Characterizing the Performance of Modern Architectures Through Opaque Benchmarks: Pitfalls Learned the Hard Way

Luka Stanisic, Lucas C Mello Schnorr, Augustin Degomme, Franz C Heinrich, Arnaud Legrand, Brice Videau

Characterizing the Performance of Modern Architectures Through Opaque Benchmarks: Pitfalls Learned the Hard Way

Luka Stanisic*, Lucas Mello Schnorr† Augustin Degomme‡, Franz C. Heinrich§, Arnaud Legrand§, Brice Videau§
*Inria Bordeaux Sud-Ouest, Bordeaux, France – Email: luka.stanisic@inria.fr
†Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, Brazil – Email: schnorr@inf.ufrgs.br
‡Université de Basel, Basel, Switzerland – Email: augustin.degomme@unibas.ch
§CNRS/Inria/University of Grenoble Alpes, Grenoble, France – Firstname.Lastname@imag.fr

***Abstract*—Determining key characteristics of High Performance Computing machines that would allow to predict its performance is an old and recurrent dream. This was, for example, the rationale behind the design of the LogP model that later evolved into many variants (LogGP, LogGPS, LoGPS) to cope with network technology evolution and complexity. Although network has received a lot of attention, predicting the performance of computation kernels can be very challenging as well. In particular, the tremendous increase of internal parallelism and deep memory hierarchy in modern multi-core architectures often limit applications by the memory access rate. In this context, determining the key characteristics of a machine such as the peak bandwidth of each cache level as well as how an application uses such memory hierarchy can be the key to predict or to extrapolate the performance of applications. Based on such performance models, most high-level simulation-based frameworks separately characterize a machine and an application, later convolving both signatures to predict the overall performance. We evaluate the suitability of such approaches to modern architectures and applications by trying to reproduce the work of others. When trying to build our own framework, we realized that, regardless of the quality of the underlying models or software, most of these framework rely on "opaque" benchmarks to characterize the platform. In this article, we report the many pitfalls we encountered when trying to characterize both the network and the memory performance of modern machines. We claim that opaque benchmarks that do not clearly separate experiment design, measurements, and analysis should be avoided as much as possible. Likewise an *a priori* identification of experimental factors should be done to make sure the experimental conditions are adequate.**

I. Introduction

A rigorous performance characterization is decisive to evaluate the requirements introduced by modern HPC applications and to plan platform upgrades. The ultimate goal is to predict application performance on a given platform, enabling studies of scalability, deployment optimizations, extrapolation and what-if scenarios. A common approach to perform such studies consists in convolving platform with application characteristics through a simulator [1]. However, faithful predictions requires to instantiate such simulators with measurements obtained on existing platforms.

Such measurements are generally obtained through well-known or hand-tailored benchmarks. They measure relevant platform characteristics such as network latency and bandwidth, software overhead, memory speed, floating point performance or energy use. These benchmarks generally do precise sequences of operations and rely on data captured from CPU hardware counters, from dynamic binary instrumentation, and from MPI tracing hooks, for instance.

Nevertheless, most of these benchmarks have "opaque" and sometime even arbitrary procedures, especially as hardware evolves. To achieve a minimal measurement duration and memory footprint, they perform both measurements and the corresponding statistical analysis in a black-box fashion, directly generating statistical summaries as output. No intermediary data is kept after the benchmark has finished the measurement and analysis tasks. This absence of intermediary raw measurements makes any performance investigation and model validity assessment impossible to accomplish. This ultimately leads to inaccurate models and wrong conclusions that easily go unnoticed. Furthermore, modern architectures and operating systems have become particularly complex. It is thus essential to meticulously setup and control the environment to establish meaningful benchmark conditions. We believe the experimental design, the measurements and the statistical analysis should be done through rigorous and open tools to provide sound and faithful data for model and simulation instantiation.

In this article we explain how opaque benchmarks typically work and explain how they should instead be organized to provide better information to instantiate performance models. We motivate such claim by reporting many pitfalls we encountered when trying to reproduce earlier strategies for performance characterization of recent hardware. The encountered pitfalls include the impact of code and compiler optimizations, the DVFS policy, the operating system scheduler, and peculiarities of the machine architecture. We built our analysis on simple scripts, the R programming language [2], the org-mode's literate programming approach [3] to log our activity in a reproducible research spirit so that even though some of these experiments were done five years ago, we are still able to faithfully exploit them.

Section II presents the related work on performance prediction through simulation that motivated our investigation as well as the underlying network and memory models, which allows to understand the rationale behind most benchmarks' structure. Sections III and IV respectively present performance characterization of the network layer and of the

memory hierarchy, illustrating the many pitfalls one might stumble upon when pursuing such task. Section V presents the measurement and analysis methodology we believe is the best when characterizing the performance of modern HPC architectures for model instantiation purposes. Section VI summarizes the main contributions and future work.

II. Related Work

We present the general context of HPC performance prediction through simulation and the most common underlying models. This motivates the presentation of common benchmarks for network and memory performance characterization that are detailed in Sections III and IV.

A. Performance Prediction with Simulation

A lot of effort has been endured to very accurately model HPC applications. Depending on the expectations and scope, some approaches rely on simple first-order analytic models (e.g., Amdahl's law or roofline models [4]), while others rely instead on cycle/packet-level simulations. For MPI applications, the most common is an intermediate approach [1], [5], [6], [7], [8] that convolves the application and machine signatures through a high level discrete event simulator. The PMaC framework [1], depicted in Figure 1, offers a good example to explain the method. Computational and communication capabilities are first considered separately to predict the performance of a given application `A` on a machine `M`. An MPI application can be seen as an arrangement of *sequential computation blocks* interleaved with MPI calls. The processor usage (e.g., number of instructions and of floating point operations, or the hit rate in each level of the memory hierarchy) of each block may be obtained through an instrumented execution, a detailed cycle-level simulation (the MetaSim tracer in PMaC), or a static analysis. The performance of the processor (e.g., the peak performance of each level of the cache hierarchy and of the memory) is measured independently by a benchmark (MAPS in PMaC) and both series of values are convolved using the MetaSim convolver through a formula (e.g., the maximum or the sum over cache levels of the products of the hit rate, the number of accesses and the inverse of the bandwidth). Likewise, the sequence of MPI operations are traced (with MPIDtrace in PMaC) and the characteristics of the network are benchmarked (with PMB [9] in PMaC) and later convolved (with the DIMEMAS [5] discrete event simulator in PMaC). Other simulators differ in the degree of sophistication of the underlying models but the general trend remains similar.

