Parallel Application Signature for Performance Analysis and Prediction

Alvaro Wong, Dolores Rexachs, and Emilio Luque

Abstract—Predicting the performance of parallel scientific applications is becoming increasingly complex. Our goal was to characterize the behavior of message-passing applications on different target machines. To achieve this goal, we developed a method called parallel application signature for performance prediction (PAS2P), which strives to describe an application based on its behavior. Based on the application's message-passing activity, we identified and extracted representative phases, with which we created a parallel application signature that enabled us to predict the application's performance. We experimented with using different scientific applications on different clusters. We were able to predict execution times with an average accuracy greater than 97 percent.

Index Terms—Parallel application, performance prediction, application signature

1 Introduction

To measure the performance of a parallel machine, researchers have often used a set of application kernels as benchmarks. However, it is not always possible to characterize the performance using only benchmarks [1], as each benchmark usually reflects a narrow set of kernel applications at best. Computers exhibit different performance indices associated with applications as they run them. Accurately predicting the performance of parallel applications is becoming increasingly complex, but the time required to run the application thoroughly is an onerous requirement, especially if we want to predict the performances of different systems.

It is important to determine which system is more appropriate to execute a scientific algorithm and predict its execution time. Accurate performance estimations are thus instrumental in helping a system resource scheduler efficiently schedule user jobs. if the system resources administrators know the number of requested resources and how long the resources are requested using the signature, they can make an efficiently queued plan for the system.

We propose the extraction of valuable information about the performance characteristics of an application and lets us predict the performance of the application on parallel machines (clusters) and use this information without needing to run the full application. This performance characterization (the signature) will constitute the performance metadata of an application.

The system throughput is defined as the number of jobs completed per unit of time and is an important performance metric for users, who expect minimal response time. The signature can help the system administrators who are concerned with the overall resource utilization by knowing the execution time of the application due to the use of the signature. A job schedule can maximize the system throughput especially in high-throughput computing clusters.

To determine the performance of a parallel application, our methodology can help the user to compare using different structuring strategies, minimizing the communication delays, quickly and precisely. The signature execution time (SET) is shorter than the application execution time (AET), allowing the user (developer) to concentrate on the significant portions of the application (phase), which are the components of the signature.

We have created a methodology called parallel application signatures for performance prediction (PAS2P), that consists of two stages:

A) Application analysis and signature generation. PAS2P analyzes the application behavior with a specific data set (workload), in a previous work [2], we explain how we can predict the performance changing the workloads.

To characterize message-passing applications, PAS2P instruments and executes applications on a base machine and produces a trace log. The collected data are used to characterize computation and communication behavior. Fig. 1 shows an overview of our approach. To obtain the machine-independent application model, the traces are assigned timestamps from a global clock according to causality relationships between communication events, using an algorithm inspired by Lamport and Time [3]. We thus obtain a single logical trace of the complete distributed system. When we have the logical trace, it is important to identify the most relevant sequences (phases).

Our goal at this point is to gather these phases and assign them a weight; such phases will be selected to constitute the signature according to their weight (number of times they occur), and their execution time.

B) Performance prediction. When we have the signature, we can execute it on different systems. The signature measures the execution time of each phase and estimates the

• *The authors are with the Computer Architecture and Operating System Department, Universitat Autonoma de Barcelona, Barcelona, Spain. E-mail: alvaro@caos.uab.es, {dolores.rexachs, emilio.luque}@uab.es.*

Manuscript received 21 June 2013; revised 28 May 2014; accepted 29 May 2014. Date of publication 8 June 2014; date of current version 5 June 2015. Recommended for acceptance by B. de Supinski.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2329688

