DOI: 10.14529/jsfi140201

Scalability prediction for fundamental performance factors

Claudia Rosas[1], *Judit Giménez*[12], *Jesús Labarta*[12]

Inferring the expected performance for parallel applications is getting harder than ever; applications need to be modeled for restricted or nonexistent systems and performance analysts are required to identify and extrapolate their behavior using only the available resources. Prediction models can be based on detailed knowledge of the application algorithms or on blindly trying to extrapolate measurements from existing architectures and codes. This paper describes the work done to define an intermediate methodology where the combination of (a) the essential knowledge about fundamental factors in parallel codes, and (b) detailed analysis of the application behavior at low core counts on current platforms, guides the modeling efforts to estimate behavior at very large core counts. Our methodology integrates the use of several components like instrumentation package, visualization tools, simulators, analytical models and very high level information from the application running on systems in production to build a performance model.

Keywords: parallel efficiency, curve-fitting, exascale computing, analysis and prediction.

Introduction

Within the race toward exascale computing, to infer the scaling capacity of current parallel codes has become essential [1]. If we are able to identify primary factors that define the efficiency, we can use them to predict the scalability of the code. Systems and codes are getting more complex and their performance analysis may result costly in time. At the same time, validation on current non-existent machines is not an option, and in consequence, scientists must take advantage of already known techniques and tools to overcome these restrictions. Tools are helpful to identify, in a short time, the primary factors of real parallel codes executed in the available machines. Then, collected information can be used to outline potential restrictions on future computational systems [2].

Different philosophies have been followed to perform prediction studies in the past. The approaches range from a vision based on first principles on one side to blind fitting of metrics and extrapolation on the other. An effort to investigate the performance of MPI applications at large core counts uses parallel discrete event simulations to run the application in a controlled environment [3]. Most of them demand from specific models or abstractions of the parallel code (and the system) [4], [5] or massive fittings of time-based metrics to predict performance of specific functions [6]. However, there is low information about the insights of the real underlying cause of the inefficiencies, and the knowledge about the influence of different architectural characteristics can be useful to improve the code.

This work further develops the modeling and extrapolation tasks initially presented in [7], broadening the scope and scale. We consider that the components that represent essential features of the program and their evolution may be related to primary models of parallelism such as Amdahl's law. This proposal starts by capturing detailed data from traces of very few runs of the parallel code at low core counts in machines that are in production, i.e. without additional tunings for exclusivity. From the traces, significant performance components such as load balance and transfer can be measured, fitted and extrapolated at large core counts.

Our method relies on the detailed preprocessing of available traces to determine appropriate sections with minimal noise perturbation. Two arguments sustain the use of traces. First, having

[1]Barcelona Supercomputing Center (BSC), Barcelona, Spain
[2]Technical University of Catalonia (UPC), Barcelona, Spain

few points to fit, the blind use of functions with many parameters would lead to undetermined systems with many possible solutions in the explored core count range. These solutions may have huge differences in performance when extrapolating for large core counts. Second, it is our belief that low core count runs provide enough information on the fundamental behavior of parallel code, and can easily complement existent time profiles reports of the different routines.

We provide an automatic framework to produce the models and reports, automatizing the error-prone task of collection and processing the measurements to increase the availability of the analyst for observation, interpretation and let him/her focus on the factors or significant interest. The analyst can perform additional measurements, or what-if predictions, using analysis and simulation tools (Paraver, Dimemas, Clustering, etc.) to find out outliers and for better understanding of the fundamental factors. Collected data is used to model the expected performance of an application in that specific machine for larger core counts.

The use of fundamental factors is highly informative for developers and can guide optimization efforts in the most productive direction. For example, for an application whose main problem is load balance it is highly non-productive to spend time and energy re-factoring it using non-blocking MPI calls, even if the standard time profile indicates that a large fraction of the time is consumed by MPI. Compared to the first principles approach, our method does not require prior knowledge from the analyst on the code nor a specific modeling effort for each new one.

This work combines a set of steps that can be performed semi-automatically to reduce the total time for data extraction and processing. The main stages are:

- **Identify Structure:** starting with traces from an instrumented execution of the parallel application at different core counts identify relevant sections to analyze and extrapolate;
- **Phase Performance Analysis:** compute fundamental efficiency factors (load balance, serialization, transfer) for each identified region;
- **Scalability Prediction:** extrapolate the fundamental efficiency factors computed at low core counts to infer their evolution at large core counts.

The main contributions of this work are:

- A methodology to extrapolate the efficiency computed at low core counts to large core counts in the same platform;
- A validation of the methodology on 4 cases corresponding to different applications, platforms and strong or weak scaling runs;
- An extension of the methodology to predict the impact at large core counts of architectural or system improvements (in particular OS and network noise elimination).

The rest of this paper is organized as follows. Section 1 outlines our approach and describes the underlying prediction model in detail. Section 2 presents the experiments performed to evaluate the effectiveness of our method. Related work is discussed in Section 3, and final remarks and further steps are in Section 4.

1. Description of the methodology

1.1. Identify Structure

Our method begins with a trace for a number of MPI ranks, obtained from executing an instrumented parallel code or by simulating the execution on a target machine. Then, we visualize the trace to generate clean cuts of the representative regions or phases from the temporal structure of the execution. A main phase suggests regions of computation and/or communication that may show different behaviors or that are independent between them, e.g. separating long computational regions from communication intensive phases. In this step, we can measure duration of the computation, or number of MPI calls to highlight additional details in the interval that may suggest additional phases (other regions that may become interesting to analyze). To reduce potential noise introduced by the machine during the execution, the output of this step is one or more cuts of the cleanest regions of the trace for each core count. A region may contain one or more iterations, and it represents a part with potential performance bottlenecks or that may have significant impact to the execution.

1.2. Phase Performance Analysis

To report a quantitative summary of the performance of identified phases, all the trace cuts will be processed using our automatic framework. The model decomposes the parallel efficiency metric as a product of factors with normalized values between 0 (very bad) and 1 (perfect) [7]. The factors correspond to fundamental behavioral aspects of parallel codes and are load balance, serialization and transfer.