Many network performance models [10], [11], [12], [13], [7] have been developed since the last 30 years. Memory performance modeling, on the other hand, is much more recent, especially in the context of multi-core machines. Complex architectures with the undisclosed hardware implementation details, as well as the user-transparent memory operations,

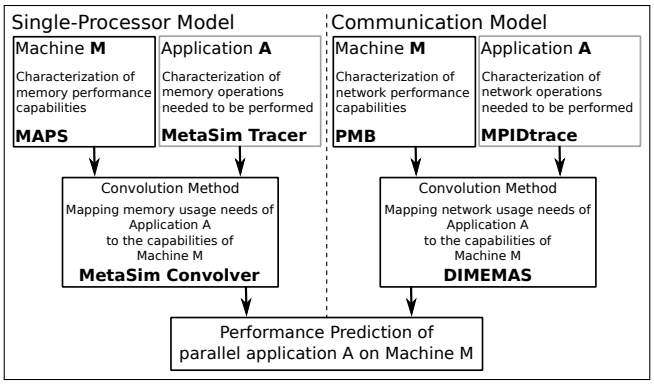

Figure 1. General approach proposed in the PMaC Framework [1] to predict performances of HPC applications: application and machine signatures are convolved through a network and memory model.

make it difficult to create a proper model, because it is hard to obtain direct and faithful observations. In what follows, we briefly discuss the efforts regarding network and memory performance characterization.

B. Network Performance Characterization

Network performance modeling generally involves three aspects: the adoption of parameters originally described in the LogP model [14], the synchronization mode and the presence or not of piecewise modeling for each of the modes. In LogP, o is the software overhead and models the CPU occupation per message, L is the minimal transmission delay over the network (latency) and g is the gap between two messages (i.e., the invert of the bandwidth). One usually distinguishes between three synchronization protocols: eager, detached and rendez-vous. Finally, the piecewise modeling relates to the fact that different values for the previous parameters may be used depending on the range in which the message size falls. The more elaborate models comprise all such aspects and are generally needed to capture the behavior of modern MPI implementations and interconnects.

Several tools enable the measurement of network parameters such as latency, gap and software overhead. The Pallas MPI Benchmarks (PMB) suite [9], SkaMPI [15], Conceptual [16], NetGauge [17] and Confidence [18] all demonstrate unique qualities in such measurement task. The Pallas MPI Benchmarks (PMB) suite [9] provides a framework to measure a subset of MPI operations, and is detached from a performance model. It characterizes the network performance, identifying potential problems and improvements. PMB only reports mean values for each requested message size and number of repetitions.

SkaMPI and Conceptual feature a Domain-Specific Language (DSL) to describe how experiments should be accomplished. While SkaMPI focuses only on MPI, Conceptual has a much broader set of backends, including MPI. Both are capable to generate programs very rapidly with few lines of DSL code. Unlike the previous two tools, NetGauge

also provides a way to explicitly output all the necessary parameters to instantiate the LogGP [10] and PLogP [11] models for a given machine. NetGauge supports many communication protocols including InfiniBand, Myrinet/GM, TCP/IP as the Ethernet Datagram Protocol (EDP) and the Ethernet Streaming Protocol (ESP), and MPI.

Finally, the authors of Confidence [18] note that many sources of performance variability can be found in modern HPC systems (e.g., OS noise, network collapse or transient effects resulting from user timeshare) and focus on reporting the variability that users may actually face and which is hidden by common benchmarks. Such information about variability could be used for simulation purposes provided it is adequately combined with message size dependencies.

C. Memory Performance Characterization

Roofline estimations [4] are the simplest way to estimate memory access performance. The principle of such benchmarks is to saturate the memory utilization, effectively defining the peak access rate (GB/s). The Performance Modeling and Characterization (PMaC) framework [1] relies on the MultiMAPS benchmark to measure the memory bandwidth for different data sizes, strides and line sizes. By changing the data sizes, this benchmark captures different characteristics of the memory cache levels. PChase [19] extends MultiMAPS by assessing memory latency and bandwidth on multi-socket multi-core systems, capturing the interference between CPUs and cores when accessing memory, and ultimately providing a richer model.

III. Network Layer Performance Modeling

The simplest network characterization strategy consists in measuring the time taken to transfer a given amount of data between two endpoints. Measurement variability should also be assessed since the network stack interactions are complex. Figure 2 presents this prevailing approach shared by many tools (described in Section II). Increasing the data size in powers of 2, the benchmarks measure N times the same experimental configuration. In this example, a simple send/recv is placed to illustrate the principle although more complex operations may be also used (e.g., non-blocking, one-sided, collective operations). The measure operation can also be a more complex pattern involving several message exchanges, as in the LogGP [20], LoOgGP [21], and the PLogP [11] benchmarks that directly output model parameters. In most strategies, per message size measurements are used to calculate statistics (bandwidth, latency, etc.) in an online fashion. If piecewise modeling is concerned, the behavioral breaks are automatically detected during the experiment, using linear extrapolations from the already measured points or outlier definition schemes. At the end of the main loop, tools report the aggregated results per operation and per data size in textual or CSV file format for external exploitation.

```
for datasize in (0, 1, 2, 4, 8, ..., 2^16) {
  for repetition in (1..N) {
    timer_start();
    if (sender) Send() else Recv()
    timer_stop();
  }
  # Compute statistics (average, standard deviation);

  # Test for performance rupture and possibly
  #      reset the corresponding breakpoints;
}
# Summary report of network metrics for each data size
```

Figure 2. Pseudo-code to measure network performance, varying the data size and repeating the experiment to quantify variability; with on-the-fly statistics, piecewise models' breaks are immediately defined.

