The Value of Variance

Jesun Sahariar Firoz
jsfiroz@iu.edu

Martina Barnas
mbarnas@indiana.edu

Marcin Zalewski
zalewski@iu.edu

Andrew Lumsdaine
lums@iu.edu

Center for Research in Extreme Scale Technologies (CREST)
Indiana University, Bloomington, IN, USA

ABSTRACT

Measurements for distributed algorithms, such as performance results, are usually reported using averages, similarly to prevailing practice in other areas of computer science. We argue that including standard deviations offers additional information and that the minimal burden of providing standard deviations is outweighed by the benefits. We propose a new way of reporting run time speedup that incorporates standard deviation and demonstrate its usefulness in terms of two distributed graph algorithms.

Keywords

Statistical measurement; Speedup; Distributed algorithms; Performance metric; Standard deviation

1. INTRODUCTION

The increasing complexity of the software/hardware stack of modern machines, especially supercomputers, has made it hard to analyze performance of different algorithms for large-scale applications and reproduce relevant empirical results across different platforms. The variabilities incurred during reproduction of experimental results are generally acknowledged by the community and are attributed to the platform-dependent many-dimensional parameters. Considering all factors involved in experimental design, performance analysis has become an experimental science, made even more challenging due to the presence of massive irregularity and data dependency in important emerging problem areas. Hence, the baseline experimental analyses seldom incorporate uncertainty measures while reporting performance and thus lack in giving insight about an algorithm's performance.

Recently, Hoefler and Belli [14] compiled several guidelines to report results and advocated for the term *interpretability* in place of reproducibility. The authors call an experiment *interpretable* "if it provides enough information to allow scientists to understand the experiment, draw own conclusions, assess their certainty, and possibly generalize results". Many papers in which experimental algorithms are proposed, lack the characteristic of being interpretable. In the papers reporting parallel performance, the general trend is to mention the performance of an algorithm in terms of speedup. Speedup is a metric for relative performance, defined as ratio of performance results (execution times). Typically, it is stated as a single number. However, due to different execution environments, system noise, network congestion etc., execution times are hardly deterministic. Hence, for example, a statement "algorithm A runs X times faster than algorithm B" is not always true. The speedup X should be bounded by an upper and lower limit based on the uncertainties from a set of sample runs. Some questions, for example: whether the execution times are normally distributed or not, how many runs are sufficient to predict the behavior of an algorithm, etc. remain at large and can, at least in principle, be tackled by some standard statistical methodologies (for example: Analysis of Variance (ANOVA) for the last case). However, compounding the issue is the expense of running experiments on supercomputers.

Another aspect of interpreting performance results comes from their role in aiding design of software systems. For example, our primary interest centers on designing runtime system for exascale. We use, e.g., performance of mini apps to guide the development of the runtime system. In this context, the goal is not to answer fundamental questions regarding validity of statistical approach; rather, it is to infer insight about the system such identifying bottlenecks. It is not desirable to run many experiments both due to cost and due to time it takes; on the contrary, we want to maximize insight while minimizing number of runs even if it is at the expense of rigor.

A good example to investigate the implications of proper performance reporting are irregular applications such as distributed graph algorithms. We have shown previously that for performance engineering of distributed graph algorithms, concentrating on the algorithm part of the application is not sufficient [10]. We called for more transparency in reporting results in literature, and advocated for documenting lower level runtime features that are usually overlooked [9]. This would allow us collectively construct a deeper understanding of these complex issues in order to uncover practical implications for performance engineering.

In order to be able to learn as a community, lessons learned across the field need to be generalizable and transferable. This condition is a given in hard sciences such as physics where what we know and how well we know it is inseparable. However, in computer science reporting results of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPE'16, March 12-18, 2016, Delft, Netherlands

© 2016 ACM. ISBN 978-1-4503-4080-9/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2851553.2851573

experiments falls short of this expectation (for review, see [14]). As previously mentioned, what is typically reported are mean values without uncertainty of measurements. Thus, rigorously speaking, comparing results across different experiments or runs is not meaningful. The lack of rigor at this level percolates to undermine the ability to draw conclusions about more complex phenomena.

In this study we examine how inclusion of standard deviations of measurements illuminates performance results. For simplicity, we consider different *runs* on the same supercomputer. Each run consists of solving a set of *problem instances*. We acknowledge that methodologically, experiments in computer science are not as clear-cut as in physics, and lie somewhere in between physics and social science [17] inquiry.

Standard deviation is arguably the simplest of statistical measures, and is adequate in much of physics measurements. It is easy to implement, and we posit that it would improve the state of experiment analysis for practitioners. For these reasons, adding standard deviation as a measure of uncertainty is the focus of this present work. Moreover, standard deviation enable to expand definition of speedup, a commonly reported quantity, in a more meaningful way that allows for comparison across different runs and experiments. We refer to the expansion as *adjusted speedup equation*.

For the purpose of our study, we have chosen the prototypical irregular problem of graph traversal, in particular the single source shortest path (SSSP) problem. Graph traversal is a basic building block of other graph algorithms used in social network analytics, transportation optimization, artificial intelligence, power grids, and, in general, any problem where data consists of entities that connect and interact in irregular ways. Given a source and a destination in a graph, the SSSP problem asks to find the shortest route between the source and the destination. Specifically, we have chosen two different algorithms for finding single-source shortest paths, Δ-stepping algorithm [19] and K-level Asynchronous algorithm (KLA) [13], implementeded in two asynchronous many-task runtime systems called High Performance ParalleX 5 (HPX-5)[2] and AM++ [24]. The study was done within the context of development work of HPX-5.

The paper is organized as follows. Sec. 2 gives a summary of the method for evaluating and expressing uncertainty when multiple input quantities are involved. We then propose an adjusted speedup equation based on this discussion. Next, to make the paper self-contained, in Sec. 3, we give a brief overview of Δ-stepping and KLA based SSSP algorithms. In Sec. 4, we discuss the High Performance ParalleX 5 (HPX-5) and AM++ runtime we used to implement our SSSP algorithms. In Sec. 5, we show how including standard deviation in presenting and comparing performance results conveys valuable information that would otherwise be impossible to infer. In Sec. 6, we give a synopsis of related work. We provide our concluding remarks in Sec. 7.