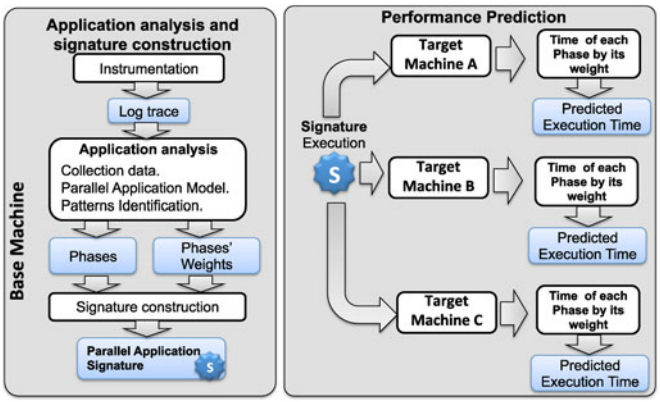


Fig. 1. PAS2P overview.

entire application run time on each system by aggregating all relevant phase execution times (PhaseET) using the obtained weights.

To evaluate the quality of the predicted performance, we conducted a set of experiments to extract signatures from different applications, including CG, BT, LU and SP from NPB [4], Sweep3D [5], Parallel Ocean Model (POP) [6], SMG2000 [7], GROMACS [8] and MD Moldy [9]. We ran the obtained signatures on three different clusters, using a different number of nodes (cores) and including different interconnection networks. We were able to predict the execution time with an average accuracy of more than 97 percent.

In the following section, we present related work. In Section 3, we describe the proposed methodology. In Section 4, we present the performance prediction model. Section 5 provides the experimental results and discusses the obtained prediction results, and Section 6 presents the performance of the PAS2P tool. In the last section, we present conclusions and future work.

2 Related Work

Gustafson [10] proposed a method that creates two profiles, a hardware signature and an application signature. For this method, information about the application workload, hardware characteristics and mapping becomes necessary. In our proposed method, these hardware characteristics are obtained when the signature is executed. Its execution time is smaller than the execution time of the entire application, the signature carries out a fast characterization of the machine.

Other studies have focused more on the creation of an application signature. Laura et al. [11] extracted the signature of an application using tools that allowed them to capture its profile, emphasizing memory access patterns. They ran these results on a network simulator to predict the parallel application performance. The difference between this approach and our approach is that our signature is the real code of the application; when we execute it on different parallel computers, real memory access patterns and the real computational resource requirements are used to evaluate the performance.

Sodhi et al. [12] and Wu et al. [13] claimed that it is possible to obtain a benchmark of an application using execution traces. They extracted information from the communications trace seeking similarity between those communications and creating mimic code to measure application performance. We propose the extraction of inter-process communication information from the trace and the attachment of computational timing. We used this method to create a signature by means of checkpoints, i.e., using application segments to predict application performance instead of creating mock-ups.

Girona et al. [14] is a performance predictor simulator for message-passing applications. It helps users develop and tune parallel applications on any machine while providing an accurate prediction of their performance on the target parallel machine. In our approach, we create a signature that represents the application; this signature can be executed on different systems quickly without needing a simulator.

The SimPoint tool [15] searches for phases in the behavior of parallel programs on shared-memory machines. Perelman et al. first focus on demonstrating the ability to identify similar intervals of execution across threads in a single run. Finally, phase analysis is used to select simulation points to guide multi-threaded simulation. Our technique creates a methodology to generalize the focus to a broader spectrum, i.e., all message-passing scientific applications, and to create a signature.

Bohrer et al. [16] is a simulation system used to model systems based on the PowerPC architecture. It provides construction blocks for the creation of simulators that range from functional to highly accurate but increases the simulation time. Conversely, what we propose is the creation of a signature of the parallel application that represents the same application and is executed on real machines, with the potential advantage of running it on different real systems quickly (as its run time is only a small fraction of the execution time of the entire application). Without a network simulator, it can also be executed in simulated systems under development.

Yang et al. [17] developed a methodology for predicting applications using "partial performance". They argued that it is enough to observe partial executions of a parallel application because codes are iterative and behave predictably after an algorithm initialization period. According to their methodology, a limited number of *time steps* are used to capture the performance of an application, which are maintained throughout the entire execution. Our signature intends to analyze the entire execution to provide better prediction quality.