The automatic detection of protocol changes depending on the message size has been the object of several strategies. NetGauge [20], PLogP [11] and LoOgGP [21], for instance, provide good examples. When linearly increasing the message size, and for every new measurement, NetGauge checks for protocol changes by using the mean least squares deviation (lsq) between the previous point that started a new slope and the latest measurement. If the lsq has changed more than a factor defined by the analyst, NetGauge waits for five new measurements before confirming the protocol change. This technique avoids that "anomalous" measurements mislead the detection of true protocol changes. In PLogP, at every new measurement when increasing the message size in powers of 2, the implementation extrapolates the previous two measurements and checks if the difference between the new measurement and the linear extrapolation is within an acceptable range. If that is not the case, a new measurement is undertaken with a message whose size is the mid-value between the latest two measurements. This is repeated, halving the intervals, until the extrapolation is matched by measurements or a maximum number of attempts is attained. The LoOgGP linearly increases the message sizes, using the same approach as NetGauge, but adopts an offline analysis with user intervention. After removing outliers, a local neighborhood of a configurable extent is defined for each measurement. If a measurement has a maximum value in a neighborhood, it is considered as a protocol change. It is up to the analyst, using plots for $T_o(s)$ (overhead) and $T_g(s)$ (gap), to finally decided if detected points are real protocol changes or no. Despite the analyst mediation, as authors state, the mechanism is sensitive to the neighborhood size and the message size steps during the measurement stage.

We detail potential problems faced by such common methods to model network performance. Our recommendations to avoid such pitfalls are presented in Section V.

A. Evaluation Pitfalls

1) Impact of Temporal Perturbations: The system being measured may suffer temporal perturbations which may mislead completely the result interpretation. Perturbations can be natural or caused by external activity in a poorly

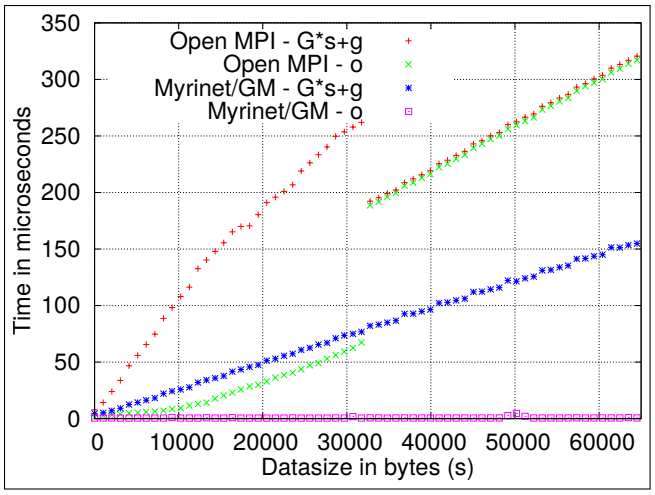

Figure 3. Time as a function of message size for different communication libraries using a Myrinet/GM network (taken from [20] as example).

isolated system. To repeat the benchmark would probably lead to divergent estimations. Without a manual check, the measurements can be filtered out as outliers, or affect the online detection of protocol changes. An anomaly could pass as a break point by the heuristics implemented in NetGauge and PLogP, for instance. Concerning results, there is no guarantee that the reported breaks are actually meaningful.

2) Impact of Message Sizes in the Network Modeling: Another issue regards the input message sizes used in the experiments, which is, very often, biased. Using messages in powers of 2 may miss the real behavior of the network software stack. Some values, such as 1024 for instance, may have special behavior coded in the network layers that are nonlinear when compared with close values directly smaller or larger than that one. Such small changes in the data sizes might pass undetected by statistical analysis that is conducted without supervision. On benchmarks that use linear increments, such as NetGauge and LoOgGP, the bias issue is still present because the measurements depend on the selected starting message size and the increment value.

3) Impact of Preconceived Assumptions in the Analysis: Figure 3, reported by Hoefler et al. [20], demonstrates a typical case of a piecewise network modeling, detecting protocol changes. They report a single protocol change for messages larger than 32KBytes. However, a review can indicate another break at 16KBytes, when the slope of the OpenMPI ($G * s \times g$) and OpenMPI (o) slightly change. So the assumption of the presence of a fixed number of breakpoints might mislead the network modeling.

Current MPI implementations might have several synchronization modes (eager, detached, rendez-vous) and each of these might need a piecewise modeling. Figure 4 depicts an example for the Grid5000's Taurus cluster (OpenMPI 2.0.1 with TCP and 10Gb Ethernet). We depict the network modeling of the send overhead (left), receive overhead (center), and the network latency and bandwidth (right). Raw observations (points) and the piecewise linear (the lines are bent because of the logarithmic scale) regression calculated from them (black line) are plotted as a function of the message size. The color indicates probable variations in the communication protocols. The receive operation (blue area on the center plot) for the medium message size has a much higher variability than for other message sizes. The same happens, but with a different pattern, for the send overhead (yellow on the left). In our case, since sizes were randomized (instead of taking measurements in an incremental order, as it is commonly done), we can safely conclude that this variability is a real phenomenon and not an artifact.

IV. Memory Hierarchy Performance Modeling

Our initial objective was to build upon MultiMAPS benchmark. This benchmark is an upgraded version of MAPS benchmark, which itself is derived from STREAM [22]. Figure 5 presents the pseudo-code of the main algorithm behind MultiMAPS. It measures the time to execute the for loop in which it makes consecutive memory accesses (by stride) to an array of elements. At the end, it computes the memory bandwidth. Two factors affect such bandwidth estimation: the buffer size and stride of access by which the buffer is traversed. Together, they should capture the temporal and spatial locality behavior of the memory hierarchy.

Figure 6 shows the typical MultiMAPS results, in this case for strides 2, 4 and 8 on an Opteron machine (2.8GHz with 2 level of caches where L1 cache is a 64KB 2-way associative cache and L2 cache is a 1MB 16-way associative cache). As the size of the buffer increases, the memory bandwidth decreases. Three plateaus directly correspond to the L1 cache, L2 cache, and main memory sizes. Strides have no impact when all accesses are done inside L1. However, they play an important role when the array size no longer fits in L1, since bandwidth is almost reduced by a factor 2.

