SANDIA REPORT

SAND2008-8260
Unlimited Release
Printed December 2008

Scalable Descriptive and Correlative Statistics with Titan

Philippe Pébay, David Thompson

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

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

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

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

SAND2008-8260
Unlimited Release
Printed December 2008

Scalable Descriptive and Correlative Statistics with Titan

Philippe Pébay
Sandia National Laboratories
M.S. 9159, P.O. Box 969
Livermore, CA 94551, U.S.A.
pppebay@sandia.gov

David Thompson
Sandia National Laboratories
M.S. 9159, P.O. Box 969
Livermore, CA 94551, U.S.A.
dcthomp@sandia.gov

Abstract

This report summarizes the existing statistical engines in VTK/Titan and presents the parallel versions thereof which have already been implemented. The ease of use of these parallel engines is illustrated by the means of C++ code snippets. Furthermore, this report justifies the design of these engines with parallel scalability in mind; then, this theoretical property is verified with test runs that demonstrate optimal parallel speed-up with up to 200 processors.

Acknowledgments

The authors would like to thank:

- Brian Wylie, for his comments on the integration of scalable statistical tools in VTK/Titan,
- Ken Moreland, for having reviewed this report and having helped us better assess parallel speed-up and scalability properties,
- Jackson Mayo, for his precursor work on a serial version of the multi-correlative statistics algorithm.

Contents

This page intentionally left blank

1 Introduction

1.1 The Titan Informatics Toolkit

The Titan Informatics Toolkit is a collaborative effort between Sandia National Laboratories and Kitware Inc. It represents a significant expansion of the Visualization ToolKit (VTK) to support the ingestion, processing, and display of informatics data. By leveraging the VTK engine, Titan provides a flexible, component based, pipeline architecture for the integration and deployment of algorithms in the fields of intelligence, semantic graph and information analysis.

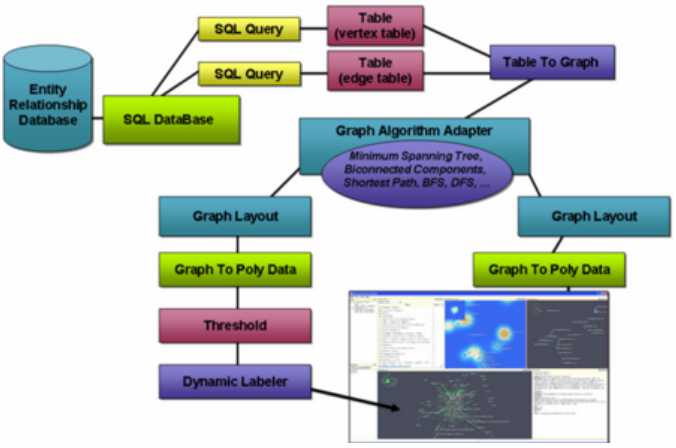

Figure 1. A theoretical application built with Titan.

A theoretical application built from Titan/VTK components is schematized in Figure 1. The flexibility of the pipeline architecture allows effective utilization of the Titan components for different problem domains. An actual implementation is OverView, a generalization of the ParaView scientific visualization application to support the ingestion, processing, and display of informatics data. The ParaView client-server architecture provides a mature framework for performing scalable analysis on distributed memory platforms, and OverView will use these capabilities to analyze informatics problems that are too large for individual workstations.

The Titan project represents one of the first software development efforts to address the merging of scientific visualization and information visualization on a substantive level. The VTK parallel client-server layer will provide an excellent framework for doing scalable analysis on distributed memory platforms. The benefits of combining the two fields are already reaping rewards in the form of functionality such as the cell lineage application below.

1.2 Statistics Functionality in Titan

A number of univariate, bivariate, and multivariate statistical tools have been implemented in Titan. Each tool acts upon data stored in one or more tables; the first table serves as observations and further tables serves as model data. Each row of the first table is an observation, while the form of further tables depends on the type of statistical analysis. Each column of the first table is a variable.

1.2.1 Variables

A univariate statistics algorithm only uses information from a single column and, similarly, a bivariate from 2 columns. Because an input table may have many more columns than an algorithm can make use of, Titan must provide a way for users to denote columns of interest. Because it may be more efficient to perform multiple analyses of the same type on different sets of columns at once as opposed to one after another, Titan provides a way for users to make multiple analysis requests of a single filter.