Casas et al. [18], show an approach focused on the automatic detection of phases of an MPI application execution. This detection is based on Wavelet Analysis. They focus on scientific applications that are executed using thousands of processors, generating huge tracefiles and using visualization tools as Vampir [19]. In the same area, Noeth et al. [20] show a method to compress tracefiles while maintaining low overhead. A major problem for visualization tools is the size of the tracefiles. If we reduce the size of the tracefile to analyze, the tool will still have to visualize thousands of processes. The algorithms that detect phases are related to our work to reduce the tracefiles, but our goal is that the users can analyze their applications, extract the phases and construct the signature to predict the application execution time on target machines without requiring visualization tools.

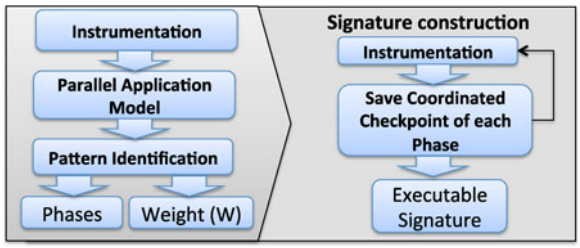


Fig. 2. PAS2P methodology.

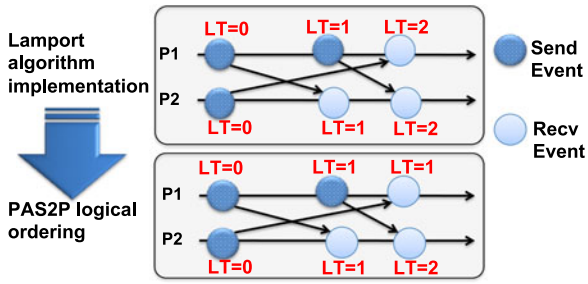


Fig. 3. Lamport to PAS2P ordering.

In this paper, we have refined the PAS2P stages of analysis and performance prediction. We extended the method to create the signature automatically, making it transparent to the user, allowing the creation and execution of the signature in parallel environments using any MPI library. We describe the stages and algorithm optimizations of the methodology centered on a single stage unlike the previous published papers. We developed an extensive experimental validation, since we were able to generate signatures by accessing a larger number of cores.

3 PAS2P Methodology

Applications typically possess highly repetitive behavior, and parallel applications are no exception [21], [22]. To characterize the computational-related and communication-related behavior of parallel applications, we identify these repetitive portions of an application. We use this information to create a signature that, when executed, allows the prediction of the execution time for the machine on which the signature was run.

As Fig. 2 shows, a sequence of stages is necessary to obtain the relevant portions (phases) of an application and their weights. With this information, we create a completely machine-independent signature for each application that can be executed on other systems in a shorter amount of time, as the signature execution time is a small fraction of the entire runtime of an application. Finally, we predict the full execution time of the parallel application by adding the execution time of all phases multiplied by their weights.

In this section, we describe each stage of the PAS2P methodology. Section 3.1, namely data collection, describes the application instrumentation. Section 3.2 describes the application model, which is a machine-independent model. Section 3.3 describes how we identify the patterns by extracting phases, and Section 3.4 describes the signature.

3.1 Data Collection

To instrument the applications, we must collect their communication and computation times. We create a dynamic library, libpas2p, using LD_PRELOAD to produce a trace of the binary (application). To instrument the application with libpas2p, it will be necessary that the application is compiled with dynamically linked libraries. Libpas2p intercepts the MPI functions before the MPI library executes it.

Starting from the concept of "Basic Block" (BB) [21], which is a code sequence with exactly one entry point and one exit point, we define similar concepts for parallel applications:

Event. The action of sending or receiving a message.

The event structure contains the following information:

- *Id*. Event identifier (given in order of occurrence).
- *Physical time*. time at which the event occurs.
- *Logical time (LT)*. time at which the event occurs depending on communications events.
- *Process*. process where the event occurs.
- *Type of event*. +K (if it is a Send) or –K (if it is a Receive), where K is the number of involved processes.
- *Size*. the communication volume of the message being transmitted (Bytes).
- *Number of event*. the number of the event in the process.
- *Relation*. the relation between one event and another, e.g., a Send event belongs to the same message as a Receive event.