Such behavior seemed quite regular and sound so we envisioned to directly use the MultiMAPS benchmark for characterizing memory capabilities. Our goal was to improve the SimGrid processor model using an approach similar to the one of the PMaC framework [23]. However, when trying to reproduce this approach with recent hardware, several unexpected difficulties forced us to change our initial plan.

First, a closer inspection of the MultiMAPS code revealed that it is much more complicated than the simple STREAM benchmark on which it was based. Many parameters require careful tuning, which is extremely hard without prior knowledge of the benchmark and the machine it is running on. We anticipated that we would have to modify many parts of the benchmark to understand more deeply the performance on modern architectures. A second issue was that the output of the benchmark is very verbose and it is unclear how the final plot is obtained from the measured data. Finally, our initial results were far from the expected bandwidth

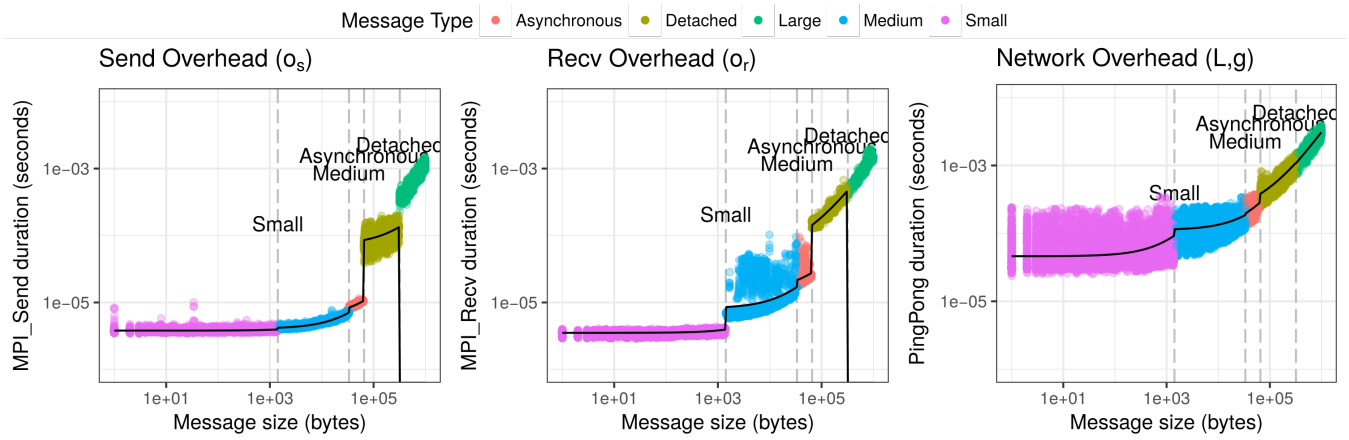

Figure 4. Network modeling of the Grid'5000 Taurus cluster: multiple protocol changes require an initial neutral look regarding the number of breakpoints.

peaks, which was surprising as we thought that achieving the maximal performance should be relatively easy with such simple programs. Unfortunately, the benchmark only reported aggregated values, without raw data to allow us to better understand where the problem could come from.

Therefore, we decided to try to reproduce the essence of the MultiMAPS approach, but with our own code. We wrote our own benchmark script inspired by MultiMAPS, trying to mimic the spirit of the reported experiments.

```
MultiMAPS(size, stride, nloops) {
  allocate buffer[size];
  timer_start();
  for rep in (1..nloops)
    for i in (0..size/stride)
      access buffer[stride*i];
  timer_stop();
  bandwidth=(naccesses*sizeof(elements))/elapsed_time;
  deallocate buffer;
}
```

Figure 5. Pseudo-code of the benchmark allowing to evaluate performance in the different levels of the cache hierarchy.

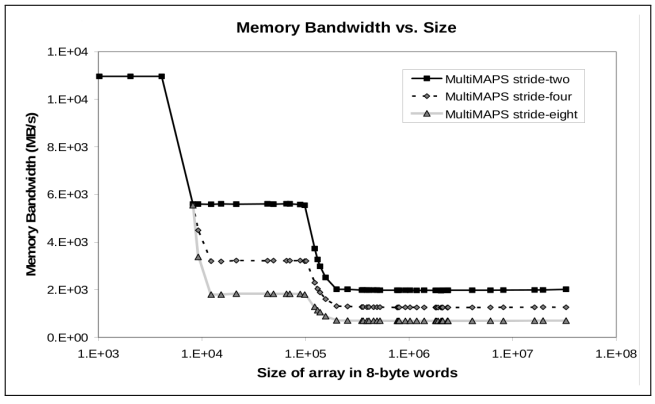

Figure 6. Excerpt from [23] exploiting the output of MultiMAPS and illustrating the impact of the working set and of the stride on performance.

Although we aimed at studying all levels of the memory hierarchy with parallel execution, we quickly discovered that there is huge number of challenges even for the simplest case. Consequently, we restrict our investigation report to characterize solely L1 cache READ bandwidth, for a single-threaded program.

Building on our previous experience with networking measurement (see Section III), our first concern was to thoroughly randomize each parameter of the micro-kernel (stride and size). Figure 7 shows the first experimental results. Each dot represents one measurement (42 repetitions for each configuration) of the bandwidth as a function of the buffer size. The color indicates the stride, while the solid lines represent smoothed local regressions indicating measurement trends. It is clear from such graph that there is an enormous experimental noise for every buffer size. Furthermore, even when ignoring the variability and focusing solely on mean values, the influence of the stride is ambiguous and bandwidth does not decrease by a factor of two as one could expect. These results are far from what was

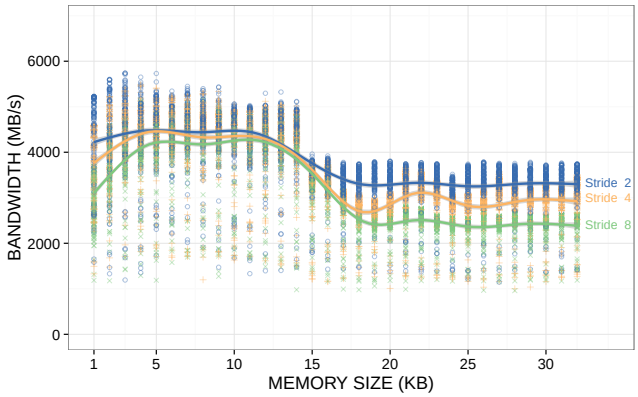

Figure 7. Attempt to replicate the behavior shown in Figure 6; our measurements our much more noisy despite the controlled environment.

expected and they are very different from the ones presented in Figure 6. We thoroughly investigated the reasons behind such behavior and can now report several evaluation pitfalls. To illustrate different phenomena on the simplest cases, the rest of the results and figures in this section are limited to the measurements conducted with the stride 1.