2. STATISTICS OF UNCERTAINTY MEASUREMENTS

Calculation of speedup involves independent measurement of some metric (e.g., execution time, TEPS, etc.). Each of these independent performance metrics has uncertainty associated with it. When computing speedup, each of these uncertainties should be taken into account. In this section, we first recap the NIST [3] guidelines on uncertainty of measurement results, which dictate how to calculate output uncertainty when two or more independent inputs and their associated uncertainties are involved. Next, we use the NIST guidelines to propose an adjusted speedup equation, taking into consideration the associated uncertainties of performance metrics.

2.1 Background

National Institute of Standards and Technology (NIST) [3] provides guidance regarding uncertainty in physical experiments. Consider a quantity Y being measured, called the *measurand*, that can be expressed as a function of N other quantities $X_1, X_2, \ldots, X_N$

$$Y = f(X_1, X_2, \ldots, X_N). \tag{1}$$

These quantities $X_1, X_2, \ldots, X_N$ can include other factors involved in a physical experiment, such as different observers, instruments, samples, laboratories and times at which observations are made. Consequently, the function f should contain all quantities that can contribute a significant uncertainty to the measurement result.

The *estimate of the measurand* or output quantity Y denoted by y, is derived from Eq. (1) using input estimates $x_1, x_2, \ldots, x_N$ for the values of N input quantities $X_1, X_2, \ldots, X_N$. Thus, the estimate of measurand is

$$y = f(x_1, x_2, \ldots, x_N). \tag{2}$$

The *uncertainty* of the measurement result y emerges from the component uncertainties $u(x_i)$, or u_i for brevity, of the input estimates x_i. Components of the uncertainty can be divided into two categories according to the method used to evaluate them: *1)Type A Evaluation:* Method of evaluation of uncertainty is based on the statistical analysis of the series of observations; and *2)Type B Evaluation:* Method of evaluation of uncertainty is based on means other than the statistical analysis of the series of observations. (These were formerly known as random and systematic uncertainty, respectively. NIST cautions against the old terminology since it can be misleading.) However evaluated, each component of uncertainty, u_i is equal to the positive square root of the estimated variance.

A useful quantity is the *relative standard uncertainty* defined as

$$u_r(x_i) = \frac{u(x_i)}{|x_i|}. \tag{3}$$

where x_i is assumed nonzero.

In this paper, we are interested in Type A evaluation. Let us consider the input quantity X_i. If we get the values for this input quantity by n independent observations $X_{i,k}$ under the same condition of measurement, then the input estimate x_i can be represented as the *sample mean*

$$x_i = \overline{X}_i = \frac{1}{n}\sum_{k=1}^{n} X_{i,k}. \tag{4}$$

An uncertainty component obtained by a Type A evaluation is represented by statistically estimated standard deviation σ_i of the sample mean, equal to the positive square root of the statistically estimated variance ${\sigma_i}^2$ and the associated number of degrees of freedom v_i. For such a component, the standard uncertainty is $u_i = \sigma_i$.

$$u(x_i) = u_i = \sigma_i = \left(\frac{1}{n(n-1)} \sum_{k=1}^{n} (X_{i,k} - \overline{X}_i)^2 \right)^{\frac{1}{2}}. \quad (5)$$

If multiple quantities $X_1, X_2, \ldots, X_N$ are involved in the calculation of estimate y, the *combined standard uncertainty of measurement results*, denoted by $\sigma(y)$, and representing the estimated standard deviation of the result, is the positive square root of the estimated variance $\sigma^2(y)$ obtained from,

$$\sigma^2(y) = \sum_{i=1}^{N} \left(\frac{\partial f}{\partial x_i} \right)^2 \sigma^2(x_i) + 2 \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{\partial f}{\partial x_i} \frac{\partial f}{\partial x_j} \sigma(x_i, x_j). \quad (6)$$

Equation (6) is based on a first-order Taylor series approximation of the measurement equation Eq. (1) and is referred to as *the law of propagation of uncertainty*. The partial derivatives of f w.r.t. the X_i are called *sensitivity coefficients*, and are equal to the partial derivatives of f w.r.t the X_i evaluated at $X_i = x_i$. $\sigma(x_i)$ is the standard uncertainty associated with the input estimate x_i; and $\sigma(x_i, x_j)$ is the estimated covariance associated with x_i and x_j. If the input estimates x_i of the input quantities X_i can be assumed to be uncorrelated, then the second term vanishes.

As mentioned in [4], if the probability distribution characterized by the measurement result y and its combined standard uncertainty $\sigma(y)$ is approximately normal (Guassian), and $\sigma(y)$ is a reliable estimate of the standard deviation of y, then the interval $y - \sigma(y)$ to $y + \sigma(y)$ is expected to encompass approximately 68% of the distribution of values that could reasonably be attributed to the value of the quantity Y of which y is an estimate. This implies that it is believed with an approximate level of confidence of 68% that Y is greater than or equal to $y - \sigma(y)$ and less than or equal to $y + \sigma(y)$ which is commmonly written as $Y = y \pm \sigma(y)$.

2.2 Adjusted Speedup Equation

Let us assume that the average (mean) execution time for Algorithm A and Algorithm B is $\bar{t}_A$ and $\bar{t}_B$, respectively. Let us denote the standard deviations σ_A and σ_B, and assume that execution times for Algorithm A and Algorithm B are independent of each other. Typically, speedup S of Algorithm B over Algorithm A is calculated as a ratio of the two execution times:

$$S = \frac{\bar{t}_A}{\bar{t}_B}. \quad (7)$$

The uncertainly component, associated with $\bar{t}_A$ and $\bar{t}_B$ contributes to the calculation of combined standard uncertainty of the measurement result S. As $\bar{t}_A$ and $\bar{t}_B$ measures are uncorrelated, according to Eq. (6), the combined standard uncertainty σ of the measurement result S is

$$\sigma^2 = \frac{1}{\bar{t}_B^2} {\sigma_A}^2 + \frac{\bar{t}_A^2}{\bar{t}_B^4} {\sigma_B}^2. \quad (8)$$