3.2 Parallel Application Model

In parallel applications, logical ordering between computing nodes becomes necessary. To achieve this, we move from multiple physical local clocks to a single logical global clock. In a previous study [23], we showed a logical clock based on the order of precedence of events across processes, as defined by Lamport and Time [3]. In order to analysis the application, the happened-before relation is used [24] to evaluate the feasibility of using distributed computing infrastructures such as clusters or computational grids.

When we increased the number of processes, we found that the prediction quality was falling. An increasing number of phases meant that these phases could be grouped as similar. This problem occurred because there is a nondeterministic ordering of receives.

To solve the non-deterministic event (reception) problem, we introduced an algorithm [25] inspired by Lamport. Using this algorithm, we defined a new logical ordering in which, when one process sends a message at a logical time, its reception is modeled to arrive at LT + 1 and never afterwards, as in Fig. 3. The Lamport algorithm assigns the LT using *happened before*, but the Receive events occur as [nondeterministic] events. PAS2P ordering assigns the LT to the Recv event, based on the relation between the Recv and the Send event. In the case of collective communications (MPI_Bcast, MPI_Allreduce, MPI_Alltoall, etc.) and barriers, when a collective communication occurs, we select from all processes the event with the biggest LT and we assign LT + 1 to the events that compose the collective communication in all application processes. We show the flowchart of the algorithm in Appendix A of the supplementary

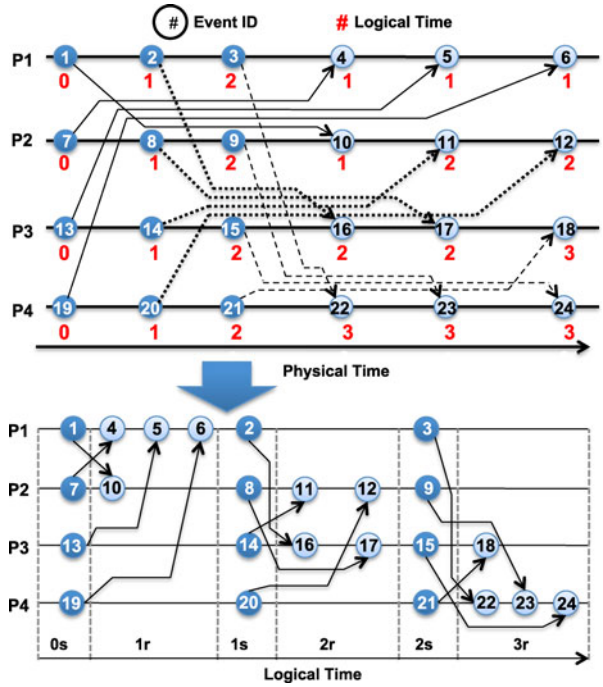


Fig. 4. Physical trace to logical trace.

material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2329688.

We now describe the operation of the algorithm through an example (Figs. 4 and 5 and Table 1), where we take the first event of each process and, as shown in Table 1, drop/insert it from/into the queue. The first column corresponds to the events being removed from the queue, the second column corresponds to events currently in the queue, and the third column shows the events being inserted into the queue.

Execution starts with the insertion of the first events from each process, IDs 1, 7, 13, and 19, into the queue (first step). We then remove the event with ID 1 from the queue (second step) and insert the event with ID 2 into the queue (third step), as in Table 1. As the event with ID 1 (CurrentEvent) has been removed from the queue, we must search for an event (BackEvent) from the same process (fourth step) and assign the corresponding LT. As the event with ID 1 is the first event in the process, we assign LT = 0 (fifth step) and LT = 1 (sixth step) to the reception event. The event with ID 7 is then removed (Table 1), and the same procedure of assigning a LT to the events we are removing from the queue is followed until the queue is empty (seventh step).