- **Parallel Efficiency** reflects the performance obtained from executing in parallel the code. It is expressed as the product of the three fundamental factors: Load Balance, Serialization and Transfer. Efficiency of the parallelization is presented in expression 1.

$$\eta_{||} = LB * Ser * Trf \tag{1}$$

- **Load Balance efficiency** reflects the potential efficiency loss caused by imbalance in the total computation time by each process. It is measured as the ratio between the average computing time ($\sum_i t_i/P$) and the maximum computation time ($max(t_i)$) from all the processes $i = \{0, ..., P\}$, as shown in expression 2.

$$LB = \frac{\sum_i t_i}{P * max(t_i)} \tag{2}$$

- **Serialization efficiency** reflects the inefficiency caused by dependencies in the code. It is measured by simulating an instantaneous communications (ideal) scenario using Dimemas and collecting the maximum efficiency achieved by a single process ($ideal(eff_i)$ in expression 3).

$$Ser = Max(ideal(eff_i)) \tag{3}$$

- **Transfer efficiency** reflects the performance loss caused by actual data transfer. It is caused by the MPI overhead plus the interconnection network noise and can be measured in expression 4, as the maximum efficiency achieved by a single process in the real execution ($Comm_{eff}$) and the inefficiency from serialization (Ser).

$$Trf = \frac{Comm_{eff}}{Ser} \tag{4}$$

In this step, we detail the performance of the application based on observations from collected measurements. The model provides a general view of the inefficiencies in the code and the relative importance of the performance factors. As a first advantage of our proposal, this type of model provides useful information to suggest the appropriate strategy to improve the performance in the application. From a broader point of view, the use of fundamental factors facilitates a unified modeling approach for strong and weak scaling scenarios, thus providing a fast and approximate approach to infer the evolution of the parallel code. Expressing performance in terms of parallel efficiency brings flexibility when selecting the number of iterations for each cut. In a steady iterative parallel code, a few iterations will typically be sufficient to represent its behavior. Reporting efficiency instead of absolute time makes the metric essentially insensitive to the number of iterations, eliminating the need to ensure that exactly the same number of iterations is analyzed for all core counts. The analysis tools (Paraver) provide mechanisms (for example analyzing histograms of the duration of computation, the cycles counter, among others) to identify regions of the traces that may be significantly perturbed by noise. Sufficiently clean cuts of the traces can thus be obtained at different core counts without the constraint of requiring them to be of exactly the same number of iterations. The advantage of this approach is that the analysis can be focused on regions with less perturbations.

1.3. Scalability Prediction

To infer the behavior of phases of an application in a specific platform at large core counts, our extrapolation approach fits the components of the multiplicative model presented in expression 1. The model is fed with data collected from traces obtained from several, not many runs using low core counts in the same target platform. We argue that the results of the analysis phase using a small number of executions for low core counts, combined with the fundamental underlying system behavior can help us identify the specific scalability model for each component of the multiplicative model. By independently extrapolating the individual components of this model we can observe their evolution; how relevant are they to the overall performance of the parallel code; potential variations of significant factor as core counts increase, and infer a potential performance for a core count that has not being executed before.

The original prediction strategy proposed in [7] observed that an extrapolation based on model factors led to more accurate predictions that extrapolate the overall efficiency. Nevertheless, these observations were based on projections for limited increases in the core counts. We aim at studying the impacts for much larger core counts. In an intent to use linear models, results where unacceptable because they get negative with large values of cores.

The basic default model is Amdahl's law formulation. This is a general and approximate model that represents a first approach to describe the effect of non-parallel regions, where inefficiencies are caused by an activity that can not be executed concurrently with other activities. This may be caused by computations being serial, but can also constitute an abstract model of other serialization behavior such as contentions in the network resources. The formulation of such an Amdahl based approach in terms of the efficiency model is presented in expression 5.

$$Amdahl_{fit} = \frac{amdahl_0}{f_{amdahl} + (1 - f_{amdahl}) * P} \tag{5}$$

Other possible pattern of concurrency corresponds to pipelined computation. Expression 6 models a behavior of alternating segments of totally parallel computation with perfectly pipelined segments.

$$Pipe_{fit} = \frac{pipe_0 * P}{(1 - f_{pipe}) + f_{pipe} * (2 * P - 1)} \tag{6}$$

Moreover, additional features of the program may suggest a different fitting for each factor, e.g. a constant behavior when there are no changes in efficiency when scaling may indicate that a factor can be treated as a constant value at very large scale.

2. Experimental Evaluation

For the experimentation, we use three applications from the CORAL suite: HACC, Nekbone, and AMG2013; and the CFD application AVBP. The Hardware Accelerated Cosmology Code (HACC) [8] parallel benchmark is a flexible framework that uses N-body techniques to simulate the evolution of the Universe from its early times to today and to advance our understanding of dark energy and dark matter. The Nekbone is a proxy-app from the Nek5000 software [9], which executes computationally intense linear solvers. Nekbone has been created to be easily adapted to different platforms, communication structures, and scalability studies. AMG2013 [10] is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. AMG2013 is a highly synchronous code and parallelism is achieved by domain decomposition. Parallel efficiency is largely determined by the size of data chunks in the decomposition, and the speed of communications and computations on the machine. AVBP [11] solves the three-dimensional compressible Navier-Stokes on unstructured and hybrid grids. AVBP includes integrated parallel domain partition and data reordering tools. The scaling: weak or strong, and the number of processes used in the runs of each application are summarized in tab. 1.

CORAL applications have been executed in MareNostrum III, a machine based on Intel Xeon E5 processors, iDataPlex Compute Racks and Infiniband network. Traces from AVBP have been obtained in Juropa, a supercomputer based on Intel Xeon X5570 processors, Sun constellation systems and Infiniband network. The machines operated in normal production using fully populated nodes, i.e. there is non dedicated network and some potential noise from OS may be introduced in the runs.

2.1. Identify Structure

To identify the structure of the application we visualized the traces for each execution with various ranks. A manual process that leads to obtain clean cuts of interesting phases within the execution. In general, applications present an iterative structure, and a clean cut is a region with low or none perturbations from the system. The identification process is based on the observation of metrics of the execution to limit the phases. Metrics as the duration of computational burst, cycles per microseconds, or the behavior of MPI calls (colored areas in the images shown below, where each color represents a type of MPI call) result very useful in this task.

In a trace of one iteration of HACC we can observe a large computationally intensive phase of $\approx 250s$. This phase, presented at the left side of fig. 1, shows low communications and some

Table 1. Applications used in for the experiments

Strong Scaling	Ranks	Weak Scaling	Ranks
HACC	16, 32, 64, 128, 256, 512, 1024, 2048, 4096	AMG2013	32, 64, 96, 128, 192, 256, 384
Nekbone	2, 4, 8, 16, 32, 64, 128, 256, 512	AVBP	16, 32, 64, 96, 128,192, 256, 520, 768, 1024,1040, 1280, 1536