Note that σ, just as the speedup S, is dimensionless while the standard deviations associated with execution times are dimensionful. We propose that speedup is reported with its uncertainty,

$$S_{adj} = S \ \pm \ \sigma, \quad (9)$$

which, combining Eq. (8) and Eq. (9), yields adjusted speedup equation in terms of observables $\bar{t}_A, \bar{t}_B, \sigma_A, \sigma_B$:

$$S_{adj} = \frac{\bar{t}_A}{\bar{t}_B} \ \pm \ \frac{1}{\bar{t}_B} \sqrt{{\sigma_A}^2 + \frac{\bar{t}_A^2}{\bar{t}_B^2} {\sigma_B}^2}. \quad (10)$$

3. OVERVIEW OF SSSP ALGORITHMS

Large scale graph processing requires distribution of the graph across multiple nodes and employing a distributed algorithm. Performance engineering for distributed graph algorithms is inherently difficult due to the irregular memory access patterns [18]. Graph algorithm performance depends not only on the algorithm logic but also on factors such as the runtime system, synchronization, lock-free data structure, processing order, etc. In this paper, we demonstrate the usefulness of including uncertainty measurement with the example of single source shortest path problem solved by two different distributed algorithms: Δ-stepping and KLA.

Conceptually, all data driven graph algorithms can be described as an *Abstract Graph Machine* (AGM) [16]. The primitive unit of processing in AGM is a *work item*, which is a tuple that has a vertex or an edge together with several *graph properties*. For example, a SSSP work item will have a vertex and distance. AGM comprises of a work item processor and a work item ordering component. The processor executes basic algorithm logic (e.g., the "relax" operation in SSSP). The ordering component partitions work items into ordered equivalence classes and feeds the smallest partition back into the processor.

Definition 1. *An **Abstract Graph Machine**(AGM) is a 5-tuple (G, WorkItems, PF, $<_{wis}$, S), where*

1. *G = (V, E) is the **input graph**,*
2. *WorkItems $\subseteq (V \times P_0 \times P_1 \cdots \times P_n)$ where each P_i represents a graph property,*
3. *$PF : WorkItems \longrightarrow \mathcal{P}(WorkItems)$ is the **processing function**,*
4. *$<_{wis}$ - **Strict weak ordering relation** defined on WorkItems*
5. *S ($\subseteq$ WorkItems) - **Initial** WorkItems **set**.*

The AGM processing is driven by the *WorkItems*. The initial *WorkItems* set, $WIS_0 = S$. Let $WIS_{current}$ be the currently processing *WorkItems* set, then we calculate the next active *WorkItems* set as follows;

Input to the processing function :–
WIS_{in} = Ordering($WIS_{current}$)

Let, $SPF \ (\subseteq WorkItems) = \bigcup_{w_j \in WIS_{in}} PF(w_j)$

Now we calculate *next WorkItems* set as follows;
$WIS_{next} = SPF \cup (WIS_{current} - WIS_{in})$

The AGM terminates when $WIS_{next} = \{\}$.

Figure 1 depicts how ordering and processing functions interact with each other. Next we describe KLA and Δ-stepping algorithm in terms of AGM. Interestingly, most of the algorithms share a common processing function. In general the SSSP processing function (*SSSP_PF*) can be defined as follows:

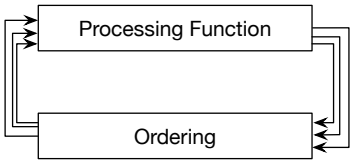

Figure 1: An Overview of *Abstract Graph Machine*

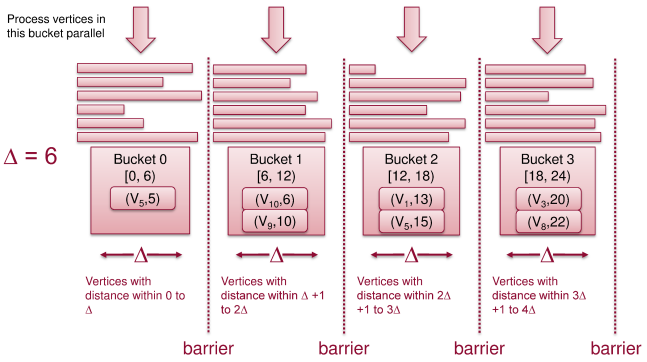

Figure 2: How Δ-stepping algorithm works

Definition 2. *SSSP_PF : SSSP WorkItems* $\longrightarrow$ *Partition* $\mathcal{P}$ *(SSSP WorkItems)*

$$SSSP\_PF(w) = \begin{cases} \{w_k | w_k[0] \in neighbors(w[0]) \text{ and} \\ \quad w_k[1] = w[1] + weight(w[0], w_k[0])\} \\ \quad \text{if } w[1] < distance(w[0]) \\ \\ \{\} \quad else \end{cases}$$

3.1 Δ-Stepping Algorithm

Δ-Stepping [19] arrange tasks into distance ranges (buckets) of size $\Delta(\in \mathbb{N})$ and execute buckets in order. Within a bucket, tasks are not ordered, and can be executed in any order (Fig. 2). Processing a bucket may produce extra work for the same bucket or for the successive buckets.

Definition 3. $<_\Delta$ *is a binary relation defined on SSSP WorkItems as follows; Let* $w_1, w_2 \in$ *SSSP Workset, then* $w_1 <_\Delta w_2$ *iff* $\lfloor w_1[1]/\Delta \rfloor < \lfloor w_2[1]/\Delta \rfloor$

Δ-Stepping algorithm partition SSSP *WorkItems* set based on relation $<_\Delta$ ($<_\Delta$ is a strict weak ordering relation.)

Proposition 1. Δ*-Stepping Algorithm is an instance of AGM where*

1. *G = (V, E) is the input graph*
2. *WorkItems = SSSP WorkItems*
3. *PF = SSSP_PF*
4. *Strict weak ordering relation* $<_{wis} = <_\Delta$
5. $S = \{<v_s, 0>\}$ *where* $v_s \in V$ *and* v_s *is the source vertex.*

3.2 KLA SSSP Algorithm

The KLA SSSP algorithm [13] requires both Distance property and Level property in the *WorkItems* set. The Level property is needed to track the number of levels, k, asynchronously processed (Fig. 3). Therefore, we define *SSSP KLA WorkItems* $\subseteq (V \times Distance \times Level)$.

