SANDIA REPORT

SAND2015-10345
Unlimited Release
Printed November 2015

An Asynchronous Many-Task Implementation of In-Situ Statistical Analysis using Legion

Philippe Pébay, Janine Bennett

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2015-10345
Unlimited Release
Printed November 2015

An Asynchronous Many-Task Implementation of In-Situ Statistical Analysis using Legion

Philippe Pébay, Janine Bennett
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551, U.S.A.
pppebay,jcbenne@sandia.gov

Abstract

In this report, we propose a framework for the design and implementation of in-situ analyses using an asynchronous many-task (AMT) model, using the Legion programming model together with the MiniAero mini-application as a surrogate for full-scale parallel scientific computing applications.

The bulk of this work consists of converting the Learn/Derive/Assess model which we had initially developed for parallel statistical analysis using MPI [PTBM11], from a SPMD to an AMT model. In this goal, we propose an original use of the concept of Legion logical regions as a replacement for the parallel communication schemes used for the only operation of the statistics engines that require explicit communication.

We then evaluate this proposed scheme in a shared memory environment, using the Legion port of MiniAero as a proxy for a full-scale scientific application, as a means to provide input data sets of variable size for the in-situ statistical analyses in an AMT context. We demonstrate in particular that the approach has merit, and warrants further investigation, in collaboration with ongoing efforts to improve the overall parallel performance of the Legion system.

Acknowledgments

The authors would like to thank Michael BAUER, Wonchan LEE, Elliott SLAUGHTER, and Sean TREICHLER at Stanford University for their help getting started with Legion, as well as David HOLLMAN and Hemanth KOLLA at Sandia National Laboratories who provided helpful guidance with MiniAero.

Contents

1 Introduction 7
2 Background 8
2.1 Terminology 8
2.2 Legion 9
2.3 MiniAero 11
2.4 Titan Statistics 12
3 Parallel Statistics in Legion 15
3.1 Approach 15
3.2 Implementation Details 17
3.3 In-Situ Setting with MiniAero 19
4 Results 24
4.1 Algorithm Correctness 24
4.2 Parallel Scalability 26
5 Conclusion 29
References 31

This page intentionally left blank

1 Introduction

Over the last two decades the Department of Energy has established the essential role of high-performance computing in advancing a wide range of scientific disciplines. Driven by constrained power budgets, the future success of this vital component of scientific discovery faces a critical challenge from rapidly changing processor and system architecture designs. In particular, the placement and movement of data is becoming the key-limiting factor on both performance and energy efficiency. Furthermore, the increased quantities of data the systems are capable of generating, in conjunction with the insufficient rate of improvements in the supporting I/O infrastructure, is forcing applications away from the off-line post-processing of data towards techniques based on in situ analysis and visualization. Together, these challenges are shaping how we will both design and develop effective, performant and energy-efficient software. In particular, they highlight the need for data and data-centric operations to be fundamental in the reasoning about, and optimization of, scientific work flows on extreme-scale architectures.

We therefore aim to design, develop, and evaluate a unified data-driven approach for programming applications and in situ analysis and visualization. Specifically to understand the interplay between data-centric programming model requirements at extreme-scale and the overall impact of those requirements on the design, capabilities, flexibility, and implementation details for both applications and the supporting in situ infrastructure. This work leverages the Legion programming model and run-time system that has been developed as part of the ASCR-funded ExaCT Co-Design Center.

Despite its importance, this area is largely unexplored, and the current work is therefore focused on studying the impact of:

1. supporting effective in situ data management analysis and visualization,
2. providing a foundation for building efficient and effective work flow management,
3. enabling an interactive in situ user environment on the underlying run-time software design, and
4. understanding the impact on existing applications, infrastructure and tools.

With this overarching goal in mind, the present report describes our first effort in this direction, with the design and development of statistical data analysis data algorithms in Legion using a Miniaero as a proxy application for large-scale simulation programs.

2 Background

2.1 Terminology

We begin by defining some of the terminology that will be used throughout this report. For further detail, please refer to [BC+15].

We first recall that *data parallelism* involves carrying out a single task and/or instruction on different segments of data across many computational units. Different types of data parallelism include *single-instruction, multiple-data* (SIMD), a type of instruction level parallelism where an individual instruction is synchronously executed on different segments of data and is best illustrated by vector processing on a central processing unit (CPU) or many integrated cores (MIC) architecture, and *single-program multiple-data* (SPMD), where the same tasks are carried out by multiple processing units but operate on different sets of input data (e.g., multithreading on a single compute node, and distributed computing using Message Passing Interface (MPI) communication).

In contrast, *task parallelism* focuses on completing multiple tasks simultaneously over different computational units. These tasks may operate on the same segment of data or many different data sets. In particular, task parallelism can occur when non-conflicting tasks operate on the same data, usually because they only require read-only access. At a process level, this is a form of *multiple-program multiple-data* (MPMD). Pipeline parallelism is achieved by breaking up a task into a sequence of individual sub-tasks, each of which represents a stage whose execution can be overlapped. Pipeline parallelism is most often associated with data movement operations, overlapping data fetches with computational work to hide latency and minimize gaps in the task pipeline, which can occur for both on-node and remote memory fetches.

A parallel *programming model* is an abstract view of a machine and set of first-class constructs for expressing algorithms. The programming model focuses on how problems are decomposed and expressed, and provides the mechanisms for an application to express *concurrency*. Programming models are often characterized according to their style, for example *imperative*, *declarative*, *procedural*, or *functional*. In this report we will focus on declarative programming, in which the programmer expresses or defines the desired result without specifying how the result is to be achieved.

A *programming language* is a syntax and a set of code constructs for implementing one or more programming models. For example, the C++ programming language supports both functional and procedural imperative programming models.

A *parallel execution model* specifies how an application creates and manages concurrency. Examples of various execution models include communicating sequential processes (CSP), strict fork-join, the actor model, and event-based models. The execution model classifications also distinguish how parallel data access hazards are managed. Without providing a full description of the different execution models, we note that in imperative style the programming model and execution model are closely tied and therefore not distinguished. In contrast, a declarative programming style decouples the execution model from the programming model.

A *parallel run-time system* primarily implements portions of an execution model, managing how and where concurrency is managed and created. Run-time systems therefore control the order in which parallel work (decomposed and expressed via the programming model) is actually performed and executed. Run-time systems can range greatly in complexity, comprising all aspects of parallel execution that are not explicitly managed by the application.

An *asynchronous many-task* (AMT) model is a categorization of programming and execution models that break from the dominant CSP or SPMD models. Different asynchronous many-task run-time system (AMT RTS) implementations can share a common AMT model. An asynchronous many-task (AMT) programming model decomposes applications into small, transferable units of work (many tasks) with associated inputs (dependencies or data blocks) rather than simply decomposing at the process level (MPI ranks). An AMT execution model can be viewed as the coarse-grained, distributed memory analog of instruction-level parallelism, extending the concepts of data pre-fetching, out-of-order task execution based on dependency analysis, and even branch prediction (speculative execution). Rather than executing in a well-defined order, tasks execute when inputs become available. An AMT model aims to leverage all available task and pipeline parallelism, rather than only relying on basic data parallelism for concurrency. The term "asynchronous" conveys both ideas that processes (threads) can diverge to different tasks, rather than executing in the same order, and that concurrency is maximized (minimum synchronization) by leveraging multiple forms of parallelism. The term "many-task" conveys the idea that the application is decomposed into many transferable or migratable units of work, to enable the overlap of communication and computation as well as asynchronous load balancing strategies.