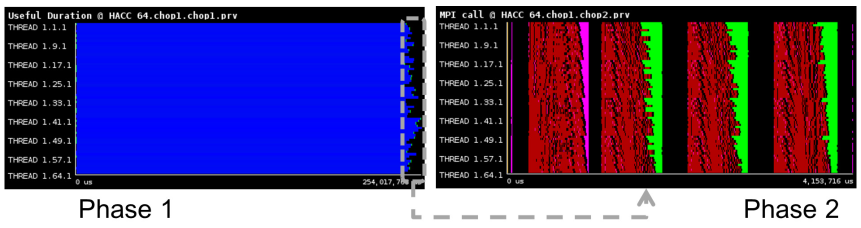
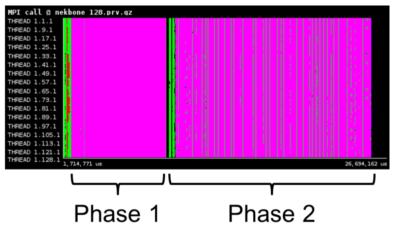
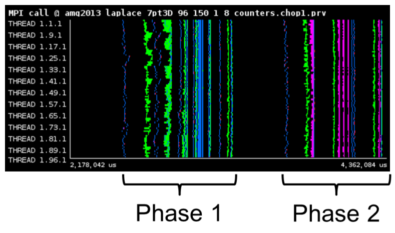

Useful Duration @ HACC 64.chop1.chop1.prv
MPI call @ HACC 64.chop1.chop2.prv
Phase 1
Phase 2

Figure 1. Phases in one iteration of HACC code




MPI call @ amg2013 laplace 7pt3D 96 150 1 8 counters.chop1.prv
Phase 1
Phase 2

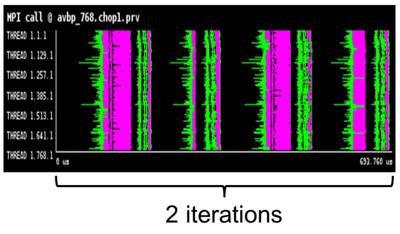


Figure 2. Phases in Nekbone **Figure 3.** Phases in AMG2013 **Figure 4.** Iteration in AVBP

imbalances at the end. The iteration has also a small communication phase ($\approx 4s$) shown at the right side of the figure. Inside this phase, we could identify sub phases of finer grain, yet for the scope of our analysis the main two phases are enough.

In Nekbone we identify two phases with an iterative structure, presented in fig. 2. To avoid the impact of potential noise of the system, we choose iterations with less variations between the bursts. At the same time, the number of iterations chosen may variate when increasing the number of core counts. A decision that does not affect the effectiveness of our method as it measures the parallel efficiency of the total region.

For AMG2013, we focus our analyses in the regions with greater presence of MPI calls, differentiating one phase dominated by point-to-point communications from a second phase that presents more collectives calls. These phases, shown in fig. 3, are consecutive within one iteration of the application.

Finally, we took two outer iterations of AVBP to test the sensitivity of our method using a coarser grain. In the cut, there are regions of compute with some imbalances (dark areas in fig. 4), which are followed by point-to-point communications and a subsequent synchronization step before starting new computations.

2.2. Phase Performance Analysis

We apply the automatic framework for basic analysis to each phase identified to extract the main performance metrics and factors of the performance model. Below we summarize the outputs of the efficiency associated to the fundamental factors for all the applications. Values range from 0 to 1 being those closer to 1 which present greater efficiency.

For phases 1 and 2 of HACC, we can observe the evolution of the performance factors when changing the number of ranks. In this parallel code, the evolution of all three factors in the computational phase (fig. 5) describes a highly efficient region with almost no imbalance or contention. In the analysis of the communication phase (fig. 6), Transfer (Trf) followed by Serialization (Ser) are dominant factors in the overall performance loss. After 256 ranks, Transfer reports an efficiency of 0.6 and Serialization reports around 0.8, thus suggesting the presence of some contention or delay in the communications between processes. The Load Balance in fig. 6 seems to remain steady for all number of ranks. To better understand the fundamental behavior of this metric, we compute the load balance across processes in terms of total number of instructions they execute rather than the time they take. The result is perfect load balancing across all the core counts. In addition, after analyzing the trace, processors seem to perform computations in stages inside phase 2 (similar to a pipeline decomposition), in consequence, we decide to model the serialization factor by expression 6.

The analysis from the two phases of Nekbone reports regions with an efficiency of above 0.8 (fig. 7 and fig. 8). Load balance shows a slight drop after 32 processes that later seems to stabilize. An analysis of the cycles per second metric in the traces confirms the existence of a small level of preemption, thus suggesting that the reduction observed in this factor derives from noise in the system.

The analysis from the phases of AMG2013, shown fig. 9 and fig. 10, suggest a performance loss mainly dominated by load imbalances. Both phases present this behavior and by checking in the trace files, we verified that this is due to preemptions introduced by noise. In addition, for larger core counts transfer efficiency in phase 1 shows greater degradations. After analyzing the traces, the inefficiencies are caused by contention in point-to-point communications.

The AVBP executions reported values with an initial steady behavior up to 384 processes (left side in fig. 11). When increasing the number of cores abrupt variations appear. The trace files reported that these variations are caused by unexpected noise in the run, further discussion of this subject is in section 2.4.

2.3. Scalability prediction

2.3.1. Model Fitting

From the collected measurements and performance observed from the analysis of the traces, we now want to generate a model to infer the expected performance at large core counts. Contention can be one of the main causes of performance loss, and Amdahl's Law reflects the presence of this type of restrictions when increasing core counts. In consequence, we propose to fit all the fundamental factors using the expression 5 by default. Metrics were fitted using the least square regression model taking as a reference current measurements from executions with low core counts. The fit considers the dependent variable to be inside the range $[0.0, 1.0]$. In the results shown below, we use expression 5 to infer the expected parallel efficiency from all the fundamentals factors for Nekbone, AVBP, AMG2013, and for Transfer and Load Balance in HACC. For almost all the applications, Amdahl's function generates a curve that follows the closest gradient to the measured points (presented from fig. 12 to fig. 17). In addition, it provides a reasonable fitting with small influence of the number of points used.

2.3.2. Validation of Results

In this section, we compare the efficiencies predicted for a machine from runs using small core counts with the values collected from real traces using at least runs 3 times larger than the number of processes used for prediction. In the available machines, we have obtained traces up to 1536 processes (AVBP in Juropa) and 4096 processes (HACC in MareNostrum III). Predicted efficiencies and their associated relative error (inside parentheses) are summarized in the tables below. Prediction results are from the phase of the application that shows the bigger relative