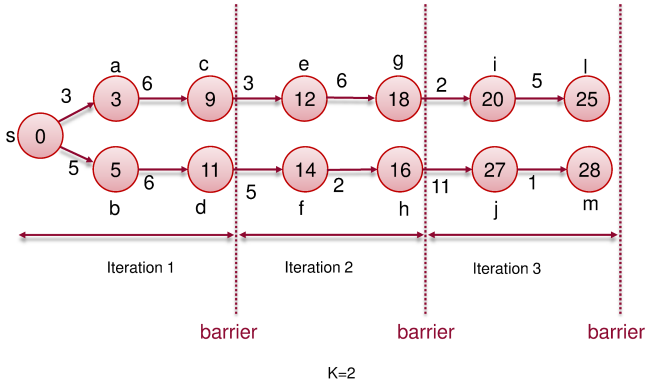

Figure 3: How algorithms progress in KLA paradigm

The processing function for KLA SSSP is defined in Definition 4.

Definition 4. *KLA_PF : SSSP KLA WorkItems* $\longrightarrow$ $\mathcal{P}$ *(SSSP KLA WorkItems)*

$$KLA\_PF(w) = \begin{cases} \{w_k | w_k[0] \in neighbors(w[0]) \text{ and} \\ \quad w_k[1] = w[1] + weight(w[0], w_k[0]) \\ \quad \text{and } w_k[2] = w[2] + 1\} \\ \quad \text{if } w[0] < distance(w[0]) \\ \\ \{\} \quad else \end{cases}$$

The strict weak ordering relation for SSSP KLA is defined in Definition 5:

Definition 5. $<_{sssp\_kla}$ *is a binary relation defined on SSSP KLA WorkItems as follows:*
Let $w_1, w_2 \in$ *SSSP KLA WorkItems, then;*
$w_1 <_{sssp\_kla} w_2$ *iff* $\lfloor w_1[2]/k \rfloor < \lfloor w_2[2]/k \rfloor$

Note, the definition of $<_{sssp\_kla}$ is quite close to the definition of $<_\Delta$.

KLA SSSP algorithm partition SSSP *WorkItems* set based on relation $<_{sssp\_kla}$ ($<_{sssp\_kla}$ is a strict weak ordering relation.)

Proposition 2. *KLA SSSP Algorithm is an instance of AGM where;*

1. *G = (V, E) is the input graph*
2. *WorkItems = SSSP WorkItems*
3. *PF = KLA_PF*
4. *Strict weak ordering relation* $<_{wis} = <_{sssp\_kla}$
5. $S = \{<v_s, 0>\}$ *where* $v_s \in V$ *and* v_s *is the source vertex.*

4. OVERVIEW OF THE RUNTIMES

We implemented two SSSP algorithms in two different runtime systems, AM++ [24] and HPX-5 [2]. HPX-5 is a high performance runtime library whose implementation is based on the the ParalleX execution model [6] targetted for exascale computing. AM++ is our legacy system centered around active messaging of the Active Pebbles [25] model.

HPX-5 comprises of a set of main components: localities, global memory, Lightweight threads and actions, Lightweight Control Objects (LCO) and parcels. These components along with the scheduler and network transport drive program execution in HPX-5. HPX-5 is intended to enable dynamic adaptive resource management and task scheduling. It creates a global name and address space (Partitioned Global Address Space (PGAS) and Active Global Address Space (AGAS)) structured through a hierarchy of processes, each of which serve as execution contexts and may span multiple nodes. It is event-driven, enabling the migration of continuations and the movement of work to data, when appropriate, based on sophisticated local control synchronization objects (e.g., futures, dataflow). HPX-5 is an evolving runtime system being employed to quantify effects of latency, overhead, contention, and parallelism. These performance parameters determine a tradeoff space within which dynamic control is performed for best performance. It is an area of active research driven by complex applications and advances in HPC architecture.

AM++ supports fine-grained parallelism of active messages with communication optimization techniques such as object-based addressing, active routing, message coalescing, message reduction, and termination detection. While less feature-rich than HPX-5, active messages share the fine-grained parallelism approach with HPX-5. In addition, AM++ is a relatively well-optimized implementation to balance the competing needs of quick delivery of work vs. minimal communication overhead.

While AM++ and HPX-5 share some features and goals, there are important differences between them. AM++ is designed for bulk processing of distributed messages, while HPX-5 is a complete system providing inter and intra-node parallelism. HPX-5 provides global address space while AM++ provides only a lightweight object-based addressing layer. In HPX-5 work is divided into first-class tasks with stacks, while AM++ only executes message handler functions on the incoming message data. These features result in significant differences in scheduling.

5. INSIGHTS FROM THE STANDARD DEVIATION

In this section we illustrate how including standard deviations provides additional insight that is not evident from the averaged quantities alone. We present weak scaling performance measurements obtained during development of HPX-5 runtime, to indicate how this way of looking at data can aid in development process. For comparison, we run the same experiments under a different, less feature rich, but comparatively well optimized runtime, AM++. We implemented KLA and Δ-stepping algorithms in these runtimes. Both of these algorithms combine asynchronous processing with a global synchronization barrier. The degree of asynchrony is regulated by a parameter (k in KLA and Δ in Δ-stepping). For the results presented here for HPX-5, we used $k = 2$ and $\Delta = 1$ which minimize the asynchronous work.[1] Even with these choices, depending on the input data, KLA can be expected to perform more asynchronous work than Δ-stepping.

[1] The reader is reminded that we wanted to document how the proposed methodology aids in development. We are not attempting to achieve optimal performance or to test the algorithms.