Table 1. A table of observations that might serve as input to a statistics algorithm.

row	A	B	C	D	E
1	0	1	0	1	1.03315
2	1	2	2	2	0.76363
3	0	3	4	6	0.49411
4	1	5	6	24	0.04492
5	0	7	8	120	0.58395
6	1	11	10	720	1.66202

As an example, consider Table 1. It has 6 observations of 5 variables. If the correlations between A, B, and C, and also between B, C and D are desired, two requests, R_1 and R_2 must be made. The first request R_1 would have columns of interest $\{A,B,C\}$ while R_2 would have columns of interest $\{B,C,D\}$. Calculating covariances for R_1 and R_2 in one pass is more efficient than computing each separately since $\mathrm{cov}(B,B)$, $\mathrm{cov}(C,C)$, and $\mathrm{cov}(B,C)$ are required for both requests but need only be computed once.

1.2.2 Phases

Each statistics algorithm performs its computations in a sequence of common phases, regardless of the particular analysis being performed. These phases can be described as:

Learn: Calculate a "raw" statistical model from an input data set. By "raw", we mean the minimal representation of the desired model, that contains only primary statistics. For example, in

the case of descriptive statistics: sample size, minimum, maximum, mean, and centered M_2, M_3 and M_4 aggregates (*cf.* [Pó8]). For Table 1 with a request $R_1 = \{B\}$, these values are 6, 1, 11, $4.8\overline{3}$, $68.8\overline{3}$, $159.\overline{4}$, and $1759.819\overline{4}$, respectively.

Derive: Calculate a "full" statistical model from a raw model. By "full", we mean the complete representation of the desired model, that contains both primary and derived statistics. For example, in the case of descriptive statistics, the following derived statistics are calculated from the raw model: unbiased variance estimator, standard deviation, and two estimators (g and G) for both skewness and kurtosis. For Table 1 with a request $R_1 = \{B\}$, these additional values are $13.7\overline{6}$, 3.7103, 0.520253, 0.936456, -1.4524, and -1.73616 respectively.

Assess: Given a statistical model – from the same or another data set – mark each datum of a given data set. For example, in the case of descriptive statistics, each datum is marked with its relative deviation with respect to the model mean and standard deviation (this amounts to the one-dimensional Mahalanobis distance). Table 1 shows this distance for $R_1 = \{B\}$ in column E.

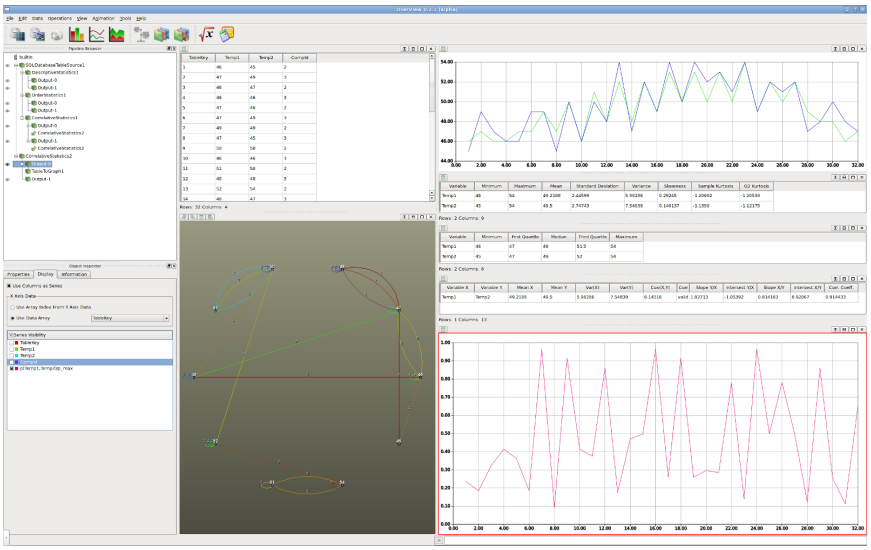

Figure 2. An example utilization of Titan's statistics algorithms in OverView.