2.2 Legion

Legion is a data-centric programming model for writing high-performance applications for distributed heterogeneous architectures, which grew out of earlier work on the data-centric Sequoia language, under the auspices of A. Aiken at Stanford. It is now a joint research effort also involving Los Alamos National Laboratory, NVIDIA Research, UC Santa Cruz, and the University of Utah. Its highly declarative program expression is a strong shift from the procedural style of MPI.

Legion programs comprise tasks that operate on logical regions, which simply name collections of objects. When writing a task, the programmer explicitly declares the properties of the data that will be operated on by the task. This includes the datas type, organization (e.g., array of structs, struct of arrays), privileges (e.g., read-only, read-write, write-only, reduction), partitioning, and coherence. The run-time system leverages these data properties to issue data movement operations as needed, removing this burden from the developer. Task dependencies can be inferred from the data properties allowing the run-time to determine when tasks can be executed, including reordering tasks and executing them in parallel.

Furthermore, the separation of the logical and physical representation of data enables the run-time to, e.g., create multiple copies of read-only data to maximize parallelism as appropriate. Legion aims to decouple the specification of a program from its optimization via its mapping interface, which gives developers control over the details of data placement and task execution.

While this data-driven approach is extremely powerful for extracting parallelism, the trade-off is that all inputs and outputs for a task must be declared a priori. In cases involving data-dependent or dynamically sparse execution, not enough information can always be expressed a priori for the Legion run-time to extract useful amounts of parallelism. Legion currently provides some mechanisms and has other proposed solutions for addressing these issues.

The following principles have driven the design and implementation of Legion:

User control of decomposition: While Legion, in contrast to MPI, can automatically manage task preconditions, it is similar to MPI in requiring that the user specify the data decomposition into logical regions. Similarly, the choice of how to decompose algorithms into tasks is also the responsibility of the application developer. MPI (other than MPI types) does not provide a data model, leaving data management at the application level. Legion provides a relational data model for expressing task decomposition that supports multiple views or decompositions of the same data.

Handle irregularity: Legion aims to support dynamic decision making at run-time to handle irregularities. This includes the ability to make dynamic decisions regarding how data is partitioned, where data and tasks are placed.

Hybrid programming model: Tasks in Legion are functional with controlled side effects on logical regions as described by the tasks data properties. However the tasks themselves consist of imperative code. This coarse-grained functional programming model enables the distributed scheduling algorithm of Legion to order tasks, while still supporting the imperative coding style within tasks that is familiar to most application developers.

Deferred execution: All run-time calls in Legion are deferred, which means that they can be launched asynchronously and Legion is responsible for computing the necessary dependencies, not performing operations until it is safe to do so. This is only possible because Legion understands the structure of program data for deferring data movement operations and because Legion can reason about the side-effects of a task on logical regions.

Provide mechanism but not policy: Legion is designed to give programmers control over the policy of how an application is executed, while still automating any operation which can be inferred from the given policy. For example, Legion provides control over where tasks and data are run, but the run-time automatically infers the necessary copies and data movement operations to conform to the specified privilege and coherence annotations on the logical regions arguments to each task. Default implementations exists for many tools like the Mapper, accelerating the path to a debug implementation, although a wide and performant set of default libraries is still lacking.

Decouple correctness from performance: In conjunction with the previous design principle, Legion ensures that policy decisions never impact the correctness of an application. This is useful from a performance portability perspective, as a common specification can be mapped to multiple machine types. The Legion run-time provides a default mapper, and the mapping interface is intentionally extensible to support both custom mappers and the creation of mapping tools for building custom mappers, for it will never be possible for a default mapper to perform an optimal mapping

for all applications and all machine architectures. In practice, the decoupling of performance optimization and application code has not always been limited to the mapper interface, e.g., explicit ghosting can be used to improve performance (which changes the application code).

Legion is designed for two classes of users: advanced application developers and DSL and library authors. It is therefore a natural candidate as an AMT run-time system to support the our target workload requirements.

2.3 MiniAero

MiniAero [FFLB15] is used as basis for this study; it is a compressible Navier-Stokes, three-dimensional, unstructured mesh, finite volume, explicit, computational fluid dynamics (CFD) mini-application that is representative of a part of the computational requirements for the re-entry application of particular interest to Sandia. There is a baseline implementation of MiniAero available online at [man] that is approximately 3800 lines of C++ code, using MPI+Kokkos [ETS14]. MiniAero solves the Compressible Navier-Stokes equations using Runge-Kutta fourth-order time marching and provides options for 1st or 2nd order spatial discretization of inviscid fluxes (employing Roes approximate Riemann solver). The boundary conditions include supersonic inflow, supersonic outflow, and tangent flow. The fluxes are five-component vector quantities (mass, momentum vector, and energy). For the viscous and 2nd-order terms, a matrix of spatial gradients is required consisting of the x, y, and z components of each of the mass, momentum vector, and energy quantities. Once the residual is computed, it can be summed into the original values to obtain new momentum, mass, and energy.

Most many-task run-time schedulers work with a directed acyclic graph (DAG), often referred to as a task-DAG that encodes all the precedence constraints in a program: each node in the DAG represents computational work, while precedence constraints are expressed via directed edges in the graph, indicating which tasks must complete before other tasks can begin. If there is no edge (precedence constraint) between two tasks then they can safely run in parallel. In general, fewer precedence constraints lead to a wider task-DAG which has more concurrency to exploit.

In its most basic form, MiniAero is, at its core, a very static application, with a fairly narrow task-DAG for, making it amenable to SPMD implementation. It it nonetheless an interesting use case for an AMT implementation, as it provides a realistic proxy for full-scale scientific applications. It has, in particular, been ported to the Legion RTS (cf. [BC$^+$15]), and this is the application code which we used in this for our AMT in-situ research work presented here.

The distribution of MiniAero comes with a test harness for non-regression testing. Each of the tests which belong to this framework can also be used in a stand-alone mode, and its input deck be modified so various parameters can be explored individually or in conjunction. For our in-situ analyses, we have retained a common test for the accuracy of computational fluid codes, initially studied by [Sod78] from whom it takes its name. Specifically, we used the test case called "3D_Sod", which implements a 1D shock tube problem, but set up in a 3D domain discretized into 3D rectilinear finite volume cells. The initial conditions are just a sharp jump in field variables

at the middle of the x-range of the domain, along which the field variables vary. The physical dimensions of the domain are specified in the input file `miniaero.inp` in each test case. For our analyses we retained the default values and we therefore do not discuss them here. The parameter which is the most important for us here, is the number of blocks into which the computational grid is to be decomposed for parallel execution. The implementation requires that this number of blocks be a power of 2; as a result, in order to always have perfectly load-balanced setting, one must ensure that the number of cells along the x-axis be itself a power of 2.