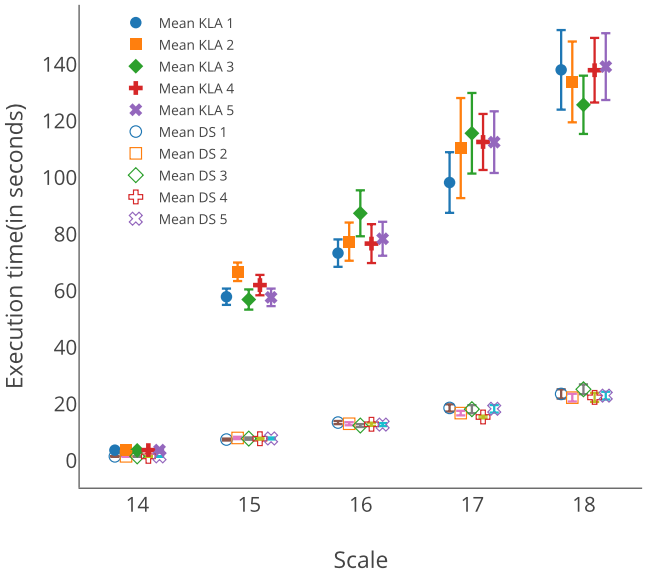

Figure 4: Weak scaling results for KLA (top) and Δ-stepping (bottom) on HPX-5 with standard deviations for 5 runs. The central point for each run is the average time for the run; the error bars show the standard deviation. Maximum edge-weight for the input graph is 255.

5.1 Experimental Setup

We conducted our experiments on Indiana University's BigRed 2 Cray XE6/XK7 supercomputer [1]. The compute nodes are connected with Gemini interconnect. Each compute node contains two AMD Opteron 16-core Abu Dhabi x86_64 CPUs and 64 GB of RAM. We used 16 threads per compute node for both AM++ and HPX-5. We compiled our program with gcc version 4.9.3 compiler with optimization flag $O3$ enabled.

For input graph generation process, we used Graph500 specification [5] with RMAT generator. Edge weights are assigned based on a pseudo-random number generator. For both AM++ and HPX-5 weak scaling results in terms of execution time and speedup, we used 1, 2, 4, 8 and 16 computing nodes for scale 14, 15, 16, 17 and 18, respectively. By scale x, we mean there are 2^x vertices in the generated graph.

Each data point at a given scale shows one run encompassing 8 different problem instances. Problem instances correspond to different starting points (sources). Both algorithms exhibit some sensitivity to the starting point. We find that the sensitivity appears to be consistent, and thus, an inconsistency suggests occurrence of a systemic effect. As evident from the figure, we measured 5 runs for each scale. We calculated the speedup by taking the ratio of execution time for KLA to Δ-stepping algorithm.

5.2 Reporting Speedup Uncertainty

5.2.1 *On HPX-5 Runtime*

Figs. 4 and 6 show weak scaling results on HPX-5 runtime for Δ-stepping and KLA SSSP algorithms with two different input graphs with maximum edge weight of 255 and 100, respectively. We chose two different graph inputs to verify whether the anticipated speedup plots (discussed later) have similar trends. The error bars shown correspond

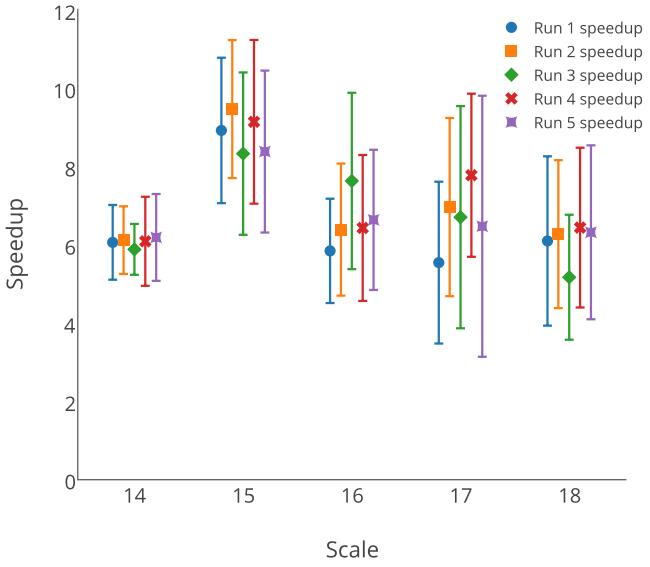

Figure 5: Adjusted speedup (ratio of KLA and Δ-stepping execution time) on HPX-5 with calculated standard deviations. Maximum edge-weight for the input graph is 255.

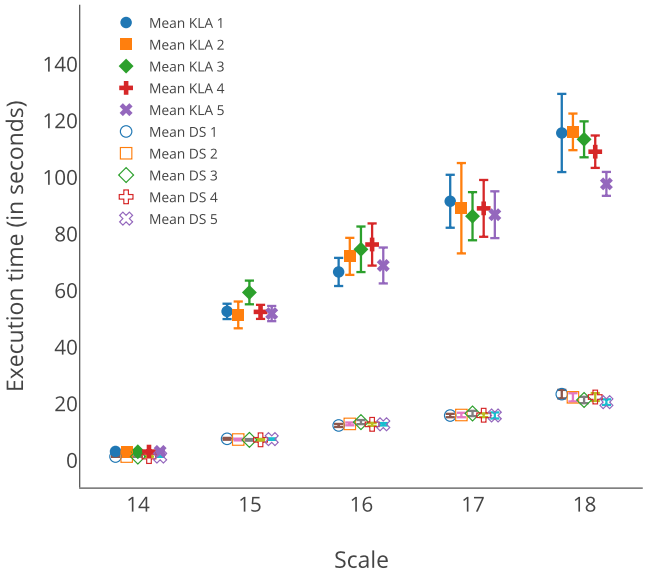

Figure 6: Weak scaling results for KLA (top) and Δ-stepping (bottom) on HPX-5 with standard deviations for 5 runs. The central point for each run is the average time for the run; the error bars show the standard deviation. Maximum edge-weight for the input graph is 100.