An example of the utilization of Titan's statistical tools in OverView is illustrated in Figure 2; specifically, the descriptive, correlative, and order statistics classes are used in conjunction with various table views and plots. With the exception of contingency statistics which can be performed on any type (nominal, cardinal, or ordinal) of variables, all currently implemented algorithms require cardinal or ordinal variables as inputs.

At the time of writing, the following algorithms are available in Titan:

1. Univariate statistics:

(a) Descriptive statistics:

Learn: calculate minimum, maximum, mean, and centered M_2, M_3 and M_4 aggregates;

Derive: calculate unbiased variance estimator, standard deviation, skewness (1_2 and G_1 estimators), kurtosis (g_2 and G_2 estimators);

Assess: mark with relative deviations (one-dimensional Mahlanobis distance).

(b) Order statistics:

Learn: calculate histogram;

Derive: calculate arbitrary quartiles, such as "5-point" statistics (quartiles) for box plots, deciles, percentiles, etc.;

Assess: mark with quartile index.

2. Bivariate statistics:

(a) Correlative statistics:

Learn: calculate minima, maxima, means, and centered M_2 aggregates;

Derive: calculate unbiased variance and covariance estimators, Pearson correlation coefficient, and linear regressions (both ways);

Assess: mark with squared two-dimensional Mahlanobis distance.

(b) Contingency statistics:

Learn: calculate contingency table;

Derive: calculate joint, conditional, and marginal probabilities, as well as information entropies;

Assess: mark with joint and conditional PDF values.

3. Multivariate statistics:

These filters all accept multiple requests R_i, each of which is a set of n_i variables upon which simultaneous statistics should be computed.

(a) Multi-Correlative statistics:

Learn: calculate means and pairwise centered M_2 aggregates;

Derive: calculate the upper triangular portion of the symmetric $n_i \times n_i$ covariance matrix and its (lower) Cholesky decomposition;

Assess: mark with squared multi-dimensional Mahlanobis distance.

(b) PCA statistics:

Learn: identical to the multi-correlative filter;

Derive: everything the multi-correlative filter provides, plus the n_i eigenvalues and eigenvectors of the covariance matrix;

Assess: perform a change of basis to the principal components (eigenvectors), optionally projecting to the first m_i components, where $m_i \leq n_i$ is either some user-specified value or is determined by the fraction of maximal eigenvalues whose sum is above a user-specified threshold. This results in m_i additional columns of data for each request R_i.

In the following sections, we present the currently available parallel version of the aforementioned algorithms, provide a basic user manual of these, and examine their correctness as well as their parallel speed-up properties.

2 Method

2.1 Parallel Statistics Classes

The main reason we split the process of creating a full statistical model into two phases is parallel computational efficiency, so that inter-processor communication and updates are performed only for primary statistics; derived statistics need only be calculated at once, without communication, upon completion of all parallel updates of primary variables. The calculations to obtain derived statistics from primary statistics are usually fast and simple. When assessing data, we assume that the data to be assessed is distributed in parallel across all processes participating in the computation, so no communication is required – every process assess all its resident data.

Therefore, the only part of the parallel versions of the statistical engines that involve inter-processor communication is the Learn phase, whereas both Derive and Assess are executed in an embarrassingly parallel fashion thanks to data parallelism. This approach is consistent with the data parallelism methodology used to enable parallelism with VTK, most notably in ParaView. Because the focus of this report is on the parallel speed-up properties of statistics engines, it is therefore not necessary to report on the Derive or Assess phases, as these are executed independently from each other, on a separate process for each part of the data partition. However, because the Derive phase provides the derived quantities to which one is naturally accustomed (e.g., variance as opposed to M_2 aggregate), the numerical results reported here are those that are yielded by the consecutive application of the Learn and then Derive phases.

At this point (December 2008) of the development of scalable statistics algorithms in Titan, the following 3 parallel classes are implemented:

1. `vtkPDescriptiveStatistics`;
2. `vtkPCorrelativeStatistics`;
3. `vtkPMultiCorrelativeStatistics`.

These are the classes whose parallel efficiency and numerical accurracy we will discuss.