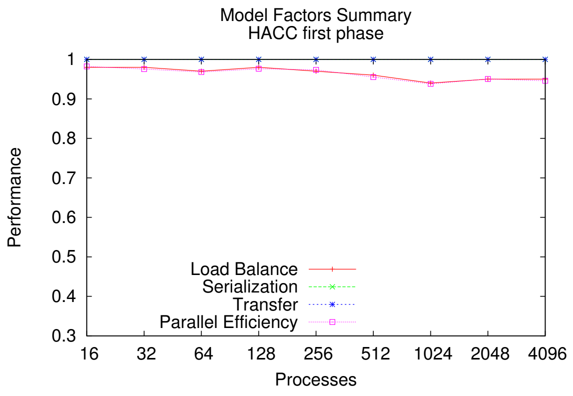


Figure 5. Model factors phase 1 HACC

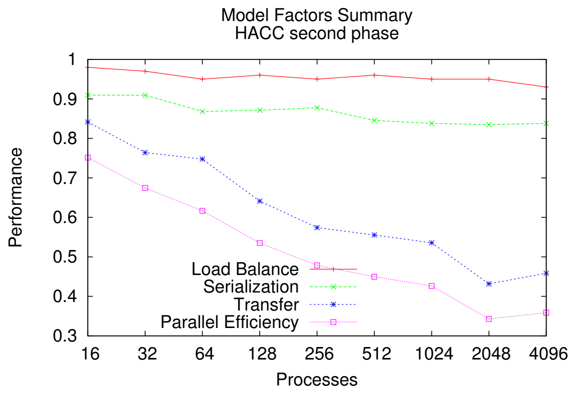


Figure 6. Model factors phase 2 HACC

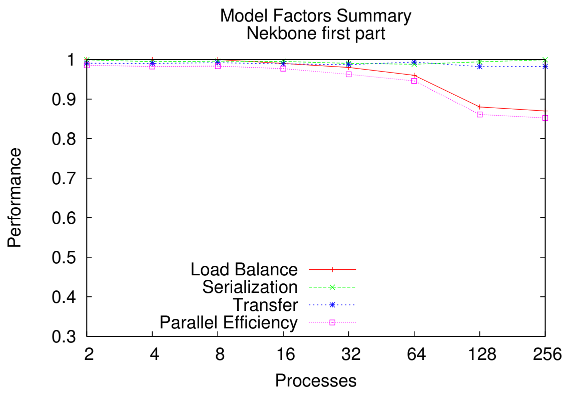


Figure 7. Model factors first part Nekbone

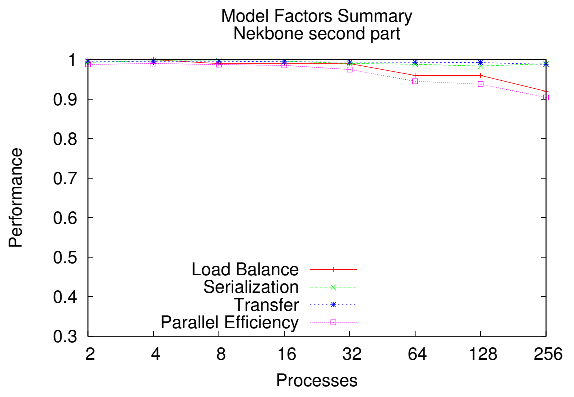


Figure 8. Model factors second part Nekbone

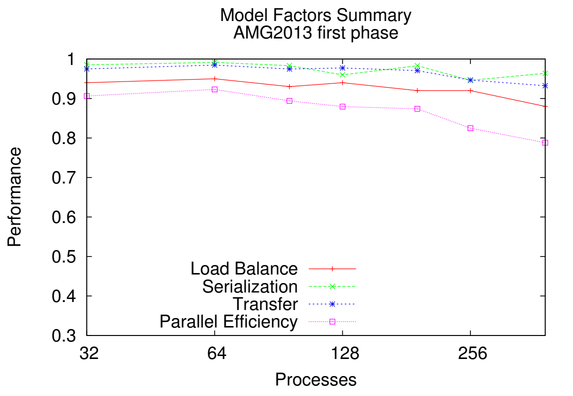


Figure 9. Model factors phase 1 AMG2013

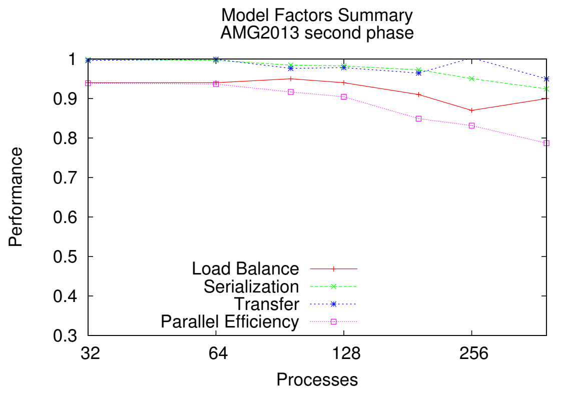


Figure 10. Model factors phase 2 AMG2013

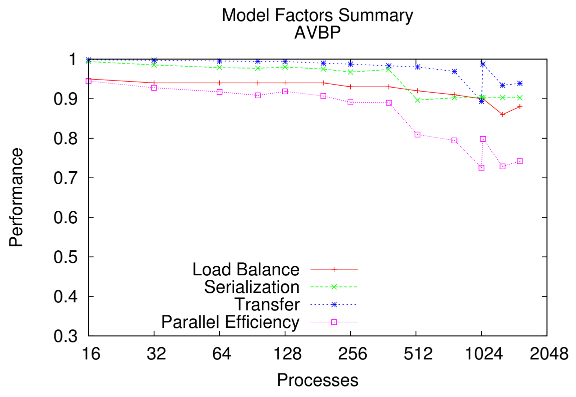


Figure 11. Model factors from two iterations of AVBP

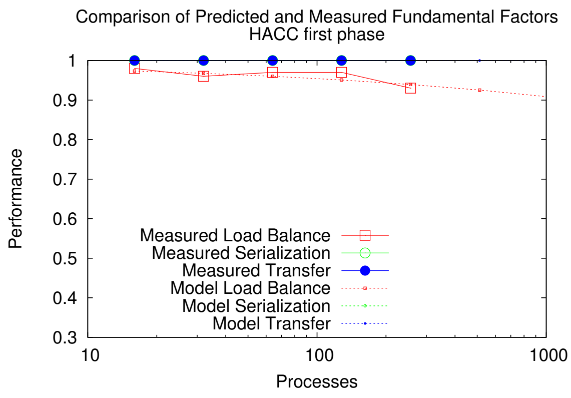


Figure 12. Fitting phase 1 HACC

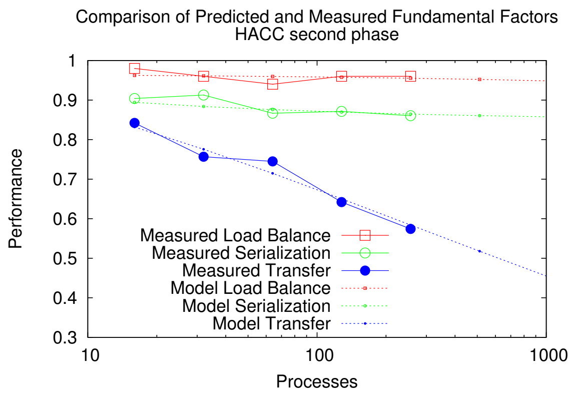


Figure 13. Fitting phase 2 HACC

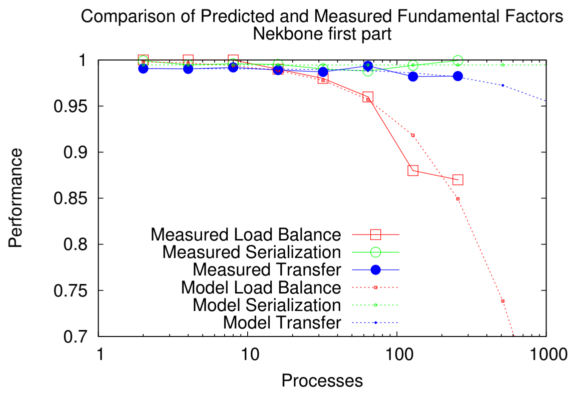


Figure 14. Fitting first part Nekbone

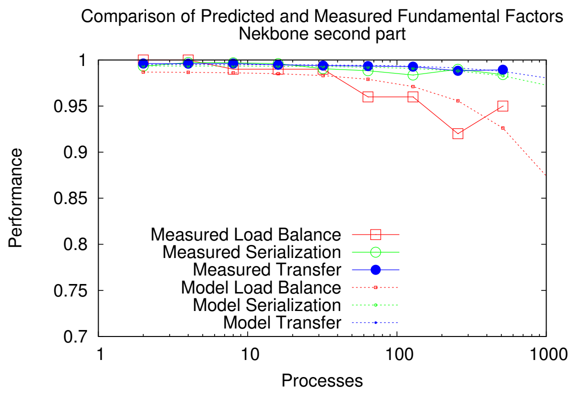


Figure 15. Fitting second part Nekbone

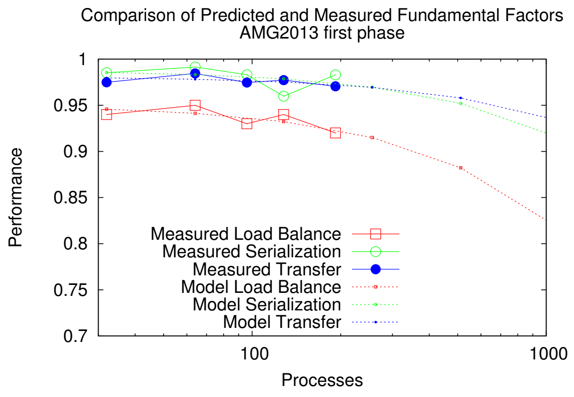


Figure 16. Fitting phase 1 AMG2013

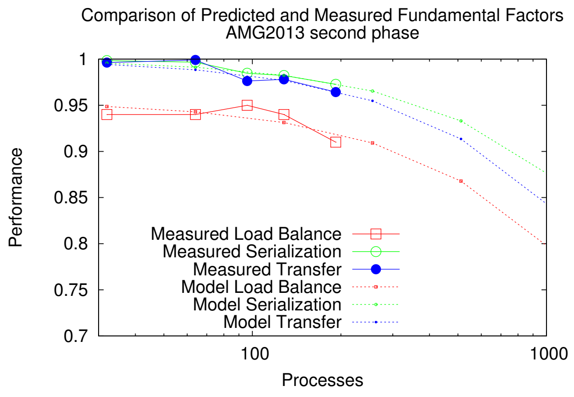


Figure 17. Fitting phase 2 AMG2013

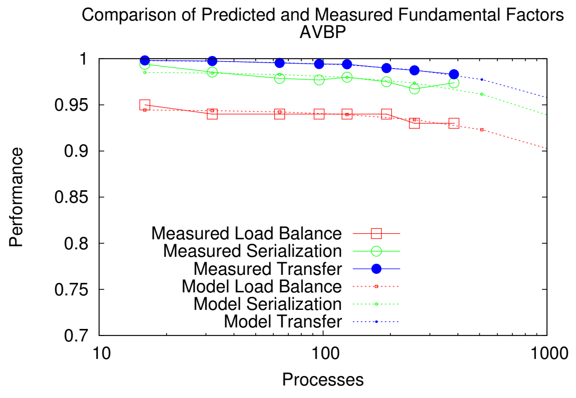


Figure 18. Fitting from AVBP

error. The error for each fundamental factor shows how optimistic (+) or pessimistic (-) is our prediction.

For HACC, phase 2 with up to 256 processes were used to project the efficiency for 512, 1024, 2048 and 4096. Results shown in tab. 2, show a low relative error in the predictions of load balance and serialization. Even for transfer, which reports variable errors, we are able to predict the expected efficiency for a number of processes 20 times larger loosing around a 25% of accuracy. With Nekbone, we use traces up to 128 processes to project the efficiency of 256 and 512. Efficiencies reported in tab. 3 show a relative error of less than a 0.1%. We used traces up to 128 processes to project the efficiency for AMG2013. Results for predicted efficiencies for 256 and 384 processes are reported in tab. 4. These results show a relative error of less than 10%. For AVBP, we used executions with up to 384 processes to project the expected efficiency for 520, 768, 1024, and 1536 presented in tab. 5, and the differences between the expected and the real measurement are not greater than 18%.