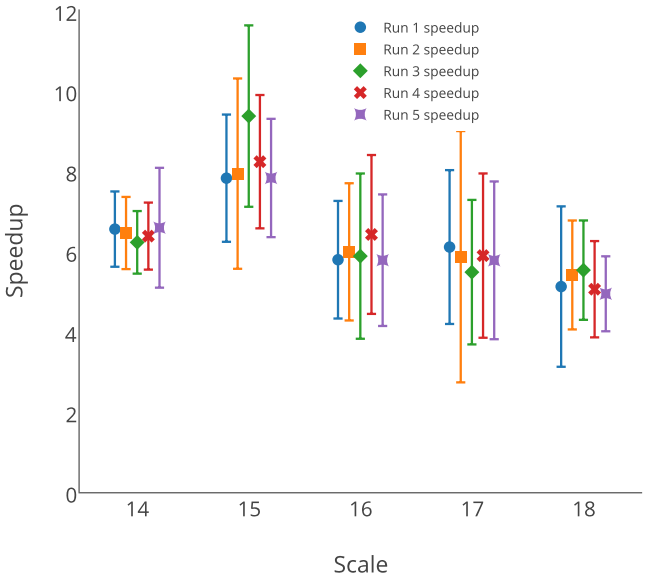

Figure 7: Adjusted speedup (ratio of KLA and Δ-stepping execution time) on HPX-5 with calculated standard deviations. Maximum edge-weight for the input graph is 100.

to the standard deviations of the average execution times. Δ-stepping algorithm is faster than KLA. The figures show that in comparison to Δ-stepping, the standard deviations from the average execution time are larger for KLA algorithm.

Although this observation is intuitive and is incorporated in practice to-date, the next related question to ask ourselves is what speedup can we anticipate when comparing the execution times for both algorithms on different runtimes or even across different runs on the same runtime? Is there a way to quantify the observable uncertainty in speedup by incorporating simple measurement like combining uncertainty measures for average execution times for both algorithms? Is saying that an algorithm runs "five times" faster good enough? We address these questions next in connection to Sec. 2.2, where we presented an equation to calculate adjusted speedup. Figures 5 and 7 present the speedup plots with standard deviations, calculated from Eq. (10). In all 5 runs, we use the same input and problem instances for a particular scale. As can be seen from the figure, the speedup can be expected to vary significantly within the approximate range of 3 to 11. But interestingly all the averages across different runs lie within the range indicated by the standard deviations for all 5 runs. For example, in Fig. 5, we can see that average speedup for run 1 centers around 6 for most cases except for scale 15, due to distributed execution on 2 nodes. But as we increase the number of nodes, the speedup again settles around 6 due to increasing network latency. Additionally, we can see that speedups across different runs cluster together pretty well. This is helpful in conjunction with the calculated deviation for speedup (Eq. (8)). Assuming that the combined uncertainty for speedup is normally distributed, Figs. 5 and 7 shows us the expected range of speedups with approximate level of confidence of 68%. Moreover Figs. 5 and 7 both show similar speedup behaviour for two different graph inputs.

5.2.2 *On AM++ Runtime*

Figure 8 and Fig. 9 show weak scaling results on AM++.

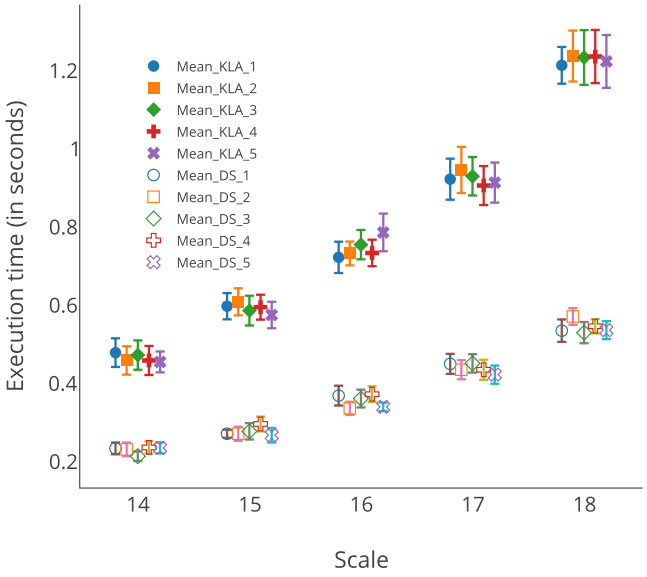

Figure 8: Weak scaling results for KLA and Δ-stepping on AM++ with standard deviations for 5 runs. The central point for each run is the average execution time for the run; error bars show the standard deviation. Maximum edge-weight for the input graph is 100.

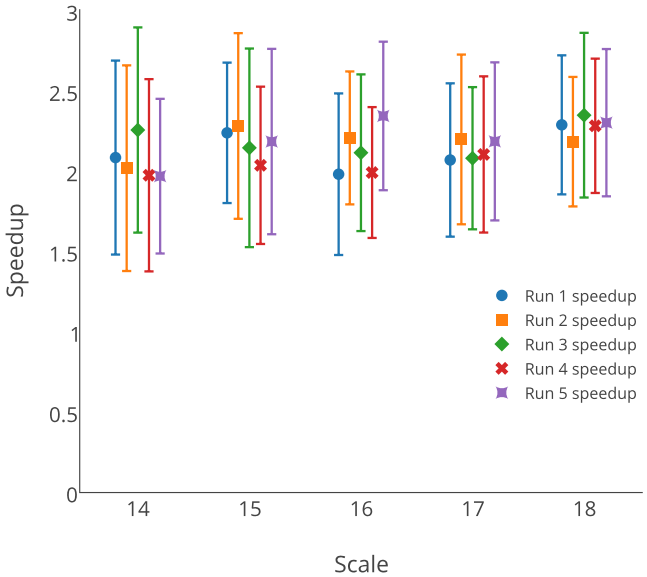

Figure 9: Adjusted speedup (ratio of KLA and Δ-stepping execution time) on AM++ with calculated standard deviations. Maximum edge-weight for the input graph is 100.

As both Δ-stepping and KLA algorithms execute faster on AM++, the standard deviations of the average of the execution time is small in AM++ compared to HPX-5. Since the execution times for both of the algorithms are significantly less, the speedup variability is also small in AM++, as can be seen from Fig. 9. This is particularly true as we increase the scale, for example scale 17 and 18.

5.2.3 *Comparing Speedup Across Runtimes*

More importantly, including speedup variability can give us additional information about the performance of Δ-stepping and KLA across two different runtimes AM++ and HPX-5. From Fig. 5 we can see that the speedup on HPX-5 centers around 6 and varies approximately within the range of ±3. From Fig. 9 we see that the speedup on AM++ is around 2 with an approximate range of ±1. So the variability in speedup is roughly 50% in both cases.