2.2 Usage

It is fairly easy to use the serial statistics classes of Titan; it is not much harder to use their parallel versions. All it takes is a parallel build of Titan and a version of MPI installed on your system.

For example, Listing 1 shows what is required to calculate descriptive statistics, in parallel, on each column of an input set `inputData` of type `vtkTable*`, with no subsequent data assessment. Note that if, instead, the Assess phase were turned on with `pds->SetAssess( true )` then, by default,

```
void Foo( vtkMultiProcessController* controller, void* arg )
{
  // Use the specified controller on all parallel filters by default:
  vtkMultiProcessController::SetGlobalController( controller );

  // Assume the input dataset is passed to us:
  vtkTable* inputData = static_cast<vtkTable*>( arg );

  // Create parallel descriptive statistics class
  vtkPDescriptiveStatistics* pds = vtkPDescriptiveStatistics::New();

  // Set input data port
  pds->SetInput( 0, inputData );

  // Select all columns in inputData
  for ( int c = 0; c < inputData->GetNumberOfColumns(); ++ c )
    {
    pds->AddColumn( inputData->GetColumnName[c] );
    }

  // Calculate statistics with Learn and Derive phases only
  pds->SetLearn( true );
  pds->SetDerive( true );
  pds->SetAssess( false );
  pds->Update();
}
```

Listing 1: A subroutine – that should be run in parallel – for calculating descriptive statistics.

unsigned deviations (Mahalanobis distance) would be calculated. To obtained signed deviations, call `pds->SignedDeviationsOn()`.

For univariate statistics algorithms, calling `AddColumn()` for each column of interest sufficient – each request R_i can by definition only reference a single column and so the filter automatically turns each column of interest into a separate request. Similarly, bivariate algorithms need only call `AddColumn()` an even number of times to unambiguously specify a set of requests. However, this is not sufficient for multivariate filters as each request might have a different number of columns of interest. In order to queue a request for multivariate statistics algorithms, call `SetColumnStatus()` to turn on columns of interest (and to turn off any previously-selected columns that are not of interest) and then call `RequestSelectedColumns()`. Consider the example from §1.2.1 and Table 1 where 2 requests are mentioned: $\{A,B,C\}$ and $\{B,C,D\}$. The code snippet in Listing 2 shows how to queue these requests for a `vtkPMultiCorrelativeStatistcs` object.

Apart from queuing requests, using the other parallel statistics engines only differs in the fact that the option to elect signed deviations as opposed to the Mahlanobis distance for the assessment

```
vtkPMultiCorrelativeStatistics* pms = vtkPMultiCorrelativeStatistics::New();

// Turn on columns of interest
pms->SetColumnStatus( "A", 1 );
pms->SetColumnStatus( "B", 1 );
pms->SetColumnStatus( "C", 1 );
pms->RequestSelectedColumns();

// Columns A, B, and C are still selected, so first we turn off
// column A so it will not appear in the next request.
pms->SetColumnStatus( "A", 0 );
pms->SetColumnStatus( "D", 1 );
pms->RequestSelectedColumns();
```

Listing 2: An example of requesting multiple multi-variate analyses.

of data is available only to the case of descriptive statistics; the concepts of left and right do not extend to dimensions higher than one.

The listings above all assume that you have already prepared an MPI communicator, loaded a dataset into the `inputData` object, and are running in a parallel environment. It is outside the scope of this report to discuss I/O issues, and in particular how a `vtkTable` can be created and filled with the values of the variables of interest. See VTK's online documentation for details [vtk]. However, we will include below a small amount of code to prepare a parallel controller.

The `vtkMultiProcessController` object passed to `Foo()` is used to determine the set of processes (which may be a subset of a larger job) among which input data is distributed. Subroutines of this form are used by VTK to specify which code should be executed across many processes. In order to execute `Foo()` in parallel using MPI, one must first (e.g., in the main routine), create a `vtkMPIController` and pass it the address of `Foo()` as shown in Listing 3. Note that, when using MPI, the number of processes is determined by the external program which launches this application.

```
vtkTable* inputData;
vtkMPIController* controller = vtkMPIController::New();
controller->Initialize( &argc, &argv );

// Execute the function named Foo on all processes
controller->SetSingleMethod( Foo, &inputData );
controller->SingleMethodExecute();

// Clean up
controller->Finalize();
controller->Delete();
```

Listing 3: A snippet of code to show how to execute a subroutine (`Foo()`) in parallel. In reality, `inputData` would be prepared in parallel by `Foo()` but is assumed to be pre-populated here to simplify the example.