2.3.3. Projection for large core counts

From collected efficiencies of each fundamental factor our framework extrapolates the expected total parallel efficiency for up to 10^6 cores. In weak scaling, phase 1 of HACC shows in fig. 19 a constant behavior of Serialization and Transfer, and a low degradation of Load Bal-

Table 2. Predicted efficiency and relative error for HACC (phase 2), extrapolated from runs using 16 to 256 cores

Ranks	Load Balance	Serialization	Transfer	Parallel Efficiency
512	0.952(−0.82)%	0.860(+1.81)%	0.517(−6.78)%	0.424(−5.61)%
1024	0.948(−0.17)%	0.857(+2.34)%	0.452(−15.47)%	0.368(−13.64)%
2048	0.943(−0.68)%	0.855(+2.45)%	0.391(−9.39)%	0.315(−7.80)%
4096	0.937(+0.82)%	0.853(+1.83)%	0.333(−27.19)%	0.267(−25.25)%

Table 3. Predicted efficiency and relative error for Nekbone (phase 2), extrapolated from runs using 2 to 128 cores

Ranks	Load Balance	Serialization	Transfer	Parallel Efficiency
256	0.972(−0.003)%	0.985(+0.002)%	0.992(−0.001)%	0.950(−0.002)%
512	0.950(−0.007)%	0.977(+0.004)%	0.989(+0.001)%	0.919(−0.001)%

Table 4. Predicted efficiency and relative error for AMG2013 (phase 2), extrapolated from runs using 32 to 192 cores

Ranks	Load Balance	Serialization	Transfer	Parallel Efficiency
256	0.921(−0.16)%	0.937(+0.95)%	0.941(+0.55)%	0.813(+1.45)%
384	0.908(−3.08)%	0.908(+6.05)%	0.914(+1.94)%	0.754(+4.42)%

Table 5. Predicted efficiency and relative error for AVBP, extrapolated from runs using 16 to 256 cores

Ranks	Load Balance	Serialization	Transfer	Parallel Efficiency
520	0.907(−1.42)%	0.925(+3.23)%	0.973(−0.70)%	0.816(+0.93)%
768	0.886(−2.54)%	0.893(−1.00)%	0.961(−0.82)%	0.761(−4.15)%
1024	0.867(−3.66)%	0.862(−4.40)%	0.948(+6.21)%	0.709(−2.16)%
1536	0.830(−5.68)%	0.807(−10.54)%	0.925(−1.45)%	0.620(−16.46)%

ance, thus resulting in a region that will scale relatively well at very large core counts. For phase 2 (fig. 20), at 2k cores the parallel efficiency is below 0.4, thus suggesting that the problem will be the communication contention, while a pipelined serialization structure may represent a secondary but less relevant cause of inefficiency.