2.4 Titan Statistics

In [PTBM11] we have presented a comprehensive view of the scalable, parallel statistical analysis library which we designed and implemented and released as part of the open-source Visualization Tool Kit (VTK) [Kit10], itself part of the Titan Informatics Toolkit [WB09]. Without providing a comprehensive view of this library, which would be beyond the scope of this report, we hereafter justify the choice it as a first candidate for our research on AMT in-situ analysis using Legion.

The Titan parallel statistics *engines* are a set of C++ classes, developed at Sandia National Laboratories between 2008 and 2013, in particular by the authors of this report, based on SPMD data parallelism using MPI. We therefore knew this tool kit very well before starting this study; we knew in particular the limitations of the bulk-synchronicity when attempting to achieve extreme-scale scalability. There was therefore a strong motivation to research other approaches, besides SPMD, which would to allow for data analysis at scales orders of magnitude beyond what we were able to achieve with MPI (optimal scalability with up to $O(10^5)$ cores).

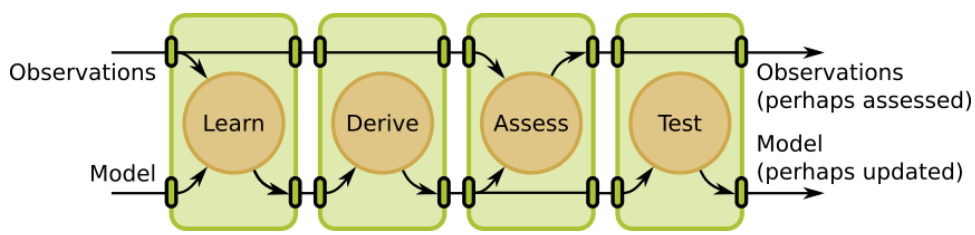

Figure 1. The 4 operations of statistical analysis and their interactions with input observations and models. When an operation is not requested, it is eliminated by connecting input to output ports.

The design decisions made during development were motivated by two primary factors:

(i) We wanted to mimic the predominant types of data analysis work flows, so that a data analyst using our framework would find it natural and intuitive to use.

(ii) The design had to be conducive to embarrassingly parallel implementations when possible.

In order to meet these two overlapping but not exactly congruent design requirements, we isolated those parts of the analysis which by construction are not embarrassingly parallel (due to the mathematics of the statistical analysis itself, not due to our design) so that parallel design trade-offs be limited to those components where embarrassingly parallel implementations are not viable. Specifically, the statistical analysis work flow is split into 4 disjoint operations:

- Learn a model from observations,
- Derive statistics from a model,
- Assess observations with a model, and
- Test a hypothesis.

These operations, when all are executed, occur in order as shown in Figure 1. However, it is also possible to execute only a subset of these, for example when it is desired that previously computed models, or models constructed with expert knowledge, be used in conjunction with existing data. Note that in earlier publications (e.g., [PTB10]) only the first 3 operations are mentioned; the Test operation, which we initially saw as a part of Derive, was separated out afterwards for reasons both theoretical (statistical hypothesis testing relies on assumptions not accepted by all statisticians) and practical (second pass through the data might be necessary, as well as calls to outside libraries). We therefore always considered the Test operation as an experimental feature, not implemented for all statistics engines, and therefore we leave it aside for the current study.

From the parallelism standpoint, this subdivision of the work flow reduces the Learn operation to a special case of the map-reduce pattern [DG04], while the remaining two are embarrassingly parallel. Specifically, the Learn operation belongs to the map-reduce pattern in that the parallel algorithm computes (maps) a set of distributed (key,value) pairs. All local values associated with the same key are then merged by the reduce function to compute the global *primary model*. In some of our statistical algorithms, namely *moment-based*, it is not necessary to communicate the keys for there is a fixed number of them, identical across all processes, and these keys may be ordered uniquely, so sending values alone is unambiguous. In addition, the number of such keys is typically very small (less than 10), which allowed us to implement the reduce function as an AllGather MPI collective, allowing each core to perform locally all subsequent operations with no further communication. In contrast, for *quanta-based* algorithms, tables with an arbitrary number of key-value pairs must be communicated and different keys may be present on each process. We therefore implemented the reduce function using a Gather-Broadcast two-step operation, involving a small number of reduction nodes on which this procedure is performed, as illustrated in Figure 2 with a univariate analysis where the Learn operation builds a global histogram, along with derived statistics such as empirical PDF and quantiles. This can potentially cause problems as the size of the network increases to peta-scale, thereby justifying the need to investigate asynchronous reduction strategies.

We conducted several systematic, multiple-parameter studies of the scalability properties of our SPMD/MPI implementation with up to 10^5 cores on a tera-scale system, studying both weak and strong scaling properties. Those tests demonstrated, in particular, that our design allowed

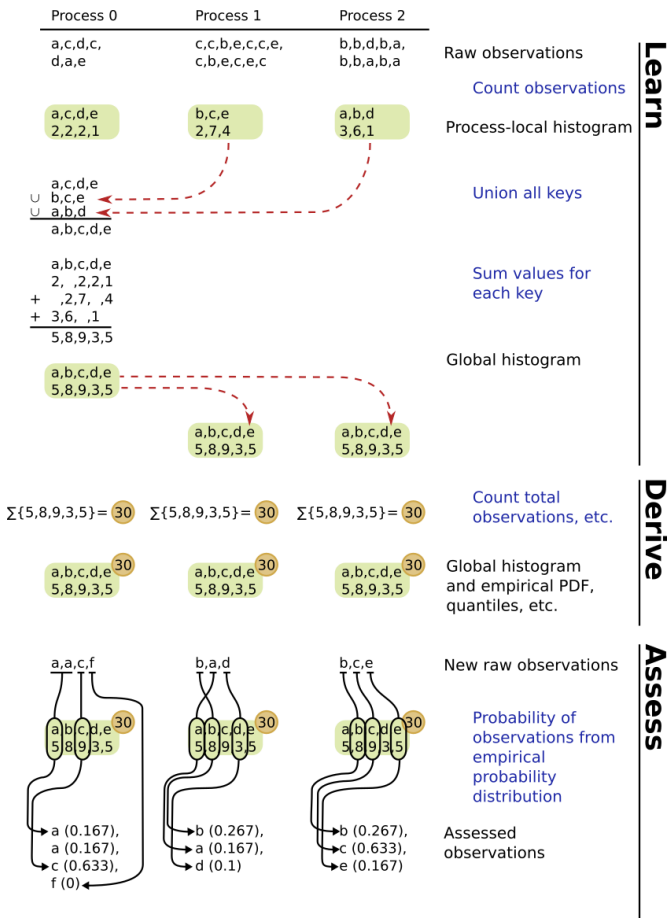

Figure 2. A simplified example illustrating the operations of the parallel order statistics; dashed red arrows indicate inter-process communication. In terms of the map-reduce pattern, keys are the raw observations (represented by letters a, b, c, d, e) and values are the number of observations.

for optimal scalability of all moment-based engines, provided the input data sets were evenly distributed across the system. Note that these studies purposefully ignored I/O and data movement issues, using instead synthetic test data sets were generated on-the-fly by each process participating in the experiment in order to isolate the analysis per se from other aspects that mostly depend on the target platform. On the other hand, we established that for quanta-based statistics, our design trade-offs (between efficiency and robustness) allows for optimal strong and weak scaling when those statistical techniques are used in their appropriate context, namely, when the input data is not quasi-diffuse, but honestly represents discrete measurements. Furthermore, those parallel engines have been successfully applied to the analysis of large-scale turbulent, reacting flow simulation data [BGP+09].