3 Results

The parallel runs have been executed on Sandia National Laboratories' `catalyst` computational cluster, which comprises 120 dual 3.06GHz Pentium Xeon compute nodes with 2GB of memory each. This cluster has a Gigabit Ethernet user network for job launch, I/O to storage, and user interaction with jobs, and a 4X Infiniband fabric high-speed network using a Voltaire 9288 InfiniBand switch. Its operating system has a Linux 2.6.17.11 kernel, and its batch scheduling system is the TORQUE resource manager [tor].

3.1 Algorithm Scalability

In order to assess speed-up independently of the load-balancing scheme, a series of (pseudo-) randomly-generated samples is used. Specifically, input tables are created at run time by generating 4 separate samples of independent pseudo-random variables, the two first (resp. last) variables having a standard normal (resp. standard uniform) distribution. Since our objective is to assess the scalability of the parallel statistics engines only, equally-sized slabs of data are created by each process in order to work with perfectly load-balanced cases. For the same reason, the amount of time needed to create the input data table is excluded from the analysis. In this test, `vtkPDescriptiveStatistics`, with Learn, Derive, and Assess modes on, is executed for each of the 4 columns, and the corresponding wall clock time is reported. Subsequently, `vtkPCorrelativeStatistics`, with Learn, Derive, and Assess modes turned on is executed on a single pair of columns (standard normal ones), and the corresponding wall clock time is also reported.

With these synthetic examples, we assess:

1. relative speed-up (at constant total work), and

2. scalability of the rate of computation (at constant work per processor).

3.1.1 Relative Speed-Up

Given a problem of size N (as measured in our case by sample size), the wall clock time measured to complete the work with p processors is denoted $T_N(p)$. Then, relative speed-up with p processors is

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

Evidently, optimal (linear) speedup is attained with p processors when $S_N(p) = p$ and, therefore, relative speed-up results for S_N may be visually inspected by plotting S_N *versus* the number of processors: optimal speed-up is revealed by a line, the angle bisector of the first quadrant.

Table 2. Relative speed-up (at constant total work), with a total sample size of $N = 25,600,000$.

N/p	p	Descriptive (sec. / $S_N(p)$)	Correlative (sec. / $S_N(p)$)
25,600,000	1	75 / 1.00	58 / 1.00
12,800,000	2	38 / 1.97	30 / 1.93
6,400,000	4	19 / 3.95	14 / 4.14
3,200,000	8	9 / 8.33	8 / 7.25
1,600,000	16	5 / 15.0	4 / 14.5
800,000	32	2 / 37.5	2 / 29.0

In the first series of test runs, in order to assess relative speed-up, the sample is subdivided into 4 columns of size $6,400,000$. Thus, the input data of the entire test case contains a total of $N = 25,600,000$ values. The values of p were chosen to be increasing powers of 2, for convenience only: making use of other values did not modify speed-up results. The results obtained on `catalyst` are provided in Table 2, and plotted in Figure 3.

As expected based on the embarrassingly parallel nature of the algorithms, the measured relative speed-up is optimal (within $\pm 10\%$ fluctuations attributable to OS jitter and such), until total wall time measurements become too small to remain accurate (less than 1 sec.), and the decreasing amount of work per processor ultimately results in a situation where overheads, even small in absolute terms, become dominant as compared to the amount of actual computational work. In this current example, it appears that with 32 processors, minimal reliably measurable wall clock time has been or is almost reached. Note that this corresponds to a per processor load of $N/p = 800,000$ points per processor.

