556

39. Potter, J.L., Meilander, W.C.: Array processor supercomputers. Proceedings of the IEEE 77(12), 1896–1914 (Dec 1989), DOI: 10.1109/5.48831

40. Shafiee, A., Nag, A., Muralimanohar, N., Balasubramonian, R., Strachan, J.P., Hu, M., Williams, R.S., Srikumar, V.: Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). pp. 14–26 (June 2016), DOI: 10.1109/ISCA.2016.12

41. Silverberg, S.: Opendedup sdfs. `http://opendedup.org/odd/` (2010), accessed: 2017-07-15

42. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of molecular biology 147(1), 195–197 (1981), DOI: 10.1016/0022-2836(81)90087-5

43. Suh, J., Li, C., Crago, S.P., Parker, R.: A pim-based multiprocessor system. In: Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001. pp. 6 pp.– (Apr 2001), DOI: 10.1109/IPDPS.2001.924932

44. Sura, Z., Jacob, A., Chen, T., Rosenburg, B., Sallenave, O., Bertolli, C., Antao, S., Brunheroto, J., Park, Y., O'Brien, K., Nair, R.: Data access optimization in a processing-in-memory system. In: Proceedings of the 12th ACM International Conference on Computing Frontiers. pp. 6:1–6:8. CF '15, ACM, New York, NY, USA (2015), DOI: 10.1145/2742854.2742863

45. Wienbrandt, L.: The FPGA-Based High-Performance Computer RIVYERA for Applications in Bioinformatics, pp. 383–392. Springer International Publishing, Cham (2014), DOI: 10.1007/978-3-319-08019-2_40

46. XtremIO, E.: X-Brick tech spec. https://www.emc.com/collateral/data-sheet/h12451-xtremio-4-system-specifications-ss.pdf (2015), accessed: 2017-07-06

47. Yavits, L., Kvatinsky, S., Morad, A., Ginosar, R.: Resistive associative processor. IEEE Computer Architecture Letters 14(2), 148–151 (July 2015), DOI: 10.1109/LCA.2014.2374597

48. Yavits, L., Morad, A., Ginosar, R.: Computer architecture with associative processor replacing last-level cache and simd accelerator. IEEE Transactions on Computers 64(2), 368–381 (Feb 2015), DOI: 10.1109/TC.2013.220

49. Zhang, D., Jayasena, N., Lyashevsky, A., Greathouse, J.L., Xu, L., Ignatowski, M.: Top-pim: Throughput-oriented programmable processing in memory. In: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing. pp. 85–98. HPDC '14, ACM, New York, NY, USA (2014), DOI: 10.1145/2600212.2600213

50. Zhu, B., Li, K., Patterson, H.: Avoiding the disk bottleneck in the data domain deduplication file system. In: Proceedings of the 6th USENIX Conference on File and Storage Technologies. pp. 18:1–18:14. FAST'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1364813.1364831