For those reasons, the Titan parallel statistics toolkit appeared as an ideal candidate for our in-situ AMT study.

3 Parallel Statistics in Legion

In this section, we discuss how we transposed the SPMD model used initially for our parallel statistics engines, into an asynchronous many-task model that nonetheless retains the Learn/Derive/Assess work flow. Retaining this flow of operations is important because the original goal of meeting the predominant types of data analysis work flows remain. In addition, the decomposition into independent operations, isolating those parts that do not require global communication from the others, allows for a somewhat elegant transition to an AMT model, as will be seen below.

3.1 Approach

As described in §2, our SPMD models can take two different incarnations, depending on whether the analysis pertains to *moment-based* or *quanta-based* statistics. In the former case, the implementation of the Learn phase relies on an AllGather MPI collective to reduce the distributed local models into the global primary model whereas, in the latter case, this reduction uses a Gather-Broadcast two-step process, for the sake of reducing inter-process communication as the size of local models is not guaranteed to be negligible.

In the AMT model however, it is no longer necessary to express explicit data movement explicitly as in done in the SPMD model, for instance with the dashed red arrow in Figure 2. Considering the same example for the sake of illustration, we can see that task parallelism can readily replace the tasks performed by the distributed processes and the only be used to execute smaller computations on segments of data but that, instead of specifying movement of data from processes to a number of reduction nodes, it is instead sufficient to define a logical region of data where the equivalent of MPI collectives will be performed. The separation of the logical from the physical representation of data inherent to Legion makes it especially apt at supporting this purely data-driven scheme, which should work, albeit maybe not efficiently, with any generic mapper.

Our proposed AMT algorithm for the Learn/Derive/Assess work flow therefore replaces communication with a logical region that contain all model information, both primary and derived, which we call the *aggregation region*. Sub-tasks launched by a top-level task pick up work to on those data segments to which they are assigned, in a similar way to what is done by parallel processes in the SPMD context, at least for the Learn and Assess phases. However, a first noticeable difference is that no communication of data is expressed (even though it will be occur under the hood, but how this happens is entirely the responsibility of the run-time system) but, instead each tasks aggregates the primary statistics it has computed over its data segment, with those already stored in the aggregation region, changing them in-place. As a result, no broadcast of the global primary model is necessary, for all tasks using it (for instance, a new set of Assess sub-tasks) will directly access the aggregate region to retrieve the values of the statistic. Moreover, the Derive operation is now performed by the top-level task, for its results, also stored in the aggregation region, will be also logically available to any Assess tasks launched from the top level. This asynchronous many-task Learn/Derive/Assess scheme is represented in Figure 3, again for the case of order statistics.

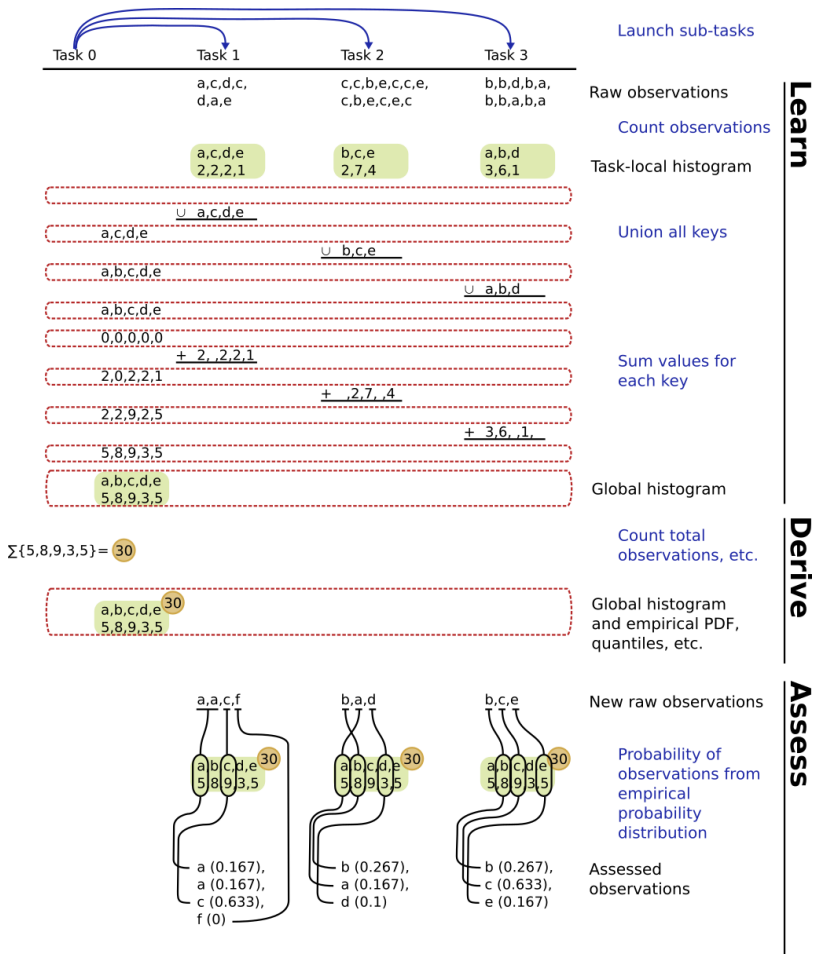

Figure 3. A simplified example illustrating the operations of the task-based order statistics; solid blue arrows indicate task launches, dashed red rectangles represent the logical aggregation region. Sub-tasks of the top-level task are not obligated complete in this order, as both union and addition operators are commutative.

Note that another benefit of this AMT model is that it allows us to unify the two SPMD implementations (AllGather and Gather-Broadcast) into a single paradigm, valid for both quanta-based and moment-based statistics. Also, because all aggregation operations (set unions, number additions and multiplications) are commutative, the learned primary model is guaranteed to be independent of the order in which tasks report their results. However strict locks must be enforced to prevent incomplete model updates: once a task t_i has read the values in the aggregation region, no other task can access this region before t_i has written its own results there.

3.2 Implementation Details

This report only discusses the descriptive statistics implementation of the AMT scheme described in Figure 3, for it is, at the time of writing (November 2015), the only in-situ analysis which we have interfaced with MiniAero-Legion. Prior to extending the list of available analyses, we will develop an in-situ analysis API which will allow such additions to be done in a generic way.