3.1.2 Rate of Computation Scalability

The rate of computation is defined as

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

where $N(p)$, the sample size, now varies with the number of processors p. We then measure its scalability by normalizing it with respect to the rate of computation obtained with a single processor, as follows:

$$R(p) = \frac{r(p)}{r(1)} = \frac{N(p)T_{N(1)}(1)}{N(1)T_{N(p)}(p)},$$

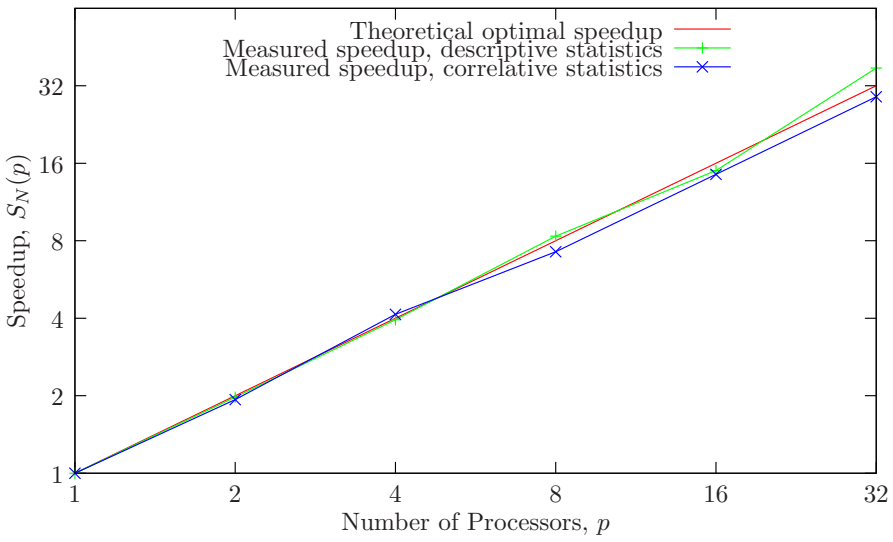

Figure 3. Relative speed-up at constant total work with a total data size of $N = 25,600,000$.

In particular, if the sample size is made to vary in proportion to the number of processors, i.e., if $N(p) = pN(1)$, then

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

and thus, optimal (linear) scalability is also attained with p processors when $R(p) = p$. Note that without linear dependency between N and p, the latter equality no longer implies optimal scalability. Hence, under the above assumptions, scalability can also be visually inspected, with a plot of R *versus* the number of processors, where optimal scalability is also indicated by the angle bisector of the first quadrant.

In order to assess rate of computation scalability (at constant work per processor), increasingly large samples are created, containing np quadruples, where $n = 10^6$ and $p \in \{1,2,4,8,16,32,64\}$ respectively denote the number of sample points per processor, and the number of processors, thus resulting in a total sample size of $N(p) = 4np$. Note that whether one or two cores per node are occupied by the np processes in each case is left for the scheduler to decide; forcing all cluster nodes to utilize either exactly one, or exactly two of their cores dit did not result in a measurable difference.

In each case, a table of size $n \times 4$ is created by each process. Corresponding wall clock times measured on `catalyst` are given in Table 3, and plotted in Figure 4 ; these clearly exhibit optimal scalability (again within $\pm 10\%$ fluctuations attributable to OS jitter and such), thus experimentally verifying the embarrassingly parallel nature of these algorithms. It is also worth noting that using 1 or 2 cores per node did not result in any measurable difference.

Table 3. Rate of computation scalability (at constant load per processor).

$N(p)$	p	Descriptive (sec. / R)	Correlative (sec. / R)
4,000,000	1	12 / 1.00	10 / 1.00
8,000,000	2	12 / 2.00	10 / 2.00
16,000,000	4	12 / 4.00	10 / 4.00
32,000,000	8	12 / 8.00	9 / 8.89
64,000,000	16	13 / 14.8	10 / 16.0
128,000,000	32	13 / 29.5	10 / 32.0
256,000,000	64	13 / 59.1	10 / 64.0
512,000,000	128	13 / 118	10 / 128
800,000,000	200	13 / 185	10 / 200

3.2 Algorithm Correctness

In order to assess algorithm correctness, we make use of the same test cases as § 3.1, for which we inspect the numerical results obtained by both the `vtkPDescriptiveStatistics` and the `vtkPCorrelativeStatistics` classes. More precisely, we examine the statistical models obtained when both Learn and Derive options are turned on. Since the statistical properties of the test cases are known, we can immediately compare them to the calculated results.