Phase 1 of Nekbone shows a total parallel efficiency equal to 0.4 at 2k cores (fig. 21). An efficiency loss mainly dominated by load imbalances. Although this suggest uneven distribution of the work, the histogram of instructions reveals that there is no computational imbalance. The imbalance appears because of IPC differences between process. This behavior would need deeper analysis. At the same point, parallel efficiency for phase 2 is also dominated by load imbalances (fig. 22), however, this phase shows a reasonable efficiency (0.7) for 2k processes and reaches the efficiency of 0.4 at a larger number of cores than phase 1 (around 8k).

In strong scaling, AMG2013 shows similar expected behaviors for both phases (fig. 23 and fig. 24). Phase 1 is mainly dominated by load imbalances, and shows a parallel efficiency of 0.5 at 2k cores. On the contrary, in phase 2 due to the coupling between all the fundamental factors, a lower efficiency (0.4) for the same number of cores is reported.

Finally, the extrapolation of parallel efficiency for AVBP shows a performance loss tightly coupled to all 3 factors. At 2k cores the total efficiency predicted is 0.7, and it is worth mentioning that the problem size provided by the developers of this application was expected to scale well up to 1k cores. In addition, for AVBP, and unlike the rest of the applications, there is a shift between load balance and serialization as the influential factor in loss of efficiency after 10k (fig. 25). From this observation, the expected dominant factor for a given number of cores may not result the same when scaling the application for larger core counts. It suggests using different optimization techniques depending on the scale at which the application will be executed.

2.4. Additional enhancements: Noise reduction

Predictions until now were based on runs of an application in current available machines. The noise introduced by the system or by the MPI engine is extrapolated and we can thus infer how the application will behave when scaling the platform. Some interesting questions emerge: Which is the impact that some characteristics of the system (i.e. noise) may have in the final prediction?. What would be the behavior if those aspects are improved or worsened?. What is the influence of noise of the machine, of interconnection network noise, of the bandwidth of the system, of disturbances introduced by the progression engine of MPI, among others. In this section, we demonstrate how our approach and infrastructure can be used to address above questions. We will demonstrate it analyzing the impact of noise in the Juropa platform used for the AVBP runs.

2.4.1. Impact of noise in communication

Recall that our model uses a simulation with no latencies (ideal) to collect the values of Serialization and the resulting values are used to calculate Transfer. By using an ideal scenario potential disturbances within MPI in the original trace are cleaned, and inefficiencies are not assigned to Serialization but to Transfer. This factor can be read as the cost of data movement in the machine where the traces have been obtained, but it also captures other inefficiencies, such as: network noise, issues with the MPI progression engine of preemptions inside MPI calls. This effect was detected on AVBP where there is a significant difference between the behavior