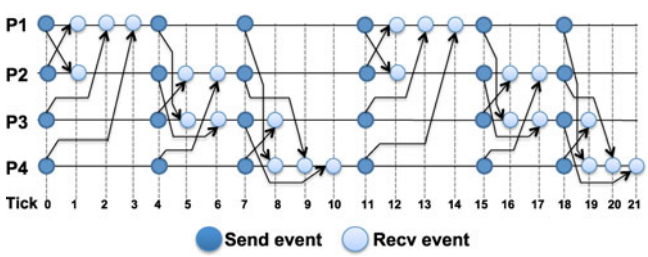


Fig. 5. Final logical trace.

TABLE 1
Dropped-Off/Inserted Events into Queue

Drop-off Event	Queue	Insert Event
1	7, 13, 19	2
7	13, 19, 2	8
13	19, 2, 8	14
19	2, 8, 14	20
2	8, 14, 20	3
8	14, 20, 3	9
14	20, 3, 9	15
20	3, 9, 15	21
3	9, 15, 21	No more events

When all events have been assigned a LT, we introduce a new concept: *Tick*: Logical time unit. We create a logical trace, where the LTs are assigned using the LT for the Send (LTSend) and Reception (LTRecv) events, as in Fig. 4. We know that the message reception ordering may change in the execution due to variable delays in the interconnection network; therefore, we perform a permutation only inside the LTRecvs of the logical trace so that the reception events are in ascending order. Finally, after locating the events, we divide the LTRecv into more LTs, as in Fig. 5; i.e. there can only be one event for each process at a particular LT.

3.3 Pattern Identification

To identify the most relevant portions (phases) of the parallel application, we have proposed a method [26] where we define a parallel basic block (PBB) as the computational time delimited by two ticks. The first tick is defined as an Entry Point and has at least one event, and the second tick is defined as the Exit Point, also having at least one event. PBBs with similar behavior (computational time between two events or communication volume and type of communication of each event) are renamed, as they essentially comprise the same PBB.

The proposed method to extract these phases creates them directly from the logical trace. Unlike the previously proposed algorithm for detecting phases, we can now identify phases as similar when before in our previous algorithm, we could not; therefore, we created a robust similarity algorithm generating fewer phases, through which the prediction quality increased. This method generates the longest possible phases; i.e. until another event occurs or is repeated and has the same type of communication in any process. Every time one phase grows in a tick, we search for an existing phase using similarity.

To explain the pattern identification algorithm, we use an example of a master/worker application where the workers start sending a message to the master then the master sends responses back to the workers illustrated in Fig. 6. Appendix B, which is available in the supplemental material section, shows the flowchart of this algorithm. We use these figures to describe how the algorithm extracts the phases from the logical trace. The steps we follow are:

1) A Startpoint is created and defined as the beginning point of a phase by a tick, from the first tick of the logical trace, as in Fig. 6a.