Relatively large input sets are used ($n = 10^6$), in order to mitigate the risk of statistical bias due to insufficient sampling. In addition, the test case is run 100 times for each random variable, and we examine the statistical dispersion of the results of the ensemble of these runs. We compare the results obtained with the Learn and Derive option of the statistical engines to the theoretical values of the random variables which serve as models for the pseudo-random inputs, namely, $\mathscr{N}(0,1)$ and $\mathscr{U}(0,1)$. This comparison is done by simple visual inspection of the numerical results, by:

1. comparing the sample mean of the quantity of interest (e.g., mean) across the a number n_r of runs to the corresponding theoretical quantity (e.g, expectation), and

2. examining the variability of the results by checking the standard deviation of the quantity of interest across the n_r runs.

Using this methodology with either $n_r = 100$ or $n_r = 200$ runs over 32 processors, the results provided in Table 4, Table 5, Table 6, and Table 7, respectively for `vtkPDescriptiveStatistics` and `vtkPDescriptiveStatistics` operating on standard uniform and standard normal pseudo-

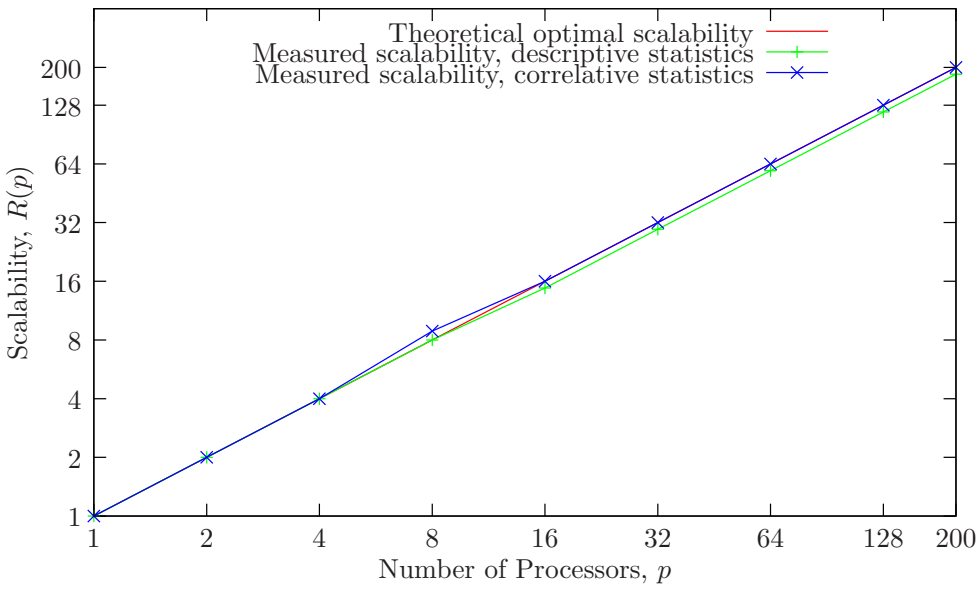

Figure 4. Rate of computation scalability at constant work per processor of $N(p)/p = 4,000,000$.

random inputs, we see that the numerical results are in statistical agreement with their theoretical counterparts, and display very limited variability across runs.

Finally, the last series checks we performed for the Learn and Derive modes consisted in verifying that the parallel descriptive, correlative, and multi-correlative classes indeed calculated the same results for those statistics they have in common: means, variances, and covariances (for the correlative classes only). And indeed, this is what we observed in all cases. In fact, this verification is now a part of the automated, "regression" tests of VTK.

Table 4. Descriptive statistics of a pseudo-random sample (size: 10^6), averaged across 200 runs, *versus* theoretical values: standard uniform distribution.

Statistic	Sample Mean	Standard Deviation	Theoretical Value
Mean	0.4999973	$5.968778 \cdot 10^{-5}$	0.5
Variance	$8.333224 \cdot 10^{-2}$	$1.292998 \cdot 10^{-5}$	$0.08\underline{3}\ldots$
Skewness	$\begin{cases} g_1 : -4.508892 \cdot 10^{-6} \\ G_1 : -4.508891 \cdot 10^{-6} \end{cases}$	$\begin{cases} g_1 : 2.985500 \cdot 10^{-4} \\ G_1 : 2.985500 \cdot 10^{-4} \end{cases}$	0
Kurtosis	$\begin{cases} g_2 : -1.200003 \\ G_2 : -1.200003 \end{cases}$	$\begin{cases} g_2 : 1.858247 \cdot 10^{-4} \\ G_2 : 1.858247 \cdot 10^{-4} \end{cases}$	-1.2

Table 5. Descriptive statistics of a pseudo-random sample (size: 10^6), averaged across 200 runs, *versus* theoretical values: standard normal distribution.