Nevertheless, the above scheme could be used to transform every parallel statistics engine currently available in the statistics filter of VTK into its Legion version. Specifically, the sub-tasks (numbered 1, 2, and 3 in Figure 3) are implemented by means of a class, `Statistics`, itself a sub-class of `LegionData` which is provided by MiniAero-Legion. The task implementation *per se* consists of a method templated on the type of the Legion *accessor* used by the run-time to access data within a physical region. Legion pointers do not directly reference data, but instead name an entry in an index space, and are used when accessing data within accessors for logical regions. In this setting, the accessor is specifically associated with the field being accessed and the pointer names the row entry. Since pointers are associated with index spaces they can be used with an accessor for physical instance. In this way Legion pointers are not tied to memory address spaces or physical instances, but instead can be used to access data for any physical instance of a logical region created with an index space to which the pointer belongs.

Given a Legion `AccessorType`, a concrete instance of `Statistics` provides a function called `learn_statistics_task()`, which implements the Learn operation of the descriptive statistics engine. In this case, the primary statistical model contains the following 8 quantities, stored in double precision: sample size, minimum, maximum, mean, and centered M_2, M_3 and M_4 aggregates (cf. [PÓ8]). The computation of the process-local model is done using the *online* versions of the respective update formulas for the quantities above, whereas the aggregation with the global model is computed by means of the `pairwise` versions; again, cf. [PÓ8] for details. In turn, the member function `learn_statistics()` is responsible for setting up the Legion `IndexLauncher` which sets up all necessary Legion environment so that as many sub-tasks as requested to operate on the partitioned input data be created and executed by the run-time. As part of this process, `learn_statistics()` also creates the Legion *region requirements* needed by the sub-task launcher. Region requirements are Legion objects, used the describe the logical regions requested by launcher objects as well as what privileges and coherence are requested on the specified logical region. In the case of a descriptive statistics Learn task, two region requirements are needed: one for the input data to be read (and only read), and the other for the output region where results are to be aggregated (and which therefore is to be allowed for both read and write access). The top-level task (labeled 0 in Figure 3) is responsible for calling `learn_statistics()`. This will be discussed below in §3.3 as this requires a modification of the driver code, MiniAero in the current study.

In addition, the `Statistics` class provides a member function called `derive_statistics()`, which is to be called only by the top-level task, for it needs only a single pass over the small set of primary statistics in order to compute the derived statistics. Namely, for descriptive statistics, these are the variance, standard deviation, skewness and kurtosis estimators. This operation is de facto negligible for it involves a dozen arithmetic operations computer on a single region, performed by a single task. Note that in this case the Legion region requirements are a read-only access to the first

field (with ID `Statistics::primary_stat_FID`) of the logical region used to store the statistical model, and write permission for the second field (with ID `Statistics::derived_stat_FID`) of the same region. This implements the Derive operation as outlined in Figure 3. Note that the current version of the Derive method only implements the *unbiased sample variance* estimator (cf. [DCD86]) of a multiset $x = \{x_i\}_{i=1}^{i=n}$, with cardinality n greater than 1, defined as follows:

$$s_x^2 = \frac{M_{2,x}}{n-1}$$

where

$$M_{2,x} = \sum_{i=1}^{i=n} (x_i - \overline{x})^2$$

is the second-order aggregate computed by the Learn operation, in a single pass (without prior computation of the mean $\overline{x}$), cf. [PÓ8] for details. The *sample variance*

$$\sigma_x^2 = \frac{1}{n} M_{2,x}$$

is currently not provided by the Legion implementation of the descriptive statistics engine, in contrast with `vtkPDescriptiveStatistics`, and Octave as well, which offer it as an option. This is because the Legion implementation still remains experimental and we have therefore not yet developed an interface supporting multiple user-defined options. We therefore use s_2 in this report, as well as its square root s_x as the standard deviation estimator, for all comparisons against reference values. Similarly, for skewness and kurtosis only the following versions are currently available, respectively:

$$g_{1,x} = \frac{M_{3,x}}{n s_x^3}$$

and

$$g_{2,x} = \frac{M_{4,x}}{n s_x^4}$$

unlike `vtkDescriptiveStatistics` and Octave, for example, which provide additional estimators as well. However, g_1 and g_2 are generally the default formulas employed by statistical software, and these are the ones which are used in this report for baseline comparisons with Octave. Note that we prefer to report the *kurtosis excess*, $g_2 - 3$, rather than raw kurtosis, for the kurtosis of a normal distribution is 3 which allows one to distinguish between *platykurtic* (thin tails; kurtosis defect) and *leptokurtic* (fat tails; kurtosis excess) distributions.

The Assess operation is not implemented at the time of writing, for this study is focused on validating the proposed AMT design: as the Assess phase is purely task-local, it does not bring any additional information in this regard. Because this design appears promising, as is demonstrated subsequently in §4, further work will involve the implementation of a `assess_statistics()` member function of the `Statistics` class, together with a `assess_statistics_task()` static function, to be launched in as many sub-tasks as necessary, similarly to what is done for the Derive operation, aside from the fact that no global aggregation is needed by the Assess operation. Every Assess task will, for descriptive statistics, mark each datum of the data subset to which it is assigned with its relative deviation with respect to the model mean and standard deviation (which

amounts to the one-dimensional Mahalanobis distance from the value predicted by a Gaussian model [Mah36]).

Finally, a few ancillary methods are provided by the `Statistics`. One must be called to initialize the aggregation region, and another one is provided to print out the results. Also, all the necessary boilerplate code required by the Legion mechanisms is implemented, and is not discussed in detail here.

3.3 In-Situ Setting with MiniAero

We now discuss how the Legion version of MiniAero was interfaced with the in-situ AMT statistics code described above.

This required the use of adapator code which we kept as little as possible; as this was a proof-of-concept implementation, it was not designed with any genericity intent in mind, even though consideration should be given to this question in the future, in order to allow the application developer to easy roll in and out different analyses as needed.

The first part of MiniAero/Legion to be modified is its `TaskIDs.h` file, whose enum `TaskIds` defines one index for each task allowed to be created at run-time. Specifically, those added for the sake of the in-situ statistics are the following :

```
initialize_statistics_TID,
learn_statistics_TID,
derive_statistics_TID,
dump_statistics_TID,
```

which respectively identify the statistics initialization, learn, derive and print-out tasks.

In addition, the file `main.cc` must

```
#include "Statistics.h"
```

so that its main routine can call out the static method

```
MiniAero::Statistics::register_tasks();
```

in order to register the tasks corresponding to the IDs listed above.