A. Evaluation Pitfalls

1) Impact of code and compiler optimization (loop unrolling and vectorization): In the previous results, it was assumed that the absolute value of the bandwidth depend mostly on the processor and memory bus frequencies, as the compiler will automatically optimize the corresponding simplistic kernel. Nevertheless, there are some programmers optimizations that can have a significant influence on the bandwidth values, sometimes (surprisingly) negative.

The first optimization concerns the type of the element the array is composed of. In the initial version of the code, the buffer is a large array of integers. Integer size in C language is 4Bytes, so if the whole buffer size is 1KB the buffer is composed of 1024B/4B=256 integers. If the buffer type is changed for *long long int* which is 8B, for the same array size of 1KB there would be two times less elements in the buffer (1024B/8B=128). The loop iterator traverses the whole buffer element by element, multiplied by stride. Hence, for the *long long int* (8B) case, we expect there are two times less accesses than for the initial *int* (4B), resulting in a higher bandwidth. This is a good example of how data level parallelism and vectorization can improve the performance.

The second optimization is related to the loop unrolling. This code transformation attempts to minimize branch penalty by unwinding the code inside the loop. Usually, such attempt reduces the execution time, but with a larger program binary size. Common sense implies that automatic loop unrolling present in many compilers always leads to performance gains. However, as we shown in Figure 8, even for the simple code of Figure 5, manually unrolling the loop proves to be very beneficial for performance.

Figure 8 shows how increasing element type from 4B *int* to 8B *long long int* essentially doubles the bandwidth. The same trend, only a bit mitigated, continues with the larger elements. Loop unrolling also has a positive effect, as the bandwidth increase in all cases except one. For the 32B vectorized instructions with loop unrolling, instead of the expected highest values, the actual results are extremely low. We did not fully investigate the reasons behind this anomaly, as such understanding is secondary to this work, and we were constrained by the time. We strongly believe that it is due to the fact that 32B vectors on this Intel architecture are composed of *double* and not *int* values.

Another very important phenomenon demonstrated by these plots is related to the point at which performance drops. One can observe that difference between the buffer

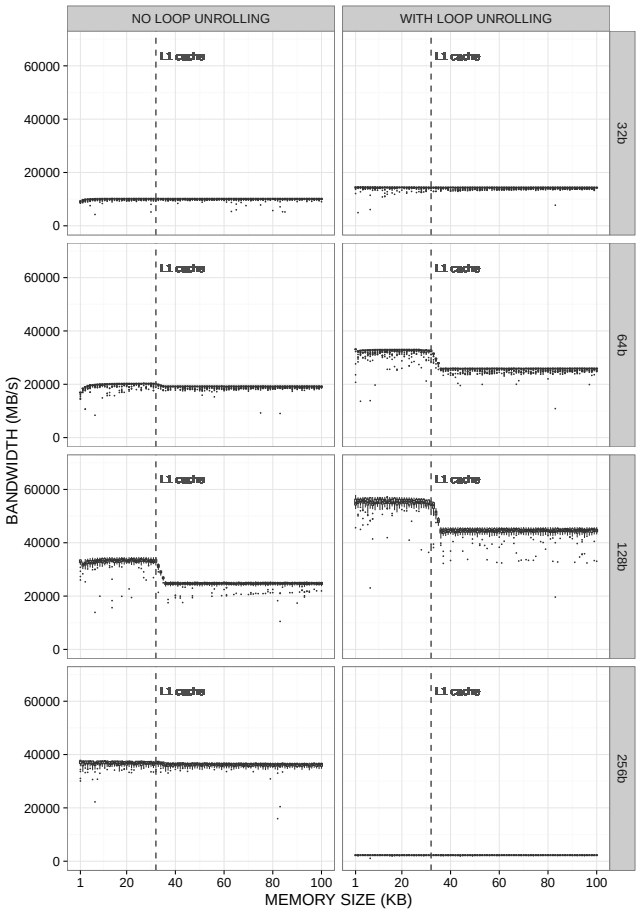

Figure 8. Vectorization (vertical facetting) and loop unrolling (horizontal) effects on memory bandwidth (on Y), for different buffer sizes (on X).

memory size that is fitting L1 cache and the bigger ones is becoming more noticeable as the absolute values of bandwidth increase. Even worse, for the 4B element type there is no drop at all when buffer size is getting out of the cache size. The main cause for such behavior is that we are not using the full processor capacity. When loop unrolling is added and the element size increased, we go towards the true performance limits of the processor. Hardware limitations becomes more apparent.

This suggests that there could be other deviations hidden when the full potential of the machine is not reached. It should be noted that that only with these two factors (element type and loop unrolling), very different results can be obtained.

2) Impact of Dynamic Voltage and Frequency Scaling (DVFS): Recently, it has become prohibitive to constantly keep the processor frequency as high as possible, due to the extensive energy consumption. Indeed, there is an increasing number of reported applications where lowering the processor frequency can lead to significant energy gains without affecting the overall execution time.

Therefore, to decrease the energy consumption, many pro-

cessors nowadays operate with *ondemand* CPU frequency governor. This policy enables operating system to scale frequency up or down, choosing the most appropriate one from several possible values (modes). These changes might take effect during the application run if specific parts of the code have different computational needs. From the user's perspective, it is very hard to know exactly how this mechanism works. It is generally known that high frequency is used for computation intensive code blocks, while low frequency is used for memory-bounded regions. Nevertheless, there are also the borderline cases which are hard to optimize and for which frequency predicting is extremely challenging.