Statistic	Sample Mean	Standard Deviation	Theoretical Value
Mean	$-1.279931 \cdot 10^{-6}$	$1.706985 \cdot 10^{-4}$	0
Variance	0.9999982	$2.711815 \cdot 10^{-4}$	1
Skewness	$\begin{cases} g_1 : 1.499232 \cdot 10^{-4} \\ G_1 : 1.499232 \cdot 10^{-4} \end{cases}$	$\begin{cases} g_1 : 3.946604 \cdot 10^{-4} \\ G_1 : 3.946604 \cdot 10^{-4} \end{cases}$	0
Kurtosis	$\begin{cases} g_2 : 4.420736 \cdot 10^{-4} \\ G_2 : 4.422611 \cdot 10^{-4} \end{cases}$	$\begin{cases} g_2 : 8.874214 \cdot 10^{-4} \\ G_2 : 8.874219 \cdot 10^{-4} \end{cases}$	0

Table 6. Correlative statistics of a pseudo-random sample (size: 10^6), averaged across 100 runs, *versus* theoretical values: standard uniform distribution.

Statistic	Sample Mean	Standard Deviation	Theoretical Value
Mean X	0.4999953	$5.779502 \cdot 10^{-5}$	0.5
Mean Y	0.4999987	$6.061554 \cdot 10^{-5}$	0.5
Variance X	$8.333061 \cdot 10^{-2}$	$1.242125 \cdot 10^{-5}$	$0.08\underline{3}\ldots$
Variance Y	$8.333405 \cdot 10^{-2}$	$1.267820 \cdot 10^{-5}$	$0.08\underline{3}\ldots$
Covariance	$6.778523 \cdot 10^{-7}$	$1.622636 \cdot 10^{-5}$	0

Table 7. Correlative statistics of a pseudo-random sample (size: 10^6), averaged across 100 runs, *versus* theoretical values: standard normal distribution.

Statistic	Sample Mean	Standard Deviation	Theoretical Value
Mean X	$-6.855281 \cdot 10^{-6}$	$1.693682 \cdot 10^{-4}$	0
Mean Y	$-2.714105 \cdot 10^{-7}$	$1.695545 \cdot 10^{-4}$	0
Variance X	1.000016	$2.883309 \cdot 10^{-4}$	1
Variance Y	0.9999838	$2.551778 \cdot 10^{-4}$	1
Covariance	$-1.007851 \cdot 10^{-5}$	$1.885743 \cdot 10^{-4}$	0

4 Conclusion

In this report, we have provided a summary of the existing statistical engines in VTK/Titan, and have presented the parallel versions thereof which have already been implemented. We have subsequently illustrated the ease of use of these parallel engines by the means of a simple example and C++ code snippets. Last, we have demonstrated that these parallel descriptive engines exhibit perfect parallel scale-up properties, as was expected in theory based on their design made with this very purpose in mind.

Future work will involve in particular:

1. The parallelization of those statistics engines which have not yet been parallelized;
2. The porting and utilization of the parallel statistics engines on a terascale computer, in order to perform statistical analysis on a scale never seen before;
3. The writing of a more detailed user manual, encompassing all parallel statistical engines with all available options and subtleties.

References

[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.

[tor] TORQUE Resource Manager. http://www.clusterresources.com/pages/products/torque-resource-manager.php.

[vtk] VTK Doxygen documentation. http://www.vtk.org/doc/nightly/html.

DISTRIBUTION:

2	MS 9159	Philippe P. Pébay, 8963
1	MS 9159	David Thompson, 8963
2	MS 9018	Central Technical Files, 8944
1	MS 0899	Technical Library, 9536

Sandia National Laboratories