We also modified the user interface of MiniAero/Legion in order to allow for the specification of additional parameters in the command line arguments of the MiniAero executable. Specifically, we added the 3 following instance variables to `Interface.h`:

```
bool descriptive_statistics;
int statistics_frequency;
int statistics_component;
```

to capture, respectively, whether the in-situ statistics should be executed and, if so, how often and for which variable amongst the 5 components of the MiniAero solution vector. Note that at this point only one variable can be analyzed at a time, but this will be modified in subsequent versions. Furthermore, the implementation of the user interface in `Interface.cc` was modified, first for the constructor which now takes in 3 the additional parameters:

```
Interface::Interface()
 : problem_type(0), num_blocks(0),
   blocks_x(0), blocks_y(0), blocks_z(0),
   interval_x(0), interval_y(0), interval_z(0),
   mesh_scale(1.0), output_frequency(1), time_steps(1),
   length_x(0.0), length_y(0.0), length_z(0.0),
   ramp_angle(0.0), dt(0.0), wait(false), viscous(false),
   second_order_space(false), output_results(false),
   descriptive_statistics(false), statistics_frequency(1), statistics_component(0)
```

with default values so that the analysis is *not* ran unless the user explicitly so requests. The `enroll_options()` function is then modified by incorporating the 3 corresponding option settings:

```
options_.enroll("descriptive_statistics",GetLongOption::NoValue,
                "If specified, will compute in-situ descriptive statistics.",
                NULL);

options_.enroll("statistics_frequency",GetLongOption::MandatoryValue,
                "Number of time steps after which a new statistical
                analysis is performed.",
                NULL);

options_.enroll("statistics_component",GetLongOption::MandatoryValue,
                "Component of solution vector upon which a statistical
                analysis is performed.",
                NULL);
```

which are also added to the parsing method `parse_options()` with, on one hand at the top of the function:

```
descriptive_statistics = options_.retrieve("descriptive_statistics");
```

and, on the other hand at the bottom of the function:

```
{
  const char *temp = options_.retrieve("statistics_frequency");
  if (temp != NULL) {
    statistics_frequency = std::strtol(temp, NULL, 10);
  }
}

{
  const char *temp = options_.retrieve("statistics_component");
  if (temp != NULL) {
    statistics_component = std::strtol(temp, NULL, 10);
  }
}
```

following the format already in place for the existing interface options. Consequently, 3 new optional command line arguments become available; for instance, the following:

```
-descriptive_statistics -statistics_frequency 20 -statistics_component 0
```

when passed to the MiniAero executable, result in the in-situ analysis tasks being invoked (the presence of `-descriptive_statistics` results in the corresponding Boolean to become `true`) every 20 time-steps, for the first component of the solution vector.

The last part of MiniAero/Legion that requires adaptor code is `toplevel.cc`, that also has to

```
#include "Statistics.h"
```

so it can instantiate a `Statistics` object. Prior to that, the interface options related to the in-situ statistics must be retrieved, which is done as follows:

```
bool do_stats = interface.descriptive_statistics;
int stat_freq = interface.statistics_frequency;
int component = interface.statistics_component;
```

in a similar way as what is done for the other interface options. Subsequent versions of the AMT statistics framework will design an interface with more options, modeled after those currently offered by the VTK parallel statistics engines. The `Statistics` object is then instantiated as follows:

```
MiniAero::Statistics statistics(meshdata,ctx,runtime,mesh->num_blocks_,
                                task_wait_all_results,component);
```

which must be done somewhere *before* the driver time-loop. This is in fact similar to what is done for the other main task objects of the Legion version of MiniAero (`bc`, `solver`, `flux`, etc.), and for this reason we placed the `Statistics` instantiation immediately after all those already present in

`toplevel.cc` for these other objects. The aggregation regions defined in §3 must also be created, which requires quite a bit of boilerplate code, which will be abstracted out within the interface in subsequent implementations. For now, one must manually create the aggregation region for primary statistics:

```
Rect<1>
primary_stat_rect(Point<1>(0),Point<1>(MiniAero::Statistics::n_primary_stat-1));
IndexSpace primary_stat_is
  = runtime->create_index_space(ctx,Domain::from_rect<1>(primary_stat_rect));
FieldSpace primary_stat_fs = runtime->create_field_space(ctx);
{
  FieldAllocator allocator
    = runtime->create_field_allocator(ctx,primary_stat_fs);
  allocator.allocate_field(sizeof(double),MiniAero::Statistics::primary_stat_FID);
}
LogicalRegion primary_stat_lr
  = runtime->create_logical_region(ctx,primary_stat_is,primary_stat_fs);
  runtime->attach_name(primary_stat_lr, "primary_stat_lr");
```

and one for derived statistics:

```
Rect<1>
derived_stat_rect(Point<1>(0),Point<1>(MiniAero::Statistics::n_derived_stat-1));
IndexSpace derived_stat_is
  = runtime->create_index_space(ctx,Domain::from_rect<1>(derived_stat_rect));
FieldSpace derived_stat_fs
  = runtime->create_field_space(ctx);
{
  FieldAllocator allocator
    = runtime->create_field_allocator(ctx,derived_stat_fs);
  allocator.allocate_field(sizeof(double),MiniAero::Statistics::derived_stat_FID);
}
LogicalRegion derived_stat_lr
  = runtime->create_logical_region(ctx,derived_stat_is,derived_stat_fs);
  runtime->attach_name(derived_stat_lr, "derived_stat_lr");
```

which will then be passed explicitly to the statistics invocation methods, *inside* the time-loop:

```
if (do_stats && !(n_s % stat_freq)) {
  statistics.initialize_statistics(primary_stat_lr);
  statistics.learn_statistics(MeshData::solution_n,primary_stat_lr);
  statistics.derive_statistics(primary_stat_lr,derived_stat_lr);
  statistics.dump_statistics(primary_stat_lr,derived_stat_lr);
}
```

with the last call being optional, to be used only if a print-out of the computed statistics is sought. This concludes the description of the adapator code that is needed at the time of writing, and

which was used to test out the code, some results of which are presented in the following section. We nonetheless acknowledge that we also instrumented the code with timers, in addition to those already present, to measure performance of the in-situ framework.

4 Results

4.1 Algorithm Correctness

We used the initial mass values of the 3D Sod problem for a $128 \times 4 \times 4$ mesh in order to experimentally assess the correctness of the AMT in-situ statistics described in §3. The statistics tasks thus operated over a multiset $x = \{x_i\}_{i=1}^{i=n}$ comprising the initial density at $n = 128 \times 4 \times 4 = 2048$ mesh nodes. Moreover, the default values provided with the `3D_Sod` case are such that the first half of the domain, along the x-axis, takes on a density value of $a = 0.103930418401153$, while the remaining half is set at $b = 0.831439030101197$. Therefore, a and b respectively are the minimum and maximum reference values, while $\frac{a+b}{2}$ is the reference mean. In addition, because x is symmetric about its mean, its skewness is 0. Regarding the second and fourth order aggregates, one has

$$\begin{aligned} M_{2,x} &= \sum_{i=1}^{n/2} \left(a - \frac{a+b}{2}\right)^2 + \sum_{i=1}^{n/2} \left(b - \frac{a+b}{2}\right)^2 \\ &= \sum_{i=1}^{n/2} \left(\frac{a-b}{2}\right)^2 + \sum_{i=1}^{n/2} \left(\frac{b-a}{2}\right)^2 \\ &= n \left(\frac{a-b}{2}\right)^2 \end{aligned}$$

and, similarly,

$$M_{4,x} = n \left(\frac{a-b}{2}\right)^4 .$$

As a result, the reference values for unbiased sample variance, and kurtosis, are as follows:

$$s_x^2 = \frac{n}{n-1} \left(\frac{a-b}{2}\right)^2$$

and

$$g_{2,x} = \frac{M_{4,x}}{n\, s_x^4} - 3 = \left(\frac{n-1}{n}\right)^2 - 3.$$

For the sake of concision, we only used these 7 statistics, and not all 11 available primary and derived statistics. However, those we retained are sufficient for there is redundancy between M_2, variance and standard deviation; M_3 and skewness; and M_4 and kurtosis excess. In any event, those statistics retained for comparison studies span across primary and derived quantities, so that both Learn and Derive phases were verified.