One example of such application is the cache memory benchmark code, presented in Figure 5. In this program, elements of the array are accessed in a loop and this is repeated *nloops* times. Increasing the number of repetitions should proportionally increase the overall time to run the program, thus the *nloops* parameter should not have any influence on the final bandwidth. However, it proved to be the opposite for the Intel Sandy Bridge machine with *ondemand* governor, as shown in Figure 9. In fact, when the amount of work required from the processor is low, the operating system decides to use the lowest frequency, thus the array is accessed slower and the resulting bandwidths are low (top left plot of Figure 9). On the contrary, when the operating system anticipates a large number of accesses, it increases the frequency to the maximum value, which leads to the highest bandwidths (bottom right plot). The most interesting results can be observed for the intermediary values of *nloops* where operating system dynamically changes its decision about the optimal frequency for a given code block. Therefore, for each of the 42 repetitions of the same buffer size, the measured bandwidth varies between several modes and the performance variability is higher. This illustrates that even for the simplistic codes it may be extremely difficult to anticipate which frequency will be used, and thus what will be the corresponding performance.

One may argue that such behavior can be avoided if user takes full control of the processor frequency. However, this requires some programming effort and an expertise in the performance of every part of the application. Every type of operating system with the underlying processor might have a different mechanism for controlling frequency, with contrasting latency for changes to take effect. Moreover, in some cases frequency changes required superuser rights that often are unavailable on production platforms. Finally, the *ondemand* governor has many advantages and most of the users want to keep part of its benefits, even though its behavior can sometimes be unpredictable.

3) Impact of Operating System Scheduler: Despite the simplicity of the single-threaded benchmark, and the exclusive access to the platform while performing experiments, inevitably some external influence still exists. Primarily, there is an operating system with its own processes running

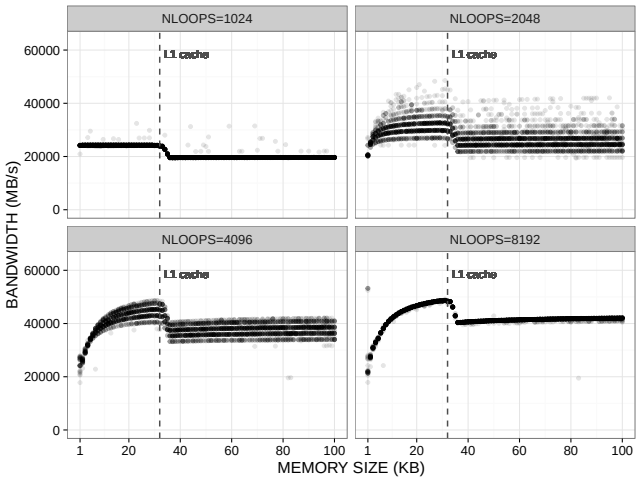

Figure 9. Memory bandwidth as a function of the buffer size for four workloads (facets) as indicated by the *nloops* parameter of code of Figure 5.

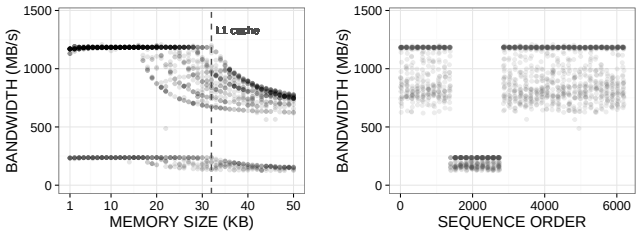

Figure 10. Real-time scheduling priority on an ARM Snowball processor: the left plot shows the bandwidth according to the buffer size, with two modes (higher and lower); the right, as a function of the sequence on which measurements are taken, highlighting the temporal anomaly.

on the processor. To minimize its influence, we removed all the unnecessary OS services. Additionally, the core on which our program is executed was strictly pinned, avoiding potential thread migrations during the execution.

Our past experience in evaluation of lightweight thread libraries had thought us that better and more stable performances can sometimes be obtained by using the real-time scheduling policy of the OS. However, this proved to be the opposite when running experiments on ARM Snowball processor with Unix-like Linaro OS. The left plot of the Figure 10 shows that there are two modes of execution. The first mode, the one with higher bandwidth values, is similar to the results we have obtained with other scheduling priorities. On the other hand, the second mode has the bandwidth values that are almost 5 times lower and they occur in approximately 20-25% of the measurements. There are 42 repetitions for each buffer size and the order in which single runs are executed is completely randomized. Approximately the same number of the second mode executions is present for all buffer sizes. The right plot of the Figure 10 shows exactly the same data as the left plot. The difference is that the x-axis now represents the order in which each measurement inside one experiment were conducted. The

right plot of the Figure 10 indicates that the whole second mode occurred throughout a single period of time during the whole experiment execution. When repeating experiments with different input configurations, the same phenomena was regularly reproduced. This suggests that the second mode is almost certainly caused by an external process running in parallel and which is occasionally scheduled to the same core when the real-time policy is activated.

There are two techniques that enabled us to easily spot the described phenomenon, while it would have probably passed unnoticed or misunderstood using classical approaches. First, if measurements had been done in a commonly used sequential order, they would wrongly suggest poor performance for a specific subset of buffer sizes. Randomizing order in which single measurements inside one large experiment are performed, solves this issue. Second, it was important to postpone the data aggregation and to log all the relevant information during the experimentation stage. By looking solely at mean bandwidth values and variance, which is a simplistic approach dominantly used in our community, the performance obtained with real-time policy appears worse, but the existence of two modes is completely hidden.

4) Impact of Architecture (the ARM paging issue): Figure 11 shows the result of four consecutive experiments on an ARM Snowball processor using exactly the same source code and inputs. The 42 repetitions for each memory size (on each plot) are represented by boxplots, demonstrating little measurement variability (thanks to a careful control and optimization of the experiments). Very surprisingly, the performance drop, as the buffer size increases, occurs at different places depending on the experiment, just as if some "external entity" was deciding at the beginning of the experiment the kind of very stable performance one would observe. Although the lower and higher values of buffer size always exhibit a similar behavior, the middle part (from 50% to 100% of the L1 cache size) is highly unpredictable. After a long investigation, we finally discovered that the source of this surprising phenomenon comes from the way the operating system allocates physical memory pages on ARM processors. In general, operating systems allocate nonconsecutive 4KB physical memory pages, choosing them randomly from a pool of available pages. The set-associativity of that generation for ARM processors is only 4, and the L1 cache size is 32KB. In such scenario, without doing the appropriate page coloring, a bad choice regarding physical pages causes more cache misses, hence the drop of overall performance. During one experiment run, although we do malloc/free repeatedly on each buffer, the same pages gets reused. Hence, the buffers actually start from the same physical memory location for each memory size during one experiment, which explains why there is no variability in the results despite the measurement randomization.