5.2.4 *Usefulness of Relative Standard Uncertainty*

We also calculate *relative standard uncertainty* (RSU) from Eq. (3). We plot the calculated RSUs in Figs. 10 to 12. These plots, based on standard deviation calculation are also useful.

For example, in Fig. 11, for scale 17 with 8 nodes, the RSU for KLA algorithm for run 2 is about 0.5. We investigated why this is the case and found out that problem instance 7 took 50% more time compared to the average execution time. Then, we looked into the execution time for problem instance 7 from other runs (for example Table 1 compares KLA algorithm execution time from run 2 and run 5). We saw that, with KLA, problem instance 7 consistently took longer time to finish across different runs. But it took the maximum time in run 2. This additional insight lend problem instance 7 for further investigation. We would have lost this valuable information if we only considered average execution time. Computing standard deviation and taking it into consideration as a measure for uncertainty empowers us with supplementary information.

We can also see that all the datapoints with maximum RSUs belong to KLA algorithm executions. This is an indication that KLA implementation in HPX-5 exercises certain runtime scheduling and network communication patterns which stress the runtime. It also tells us that KLA algorithm is more sensitive to the starting point (source vertex) of a SSSP problem. Based on this observation, we took a second look at the execution time of each individual SSSP problem instance for both Δ-stepping and KLA algorithms. We found out that Δ-stepping algorithm solves each problem instance within an average execution time of 15 seconds, having small variability in execution time for each problem instance. On the other hand, KLA execution time varies hugely among different problem instances. It is anticipated that, based on the problem instance (source) and graph input, different problem instance will take unequal time. But the execution time for Δ-stepping algorithm suggests that there is a better way to schedule tasks within runtime. This insight can be useful to optimize KLA algrithm's execution time by gathering statistics about number of exchanged network messages, scheduling polcies like number of work stealing, thread yielding, queue size of workitems etc. We leverage instrumentation infrastructure in HPX-5 for this purpose.

We have another interesting observation with reference to Fig. 12. Again, on AM++ for scale 17 with 8 nodes, we

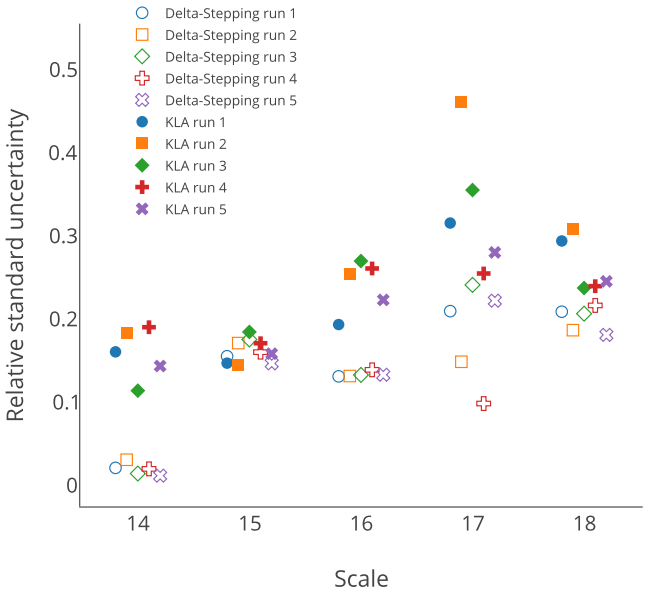

Figure 10: Relative standard uncertainty of execution time in HPX for Δ-stepping and KLA algorithms. Maximum edge-weight for the input graph is 255.

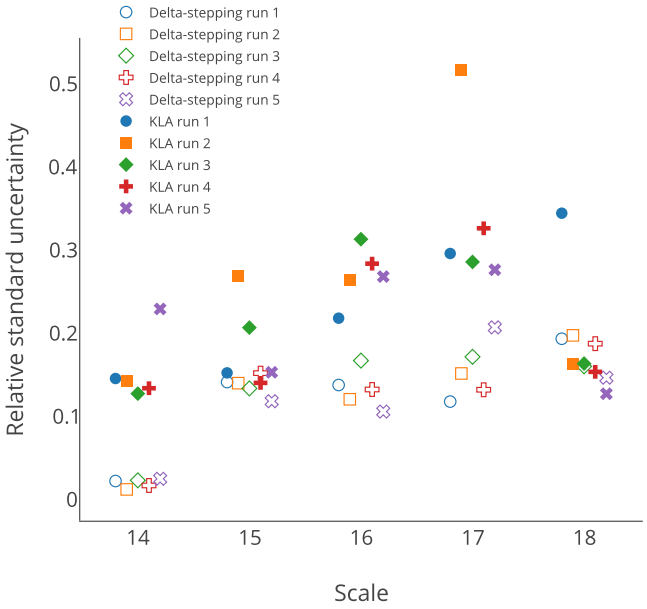

Figure 11: Relative standard uncertainty of execution time in HPX for Δ-stepping and KLA algorithms. Maximum edge-weight for the input graph is 100.

see that the RSUs are quite close for both algorithms. This also calls for further investigation.

6. RELATED WORK

Several researchers pointed out the shortcomings of presented results in computer science literature. Mytkowicz et al. [20] showed that seemingly innocuous experimental setup details, such as the UNIX environment size or the benchmark link order, can introduce a significant measurement bias in a system evaluation. Harji et al. [12] discussed about bugs and performance regressions that result as the

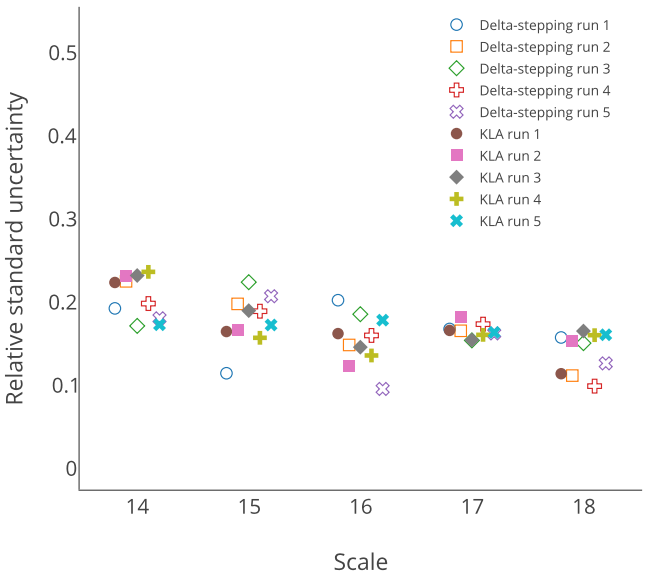

Figure 12: Relative standard uncertainty of execution time in AM++ for Δ-stepping and KLA algorithms. Maximum edge-weight for the input graph is 100.