Subsequently, MiniAero-Legion was executed with a single in-situ analysis call, performed at the initial time step, for a variable number of tasks ranging from 1 to 64 (the block decomposition scheme used by MiniAero imposes that number of mesh sub-blocks be a power of 2). This was done using the shared-memory-only low-level run-time, as recommended by the Legion team for initial experiments.

Table 1. Computed descriptive statistics of mass at the initial time step of the `3D_Sod` test case, *versus* reference values.

Statistic	Reference	1 task	4 tasks	64 tasks	Octave
Cardinality	2048	2048	2048	2048	2048
Minimum	0.1039304	0.1039304	0.1039304	0.1039304	0.1039304
Maximum	0.8314390	0.8314390	0.8314390	0.8314390	0.8314390
Mean	0.4676847	0.4676847	0.4676847	0.4676847	0.4676847
Std. dev.	0.3638431	0.3638431	0.3638431	0.3638431	0.3638431
Skewness	0	-3.2766E-16	0	3.6015E-17	-1.9708E-13
Kurtosis	-2.0009763	-2.0009763	-2.0009763	-2.0009763	-2.0000000

With those reference values at hand, computed with the same precision as that of the input values a and b, we also compared out of curiosity our results with those obtained with the statistics functions of Octave. We compare the calculated results to the reference values in Table 1. To avoid visual clutter we only report 8 digits in the table.

Skewness aside, all computed in-situ statistics agree with their respective reference values, not only for the 8 digits shown here, but also for all 15 significant digits permitted by the floating point representation in double precision. Comparing the computed values skewness *versus* the reference skewness is much more complicated, because the latter is 0, as testing equality with 0 is a very complex question in floating point representation. It is beyond the scope of this report to discuss it in detail, but suffices to remember that, short of using Units of Last Place comparison, the next best option is absolute comparison. With this criterion, the in-situ computations fluctuated about 0 within a radius less than machine ε (*ca.* 2.22E-16), with even some cases (as reported here with 4 tasks) finding a true 0. In addition, the skewness estimate computed by Octave was 3 orders of magnitude larger than machine ε.

Although we did not research this further, this is most probably caused by destructive cancellations, for skewness computations for symmetric data in floating point representation involves mutual near-cancellations between quantities which are, in relative terms, indistinguishable within machine ε. This is confirmed with kurtosis (excess), where our in-situ AMT implementation not only exactly finds (with all significant digits) the theoretical reference value, but also beats Octave which is only correct for the first 4 significant digits. The fact that our implementation always performs better than Octave (as well as SciPy, as we verified separately) for third and fourth order statistics is further vindication of our point-wise and pairwise update formulas first used in `vtkPDescriptiveStatistics`.

4.2 Parallel Scalability

In order to conduct scalability studies independently of load-balancing issues, we used the same 3D Sod test case as in §4.1, for the block-decomposition scheme results in a perfectly balanced load per task (which must be a power of 2) when the number of nodes is itself a power of 2. With this setting, we used the following evaluation criteria:

1. *Strong scaling*, i.e., at constant total work, also known as relative speed-up. Denoting $T_N(p)$ the wall clock time measured to execute the calculation, strong scaling is defined as:

$$S_N(p) = \frac{T_N(1)}{T_N(p)}.$$

   Some authors prefer to write the numerator as T_s rather than $T_N(1)$ to make it clear that the parallel algorithm should be compared to the most efficient serial implementation available and not just the parallel algorithm run on a single task. Evidently, optimal (linear) scaling is attained when $S_N(p) = p$ and, therefore, strong scaling results can be visually inspected by plotting S_N versus the number of tasks: optimal scaling is revealed by a line, the angle bisector of the first quadrant.

2. *Weak scaling*, i.e., at constant work per task, also known as rate of computation scalability. The rate of computation is defined as:

$$r(p) = \frac{N(p)}{T_{N(p)}(p)},$$

   where $N(p)$ now varies with p. Weak scaling is then measured by normalizing the rate of computation by that which is obtained with a single task. In particular, if the sample size is made to vary in proportion to the number of tasks, i.e., if $N(p) = pN(1)$, then

$$R(p) = \frac{r(p)}{r(1)} = \frac{pT_{N(1)}(1)}{T_{pN(1)}(p)} = \frac{pT_{N(1)}(1)}{pT_{N(1)}(p)} = \frac{T_{N(1)}(1)}{T_{N(1)}(p)}.$$

   Therefore, optimal (linear) scaling is attained with p tasks when $R(p) = p$. Note that without linear dependency between N and p, the latter equality no longer implies optimal scalability. Parallel scalability can thus also be visually inspected by plotting the values of R versus the number of tasks, and in this case also, optimality corresponds to the angle bisector of the first quadrant.

The information about time spent by each of the considered tasks can be extracted from the profiling output of MiniAero/Legion, using the Python script called `legion_prof.py`, which comes with the distribution of Legion. This script parses the performance output that is generated when the appropriate option flags are passed to the executable through the command line.

In order to verify that our concept of *aggregation region* introduced in §3.1 as an AMT surrogate for SPMD global reduction, a first step was to validate that some scaling could be obtained

when using 1 or 2 tasks on a 2-core machine. If this was not the case, an entire re-evaluation of the whole approach would have been warranted. With this in mind, the first scalability results presented hereafter were obtained on a system with a dual-core 2.3GHz Intel i7 CPU running Mac OS v14.3.0, with MiniAreo/Legion built together with the in-situ statistics engine using shared memory for the low-level RTS. The reference case comprised $128 \times 32 \times 32$ grid points, each of which providing a value included in the input data set for the in-situ statistical analysis, performed by a single Learn task. In order to assess strong scaling, the same mesh was analyzed by 2 in-situ Learn tasks, whereas weak scaling was measured also with 2 Learn tasks but on a grid twice comprising twice as many nodes, specifically $256 \times 32 \times 32$.

Table 2. Learn operation: strong scaling at constant total work with sample size N (left), and weak scaling at constant load of per task (right).