Since it is not possible for a user program to fully

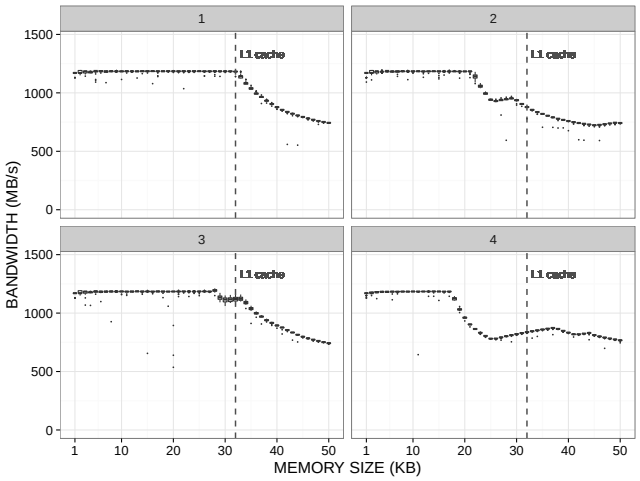

Figure 11. Four experiments (facets) on the ARM Snowball processor: boxplots depict the memory bandwidth as a function of the buffer size; an anomaly appears when the buffer size gets close to the L1 cache size.

take control of the choice of page allocation, we applied an alternative memory allocation technique for the buffer, in order to assess better this phenomenon. In the initial algorithm, we used an individual malloc function call for each buffer size, running the experiment, and finally freeing the memory at the end of the loop. In the subsequent versions, we decided to only one memory allocation at the beginning of our program for all the measurements. We would allocate one big memory block (e.g., 2 MB), much bigger than our maximum buffer memory size (50KB in these experiments). After that, for each memory size and repetition, we randomly chose the starting point inside our big allocated memory to consider it as buffer for the experiment. This way we managed to correctly evaluate the impact of different physical memory pages during one experiment run, avoiding to always use the same pages. This physical address randomization allowed us to obtain completely reproducible (although more variable internally) experiments, which provide always similar bandwidth values (typically, the top part of the left graph in Figure 10).

If we had used the second allocation technique from the start and concentrated only on maximum/median/mean values of bandwidths, we would have observed very regular behavior with a clean drop on L1 cache size. But we would have missed this physical memory allocation phenomenon, ultimately leading to incorrect model instantiating.

V. Proposal of a Comprehensible Experimental Methodology for Performance Benchmark

Following the fundamentals of the Design of Experiments [24], we propose a three-step methodology: (1) the experimental design, (2) the benchmark running engine, and (3) the corresponding statistical analysis. We believe that separated stages, together with careful documenting and

environment capture, enable us to avoid all pitfalls that have been previously presented.

The experimental design deals with the factors that drive the system behavior under evaluation. Randomization plays a central role: each factors' values and the order in which each factor combination is measured should be properly randomized. This guarantees that the presence of temporal anomalies in the setup remains independent on the factors' values. Uncertainty is evaluated and reduced through replicated measurement of each factor combination. During the second step, the order on which measurements are taken must be dictated by the experimental design. Thus, the benchmark engine reads each factor combination from its input, conducts the measurement on the target platform, and reports the details of every individual measurement in one or multiple output files, along with a lot of meta-data about the measurements and the environment (machine information, operating system and compiler versions, compilation command, benchmark parameters, network configuration, etc.). Beyond increasing the chances for reproducing the experiments, these meta-data enable better results interpretation. This is especially useful when performing comparison between two experimental campaigns that have similar inputs and completely different outputs. Statistical analysis is carried out during the third stage of our methodology, after the experiment campaign execution has finished. We avoid doing any on-the-fly aggregation and keep all information, delaying the analysis, in order to spot the outliers and strange behaviors, instead of losing them.

All source code and raw data are available (network[1] and memory[2]) to anyone interested in reviewing or running our methodology on other platforms.

A. Network Measurement Methodological Details

The factors to consider to instantiate our model are the message size, and the synchronization mode. To this end, we relied on three kinds of operations: blocking receive, asynchronous send, and ping-pong. The send and receive software overhead are calculated through a blocking receive and an asynchronous send, respectively. The benchmark guarantees that the message has already arrived in the receiver when the receive operation is called. In both cases, the elapsed CPU time captures the overhead of buffering and sending data to the network card. The network latency and bandwidth are calculated using the results of the ping-pong operation. These three network-related operations are sufficient to calculate all the parameters for any LogP-based model. To characterize the performance of communications for message sizes between a and b (in bytes), we generate random sizes using the following distribution:

$$10^X, \text{ where } X \sim Unif(\log_{10}(a), \log_{10}(b)) \qquad (1)$$

[1]Code available at https://gitlab.inria.fr/simgrid/platform-calibration/
[2]Code available at https://gitlab.inria.fr/stanisic/cache_reppar17/

The resulting combinations of operation type and sizes, one per line, are registered in a textual file that is provided to the measurement engine (the code is available in the SimGrid's platform calibration repository). This engine is a simple MPI program that registers the time it took to run the particular operation on the target network for all the message sizes indicated in the output. Finaly, in the third step of our methodology, a script written in the R Language [2] carries out a supervised analysis. The breakpoints are manually provided by the analyst and a piecewise linear regression is calculated for each of the three operations. The send and receive software overhead are measured using the blocking receive and the asynchronous send, latency and bandwidth are obtained using the ping-pong measurements. Plots are generated so a human can check the linearity assumption, if the breakpoints are coherent, and the outcome of the regressions. Despite being manual, this procedure guarantees that results are meaningful from an experienced analyst point of view.