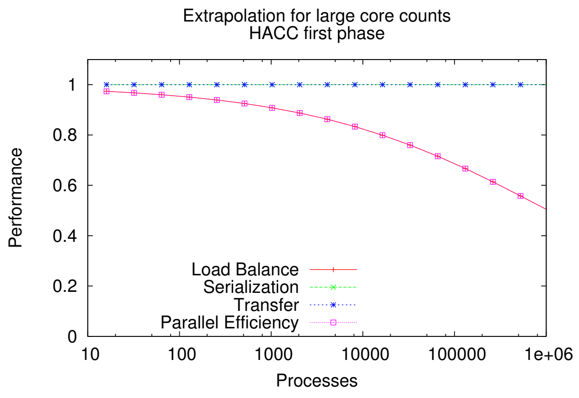


Figure 19. Extrapolation phase 1 HACC

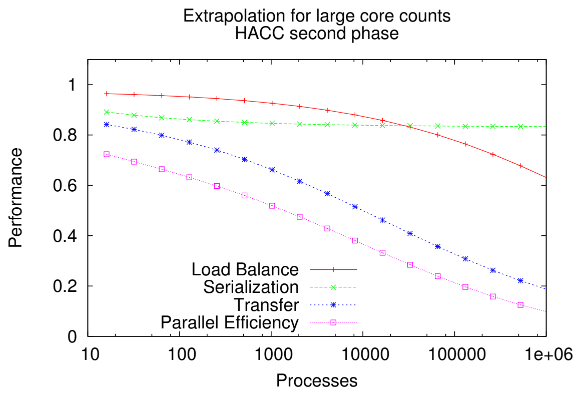


Figure 20. Extrapolation phase 2 HACC

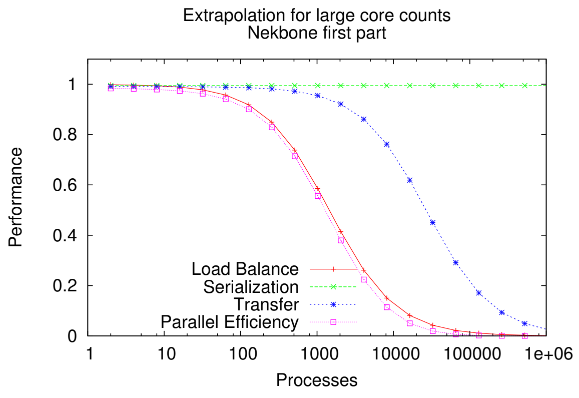


Figure 21. Extrapolation first part Nekbone

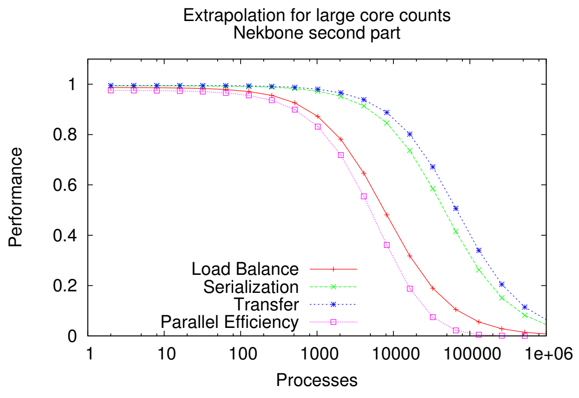


Figure 22. Extrapolation second part Nekbone

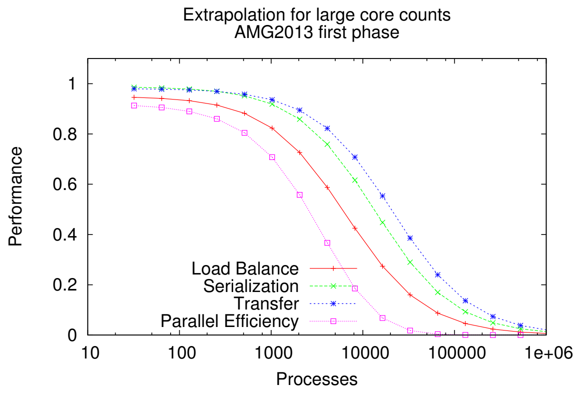


Figure 23. Extrapolation phase 1 AMG2013

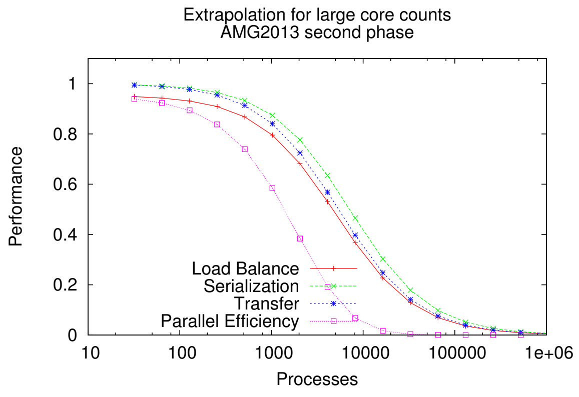


Figure 24. Extrapolation phase 2 AMG2013

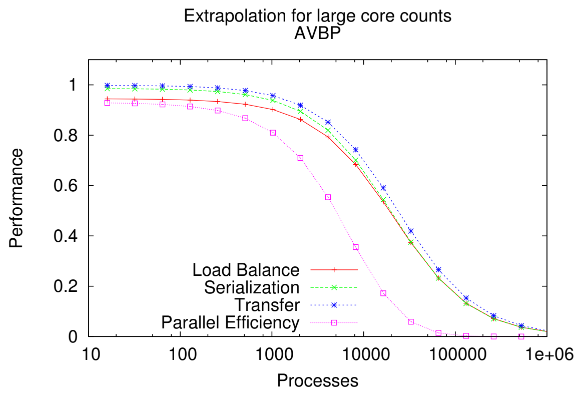


Figure 25. Extrapolation for large core counts for AVBP

of the 1024 and 1040 runs. Looking at the traces we detected problems on the MPI progression engine that can be seen on fig. 26 and affects the exit time of some collectives (Allreduce in this case).

Our apporach to eliminate the effect of such perturbations within MPI in the final results starts by generating a Dimemas trace from the original Paraver traces. The Dimemas traces capture the computation demands from the computation records in the original paraver trace. A Dimemas simulation with nominal values for the target machine rebuilds the precise time behavior on a platform without noise in the communication. The methodology presented in this paper can be applied to the paraver traces resulting from the simulation to extrapolate the efficiency factors of the application if the network/MPI noise is eliminated. The result of projecting the factors from traces of AVBP without disturbances in communication is shown in fig. 27. We can see how the transfer factor has now better scalability. Even if the network noise had an important impact in the transfer efficiency, the overall applications is still dominated by load balance and serialization and thus global performance does not significantly improves eliminating such noise.

2.4.2. Impact of noise in computation

Noise may also affect the computational phases. To eliminate noise from these phases, the original trace is translated to Dimemas format using the available cycles counter to determine the duration of the computation burst. As hardware counters are virtualized they do not count while the process is preempted. By knowing the frequency of the processor and the number of actual cycles, a very precise computation of the non perturbed duration of computation burst can be obtained.

Repeating the process described in the previous section with the new conversion mechanism we obtain the extrapolation results reported in fig. 28. Now the load balance prediction scales better, thus suggesting that part of the identified unbalance was not originated by the application but by the system noise. By eliminating the noise from the machine and from communications, the serialization is now the dominant factor in efficiency loss.

From this additional study, we conclude that when predicting scalability of parallel applications we must be aware that current machines and interfaces are introducing variations (noise) in the executions. The design of mechanisms to include these noise factors into the prediction model may result a necessity.

3. Related Work

The approach based on first principles of [4] requires a deep knowledge of the application algorithms and parallelization structure to build analytical models directly from such knowledge. As it may result costly in terms of the deep application knowledge required, is computationally inexpensive and informative, having proved useful to actually identify problems in machines that did not get the predicted performance.

The work presented by [12], uses analytical application models to derive the performance of NAS BT benchmark in future systems. The approach builds a precise model of computation, memory usage, and communication to evaluate the potential paths to scale an application. The analysis provides insights of potential bottlenecks but does not formalizes an strategy to predict the expected performance.

A multiscale simulation based methodology is introduced by [13]. This work models the cluster level of parallelism by means of Dimemas, a simulator of a distributed memory target machine, parametrized by network and overall sequential performance parameters. The application characteristics are captured in a trace of a real run with the desired number of processes on an existing machine. The sequential performance extrapolations fed into Dimemas as part of the multiscale approach are computed using instruction level simulators, thus a costly process.

In an effort to investigate the performance of Message Passing Interface applications at large core counts, parallel discrete event simulations have been used to run the application in a controlled environment [3] and observe its behavior. The work of [14] and [15], analyzed the impact of communications when scaling to a larger number of processors, and the expected degradation in the network bandwidth, respectively. These works have been focused on the communication interface and in the hardware rather than in the basic behavior of the applications using them.

The work of [6] proposes to blindly fit metrics (essentially time) measured for the main routines of a program when running at low core counts on an existing platform. A large number of fitting functions is tried and the one reporting the best fit is selected as model of that routine. The approach is useful to identify trends and point to routines that will become bottlenecks at larger core counts in the same platform. The method, although it is simple and does not require excessive calculations, offers limited insight about influential factors in the performance loss.

A methodology to extrapolate the computational behavior of large-scale HPC applications has been presented in [16]. Their method extrapolates application traces as a relevant technique to understand how an application scales on a particular system, and can be useful to detect the impact of incremental or major changes in the hardware being used to run the application.

Several efforts to evaluate scalability of parallel applications has been made in [17]. They present a performance model for an specific phase of the AMG application, exposing existent bottlenecks and predicting the expected scalability in future machines based on their analytical model for computation and communication.