Linux kernel evolves. For a comprehensive summary of related research see [8, 14].

A lot of attempts and proposals have been made to make computer and computational science experiments reproducible. Guerrera et al. [11] partitioned the space of computational experiments into *problem*, *method*, and *system*. Based on this partitioning, they bring forth a taxonomy for stencil benchmark results and categorized them as *replicable*, *recomputable*, and *reproducible*. Hunold and Träff [15] also urged for *reproducible* parallel computing research. Several researchers [21–23] advocated for including statistical analyses in computer science experiments. de Oliveira et al. [7] successfully demonstrated the use of quantile regression instead of ANOVA for non-normally distributed data to conduct performance evaluation. Very recently, Hoefler and Belli [14], coined the term *interpretability* and recommended a set of guidelines for scientific benchmarking based on statistical analyses. Their recommendation is based on the fact that algorithms designed for supercomputers heavily rely on particular architectures and execution environments, thus making reproducibility harder.

7. CONCLUSION

Performance engineering hinge on understanding results of conducted experiments. The key is to identify the bottlenecks and then put a concerted effort in removing these bottlenecks and optimize runtime parameters. For distributed algorithms, where complex machine architecture, network and system environment are integral part of execution, unpredictable behavior can always happen in any part of the system stack. We should account for uncertainty involved during the experiments. In this paper, we proposed a new equation to calculate speedup and showed how inclusion of standard deviation provide some insights about uncertainties associated with performance of algorithms. Under additional assumptions about the underlying distribution, the formula we introduced can become powerful tool to compare perfor-

Table 1: Execution time for KLA for different problem instances

Run	Problem Instances							
	1	2	3	4	5	6	7	8
2	67.1721137	69.1144691	58.2407584	95.0383578	81.5043034	66.1346017	196.359002	71.0783083
5	68.2187308	64.6999926	73.1340169	66.5868368	111.286377	78.3502649	127.3131639	96.6814628

mance across different set of parameters such as runtimes, algorithms etc. Any further interpretation of combined uncertainty is subject to future investigation.

8. ACKNOWLEDGMENTS

This work is supported by the NSF under grant 1111888 and grant no. 1319520 and by in part by Lilly Endowment, Inc.

References

[1] Big Red II at Indiana University. https://kb.iu.edu/d/bcqt#overview.

[2] HPX website. http://hpx.crest.iu.edu/.

[3] The NIST reference on constants, units, and uncertainty. http://physics.nist.gov/cuu/Uncertainty/basic.html, Sept. 2015.

[4] Combining uncertainty components. http://physics.nist.gov/cgi-bin/cuu/Info/Uncertainty/combination.html, Sept. 2015.

[5] The Graph500 List. http://www.graph500.org/, June 2015.

[6] ParalleX Execution Model. https://www.crest.iu.edu/projects/xpress/_media/public/parallex_v3-1_03182013.doc, June 2015.

[7] A. B. de Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney. Why you should care about quantile regression. *SIGARCH Comput. Archit. News*, 41(1):207–218, Mar. 2013. ISSN 0163-5964.

[8] A. B. de Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister. Datamill: Rigorous performance evaluation made easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 137–148, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1636-1.

[9] J. S. Firoz, T. A. Kanewala, M. Zalewski, M. Barnas, and A. Lumsdaine. The anatomy of large-scale distributed graph algorithms. *CoRR*, abs/1507.06702, 2015. URL http://arxiv.org/abs/1507.06702.

[10] J. S. Firoz, M. Zalewski, T. A. Kanewala, M. Barnas, and A. Lumsdaine. Importance of runtime considerations in performance engineering of large-scale distributed graph algorithms. In *Euro-Par 2015: Parallel Processing Workshops*, pages 553–564. Springer International Publishing, 2015.

[11] D. Guerrera, H. Burkhart, and A. Maffia. Reproducible experiments in parallel computing: Concepts and stencil compiler benchmark study. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I*, pages 464–474, 2014.

[12] A. S. Harji, P. A. Buhr, and T. Brecht. Our troubles with linux and why you should care. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 2:1–2:5, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1179-3.

[13] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. KLA: A New Algorithmic Paradigm for Parallel Graph Computations. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 27–38. ACM, 2014.

[14] T. Hoefler and R. Belli. Scientific Benchmarking of Parallel Computing Systems. Nov. 2015. accepted at IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC15).

[15] S. Hunold and J. L. Träff. On the state and importance of reproducible experimental research in parallel computing. *CoRR*, abs/1308.3648, 2013. URL http://arxiv.org/abs/1308.3648.

[16] T. A. Kanewala, M. Zalewski, M. Barnas, J. S. Firoz, and A. Lumsdaine. Abstract graph algorithms with spatial-temporal execution. In preparation.

[17] J. Levin, J. A. Fox, and D. R. Forde. *Elementary statistics in social research*. Allyn & Bacon, 2010.

[18] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007 2007.

[19] U. Meyer and P. Sanders. Δ-stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms*, 49(1): 114–152, 2003.

[20] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3): 265–276, Mar. 2009. ISSN 0362-1340.

[21] L. Peterson and V. S. Pai. Experience-driven experimental systems research. *Commun. ACM*, 50(11): 38–44, Nov. 2007. ISSN 0001-0782.

[22] W. F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998. ISSN 0018-9162.

[23] J. Vitek and T. Kalibera. Repeatability, reproducibility, and rigor in systems research. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 33–38, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0714-7.

[24] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. AM++: A Generalized Active Message Framework. In *Proce. 19th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 401–410. ACM, 2010.

[25] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Active pebbles: a programming model for highly parallel fine-grained data-driven computations. In *Proc. 16th ACM symposium on Principles and practice of parallel programming*, pages 305–306. ACM, 2011.