B. Memory Measurement Methodological Details

Despite the simplicity of our memory benchmark (see Section IV), the factor set revealed much larger than what we initially thought. These parameters, grouped by features, are presented in Figure 12 in a modified "cause-and-effect" diagram [24]. We have written a small independent code

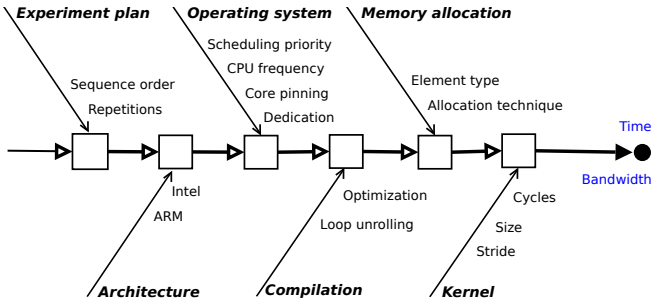

Figure 12. Influential factors to be carefully managed during experiments.

which considers all the factors listed in Figure 12. This code randomizes the order of the experiments, combining all involved factors. As output, it generates a CSV file that is the textual representation of the design. The CSV is used as an input by the benchmark which is, in the second stage of our methodology, a C program based on the code listed in Figure 5. It executes all the experiments and reports the memory bandwidth for every factor combination. The analysis is carried out using literate programming in orgmode/R.

VI. Conclusion

In this paper, we presented our attempt to reproduce the approach of previous work in the context of network and memory modeling. The model instantiation generally

builds on opaque benchmarks that can lead to many pitfalls. The investigated phenomena are so complex that simplistic approaches can lead to severely biased measurements that make simulation predictions unreliable. We explain how such biases can be avoided through relatively simple precautions. Thorough randomization is an essential ingredient but should also be allied to a white-box approach with a clear separation of concerns (experiment design, experiment engine, result analysis) and a careful documentation and capture of the environment.

In a near future we plan to work on automating and combining various tools we have built to instantiate HPC network models while keeping the same white box and randomization methodology. One of the challenges will be related to the production of a coherent and easily understandable report over a complex set of measurements and allowing to reliably characterize a whole cluster.

Acknowledgments

The authors would like to thank the SimGrid team members and collaborators who contributed to SMPI. This work is partially supported by the Hac Specis Inria Project Lab, the ANR SONGS (11-ANR-INFR-13), the European Mont-Blanc (EC grant 288777), the EU/H2020 and the MCTI/RNP-Brazil under the HPC4E Project with the grant 689772, the CNPq 447311/2014-0, and the CNRS/LICIA Intl. Lab. Experiments are carried out on the Grid'5000 experimental testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and other Universities and organizations (see https://www.grid5000.fr).

References

[1] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A framework for performance modeling and prediction," in *ACM/IEEE Conference on Supercomputing*. Los Alamitos, CA, USA: IEEE, 2002.

[2] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2016. [Online]. Available: https://www.R-project.org/

[3] E. Schulte *et al.*, "A multi-language computing environment for literate programming and reproducible research," *J. of Stat. Soft.*, vol. 46, no. 3, 2012.

[4] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Comm. of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[5] R. M. Badia, J. Labarta, J. Giménez, and F. Escalé, "Dimemas: Predicting MPI Applications Behaviour in Grid Environments," in *Proc. of the Workshop on Grid Applications and Programming Tools*, Jun. 2003.

[6] G. Zheng, G. Kakulapati, and L. Kalé, "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," in *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2004.

[7] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim: Simulating large-scale applications in the LogGOPS model," in *ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 597–604.

[8] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms," *Journal of Par. and Distr. Comp.*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014.

[9] Pallas, "Pallas MPI Benchmarks – PMB, Part MPI-1," *Hermülheimer: Pallas GmbH*, 2000.

[10] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages Into the LogP Model – One Step Closer Towards a Realistic Model for Parallel Computation," in *ACM Symposium on Parallel Algorithms and Architectures*, 1995, pp. 95–105.

[11] T. Kielmann, H. E. Bal, and K. Verstoep, "Fast measurement of logp parameters for message passing platforms," in *International Parallel and Distributed Processing Symposium*. Springer, 2000, pp. 1176–1183.

[12] F. Ino, N. Fujimoto, and K. Hagihara, "LogGPS: a Parallel Computational Model for Synchronization Analysis," in *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, Snowbird, UT, 2001, pp. 133–142.

[13] J.-A. Rico-Gallego, J.-C. Díaz-Martín, and A. L. Lastovetsky, "Extending τ-lop to model concurrent MPI communications in multicore clusters," *Future Gen. Comp. Syst.*, vol. 61, 2016.

[14] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 1993, pp. 1–12.

[15] R. Reussner, P. Sanders, and J. L. Träff, "SKaMPI: A comprehensive benchmark for public benchmarking of MPI," *Sci. Program.*, vol. 10, no. 1, pp. 55–65, Jan. 2002.

[16] S. Pakin, "The design and implementation of a domain-specific language for network performance testing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 10, 2007.

[17] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm, "Netgauge: A Network Performance Measurement Framework," in *Proceedings of High Performance Computing and Communications*, vol. 4782. Springer, Sep. 2007, pp. 659–671.

[18] B. W. Settlemyer, S. W. Hodson, J. A. Kuehn, and S. W. Poole, "Confidence: Analyzing performance with empirical probabilities," in *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, Sept 2010, pp. 1–8.

[19] A. Mandal, R. Fowler, and A. Porterfield, "Modeling memory concurrency for multi-socket multi-core systems," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010, pp. 66–75.

[20] T. Hoefler, A. Lichei, and W. Rehm, "Low-overhead loggp parameter assessment for modern interconnection networks," in *2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007, pp. 1–8.

[21] D. R. Martinez, J. C. Cabaleiro, T. F. Pena, F. F. Rivera, and V. Blanco, "Accurate analytical performance model of communications in MPI applications," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009.

[22] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Technical Committee on Computer Architecture lNewsletter*, Dec 1995.

[23] M. M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely, "A Genetic Algorithms Approach to Modeling the Performance of Memory-bound Computations," in *ACM/IEEE Conference on Supercomputing*. ACM, 2007, pp. 47:1–47:12.

[24] D. C. Montgomery, *Design and analysis of experiments*. John Wiley & Sons, 2008.