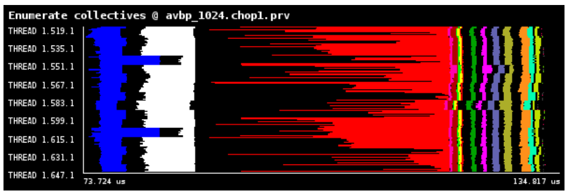


Figure 26. Delay in collectives for real trace of AVBP

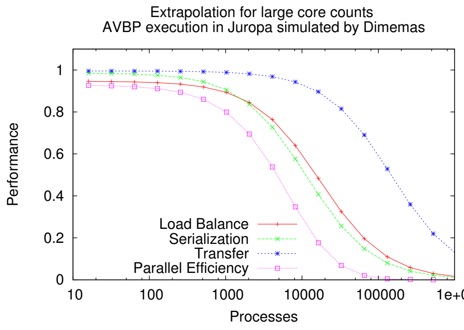


Figure 27. Extrapolation of parallel efficiency obtained from simulations of AVBP in Juropa (time)

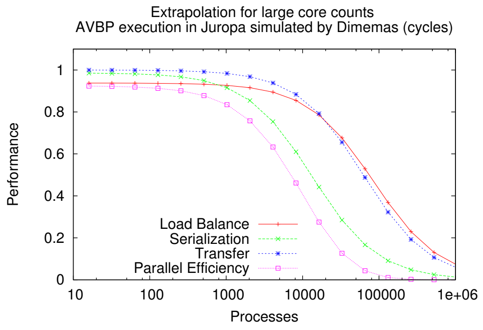


Figure 28. Extrapolation of parallel efficiency obtained from simulations of AVBP in Juropa (cycles)

4. Conclusions

In this paper, we described a methodology to collect primary components of current parallel codes and infer their expected behavior when scaled to larger core counts. To extrapolate the expected parallel efficiency, the approach extracted basic knowledge from traces obtained from runs using a low number of processes. Traces used in this work were obtained in two different machines that are currently in production (MareNostrum III and Juropa), and the process of analysis and estimation was performed by means of available performance tools and a semi-automatic framework. Our framework collected from the traces three fundamental components of parallel efficiency: load balance, serialization and transfer. Then, a first general model based on Amdahl's law was used to infer the evolution of each factor for large scale executions. We evaluated the method using 3 applications from the CORAL suite (HACC, Nekbone and AMG2013) and a CFD application AVBP. Predictions of expected efficiencies based on executions at low core counts showed a low relative error. Scalability projections showed interesting behaviors for strong and weak scaling, such as applications mainly dominated by load imbalances, efficiency loss caused by coupling of factors, among others. In general, from obtained results our method provides an inexpensive and useful tool to quickly infer the expected scalability of parallel codes.

As further steps, the method can be easily refined by including additional extrapolation models to fit different behaviors for new parallel codes. Similarly, to complement the current model with the potential effect of noise introduced by the machine or inherent noise of running a parallel code remains as a future work. The methodology can also be enriched by knowledge obtained from simulations of the parallel code on different architectures, thus providing additional insights on how the code may evolve in different platforms.

This work has been partially supported by the Intel-BSC Exascale Lab and the DEEP Project.

References

1. A. Geist and R. Lucas, "Major Computer Science Challenges At Exascale," *Int. J. of High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 427–436, 2009.

2. J. Åström, A. Carter, J. Hetherington, K. Ioakimidis, E. Lindahl, G. Mozdzynski, R. Nash, P. Schlatter, A. Signell, and J. Westerholm, "Preparing Scientific Application Software for Exascale Computing," in *Applied Parallel and Scientific Computing*, vol. 7782, pp. 27–42, Springer Berlin Heidelberg, 2013.

3. Christian Engelmann, "Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale," *Future Generation Computer Systems*, vol. 30, no. 0, pp. 59–65, 2014.

4. K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, and J. Sancho, "Using Performance Modeling to Design Large-Scale Systems," *Computer*, vol. 42, pp. 42–49, Nov 2009.

5. P. Balaprakash, D. Buntinas, A. Chan, A. Guha, R. Gupta, S. H. K. Narayanan, A. A. Chien, P. Hovland, and B. Norris, "Exascale Workload Characterization and Architecture Implications," in *Proc. of the High Perf. Computing Symposium*, pp. 5:1–5:8, 2013.

6. A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes," in *Proc. Intl. Conf. for High Perf. Computing, Networking, Storage and Analysis*, SC '13, pp. 45:1–45:12, 2013.

7. M. Casas, R. M. Badia, and J. Labarta, "Automatic Analysis of Speedup of MPI Applications," in *Proc. 22nd Intl. Conf. on Supercomputing*, pp. 349–358, 2008.

8. S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel, N. Frontiere, and Z. Lukić, "The Universe at Extreme Scale: Multi-petaflop Sky Simulation on the BG/Q," in *Proc. Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '12, pp. 4:1–4:11, 2012.

9. Center for Exascale Simulation of Advanced Reactors, "Proxy-Apps for Thermal Hydraulics." https://cesar.mcs.anl.gov/content/software/thermal_hydraulics.

10. Van E. Henson and Ulrike M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Appl. Numer. Math.*, vol. 41, no. 1, pp. 155–177, 2002.

11. N. Gourdain, L. Gicquel, M. Montagnac, O. Vermorel, M. Gazaix, G. Staffelbach, M. Garcia, J.-F. Boussuge, and T. Poinsot, "High performance parallel computing of flows in complex geometries: I. Methods," *Comput. Sci. Disc*, vol. 2, no. 1, p. 015003, 2009.

12. R. van der Wijngaart, S. Sridharan, and V. Lee, "Extending the BT NAS Parallel Benchmark to exascale computing," in *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '12, pp. 1–9, 2012.

13. J. Gonzalez, M. Casas, J. Gimenez, M. Moreto, A. Ramirez, J. Labarta, and M. Valero, "Simulating Whole Supercomputer Applications," *IEEE Micro*, vol. 31, no. 3, pp. 32–45, 2011.

14. X. Wu and F. Mueller, "ScalaExtrap: Trace-based Communication Extrapolation for Spmd Programs," in *Proc. 16th ACM Symposium on Principles and Practice of Parallel Prog.*, pp. 113–122, 2011.

15. S. Dosanjh, R. Barrett, D. Doerfler, S. Hammond, K. Hemmert, M. Heroux, P. Lin, K. Pedretti, A. Rodrigues, T. Trucano, and J. Luitjens, "Exascale design space exploration and co-design," *Future Generation Computer Systems*, vol. 30, no. 0, pp. 46–58, 2014.

16. L. Carrington, M. A. Laurenzano, and A. Tiwari, "Inferring Large-Scale Computation Behavior via Trace Extrapolation," in *Proc. IEEE 27th Intl Symp. on Par. & Dist. Proc. Workshops and PhD Forum*, pp. 1667–1674, 2013.

17. H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms," in *Proc. Intl. Conf. on Supercomputing*, pp. 172–181, 2011.