N/p	p	T (μsec.)	$S_N(p)$)
131072	1	1613	1.00
65536	2	918	1.76

$N(p)$	p	T (μsec.)	$R(p)$
131072	1	1613	1.00
262144	2	1793	1.80

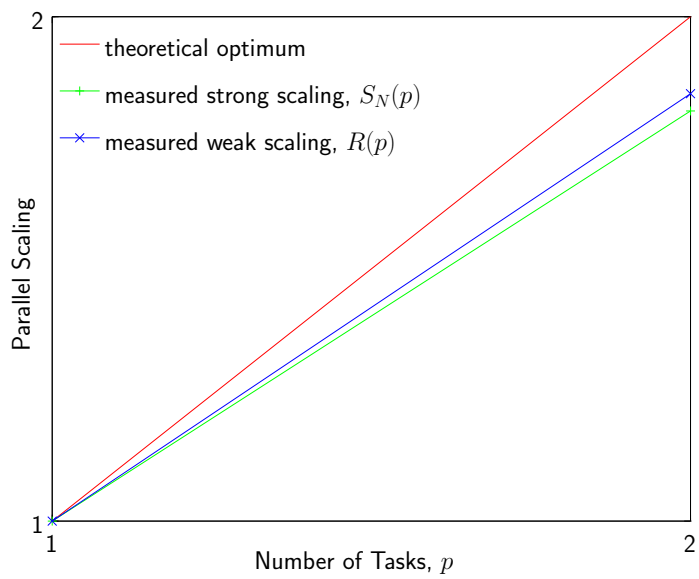

Figure 4. Learn operation: strong and weak scaling with respect to single task with $N = 131072$ values.

Table 2 summarizes the findings obtained with this setting, in terms of both strong (left) and

weak (right) scaling. A first observation was observed timings displayed noticeable fluctuations, for any given setting. Those fluctuations could represent almost a doubling of observed values, between their respective minima and maxima. It is difficult to explain such variations without specific investigation of this issue. Nonetheless, a plausible cause is that of OS jitters on a 2-core platform, where the multiplicity of background tasks prevent full utilization of the 2 cores by 2 simultaneous tasks. Each of the 3 test cases was therefore ran 8 times, and the smallest timing was retained. In each case, measured timings clustered in the vicinity of the smallest one, with only a few outliers taking much longer to complete. This finding supports the hypothesis that OS-related issues explain much of the observed timing variability.

With this convention, the computed scaling factors are both in the order of 1.8, with weak scaling being only marginally greater than strong scaling. As illustrated by Figure 4, this is sub-optimal but nonetheless promising, considering the fact that the underlying Legion build was not optimized in any way for the target platform. At least, these first findings confirm that our design, described in §3, is worth being evaluated further.

Note that, on the other hand, the Derive operation took *ca.* 31μsec. in all cases, irrespective of problem size. This finding further confirms that the Derive operation scales optimally in a distributed setting when used for quanta-based statistics. We shall study later whether this is also observed for the Assess operation, as one would expect based on theory and practice in the SMPD model.

5 Conclusion

In this report, we proposed a framework for the design and implementation of in-situ analyses using an asynchronous many-task (AMT) model. This framework was implemented using the Legion programming model, using a mini-application as a surrogate for full-scale parallel scientific computing applications.

The main contribution is the conversion into an AMT model of the Learn/Derive/Assess model which we had initially developed in the context of a single-program multiple-data (SPMD) model. In this goal, we proposed to replace the Allgather (or the Gather/Broadcast) pattern used by SPMD version of the Learn operation (the only part of the Learn/Derive/Assess that requires inter-process or inter-task communication) with a Legion logical region called an *aggregation region*. The responsibility to maintain the consistency of the aggregation region therefore falls upon the Legion run-time system (RTS). The RTS thus becomes responsible for the parallel scalability of the AMT in-situ analysis: if it cannot manage efficiently the access (both read and write) to the aggregation from multiple tasks distributed over a large system, then parallel scaling is impossible.

Therefore, our proposed method appears to be a useful exercise for two reasons:

1. it allows for the evaluation of various in-situ analysis designs; and
2. it efficiently tests parallel scalability of Legion RTS upon various hardware platforms.

Indeed, the first tests performed using the shared-memory back-end of the low-level RTS show promising scalability results. However, these results do not show optimal scale-up even at the small scales tested so far, even for a type of analysis (*moment-based*) for which we had earlier demonstrated optimal scaling properties with up to 10^5 cores in a SMPD implementation. This shows that further work is needed in terms of Legion RTS optimization, some of which might be the responsibility of the application developer, (i.e. changes to the mapper). Further work will investigate this issue, in collaboration with the Legion team.

In addition, we will develop another in-situ analysis engine that pertains to the class of *quanta-based statistics*, which will further test the parallel scaling capabilities of Legion logical regions, for the reasons which we discussed at length in the SPMD context, as a result of the variable-size of the quantized model which can be as large as the data set itself.

Finally, we need to investigate the Assess operation of the Learn/Derive/Assess analysis triptych, which we have voluntarily left aside in this study which primarily focused on the in-situ computation of a model from data in an AMT programming model.

References

[BC+15] J. Bennett, R. Clay, et al. ASC ATDM Level 2 Milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms. Sandia Report SAND2015-8312, Sandia National Laboratories, https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/ATDM-AMT-L2-Final-SAND2015-8312.pdf, September 2015.

[BGP+09] J. Bennett, R. Grout, P. Pébay, D. Roe, and D. Thompson. Numerically stable, single-pass, parallel statistics algorithms. In *Proc. 2009 IEEE International Conference on Cluster Computing*, New Orleans, LA, U.S.A., August 2009.

[DCD86] D. Dacunha-Castelle and M. Duflo. *Probability and Statistics*, volume 1. Springer-Verlag, 1986.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, December 2004.

[ETS14] H.C. Edwards, C.R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[FFLB15] K.J. Franko, T.C. Fischer, P. Lin, and S.W. Bova. CFD for next generation hardware: Experiences with proxy applications. In *Proc. 22nd AIAA Computational Fluid Dynamics Conference*, Dallas, TX, June 2015.

[Kit10] Inc. Kitware. *The VTK User's Guide, version 5.4*. Kitware, Inc., 2010.

[Mah36] P. C. Mahalanobis. On the generalised distance in statistics. *Proc. of the National Institute of Science of India*, (12):49–55, 1936.

[man] Mantevo. Online.

[PÓ8] P. Pébay. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. Sandia Report SAND2008-6212, Sandia National Laboratories, September 2008.

[PTB10] P. Pébay, D. Thompson, and J. Bennett. Computing contingency statistics in parallel: Design trade-offs and limiting cases. In *Proc. 2010 IEEE International Conference on Cluster Computing*, Heraklion, Crete, Greece, September 2010.

[PTBM11] P. Pébay, D. Thompson, J. Bennett, and A. Mascarenhas. Design and performance of a scalable, parallel statistics toolkit. In *Proc. 25th IEEE International Parallel & Distributed Processing Symposium, 12th International Workshop on Parallel and Distributed Scientific and Engineering Computing*, Anchorage, AK, U.S.A., May 2011.

[Sod78] G. A. Sod. A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. *J. Comput. Phys.*, (27):1–31, 1978.

[WB09] B. Wylie and J. Baumes. A unified toolkit for information and scientific visualization. In *Proc. SPIE 7243, Visualization and Data Analysis*, San Jose, CA, U.S.A., January 2009.

DISTRIBUTION:

2	MS 9018	Central Technical Files, 8944
1	MS 0899	Technical Library, 9536
5	MS 9159	Philippe Pébay, 8953

Sandia National Laboratories