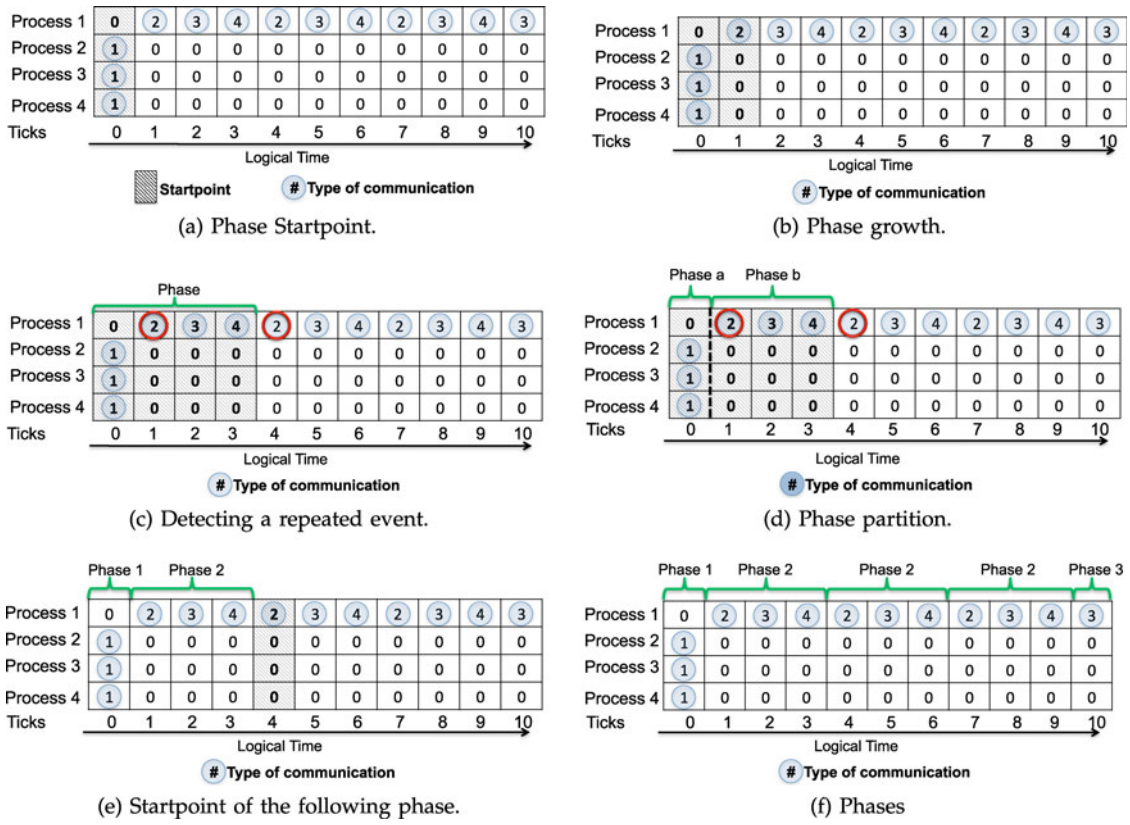


Fig. 6. Steps of the extraction phases algorithm

2) The phase extends to the following tick if it is not the end of the logical trace, as in Fig. 6b. Every time it expands by a tick, the existing phases are searched for a match.
3) If there is a process where no event with the same communication type occurs, we return to step 2 until this events occurs, as in Fig. 6c.
4) If an event with the same type of communication occurs:
   a) Verify whether the first repeated event is in the Startpoint; if so, we search by similarity (step 5) if the phase already exists.
   b) If the repeated event is not in the Startpoint, as in Fig. 6d, the phase is partitioned into two subphases:
      - *Phase a*. Begins at the Startpoint and ends just before the repeated event occurs.
      - *Phase b*. Begins at the tick where the first repeated event occurs and ends just before the second repeated event occurs with the same type of communication.
5) The similarity is used to determine whether the phase already exists and meets this criteria:
   a) The phase sizes (number of ticks) to be compared must be the same.
   b) Every event in the phase is compared thus:
      - We make a comparison of two events to see if they have the similar communication type and the communication volume.
      - A computational time between two events: if two events have similar computational time (greater than or equal to 85 percent).
      - A communication of type "0" represents that no event occurred at the tick of a process; when we compare "0" communication with any communication type, we label them as similar events.
   c) A phase is similar if the number of similar events is greater than or equal to 80 percent (configurable value) of the total number of events that comprise the phase.
      - If a phase is similar, the weight of the existing phase increases.
      - If it is not similar, it is saved as a new phase.
6) We go back to step 1 to create a Startpoint from the tick at which the last saved phase ends, as in Fig. 6e.

After completely shifting the logical trace, we obtain the phases shown in Fig. 6f, create weights for each phase and define the relevant phases.

Weight. This will be given by the frequency at which each phase repeats.

Relevant phase. A phase representativeness is given if the phase represents 1 percent or more of the entire application execution time. This is, when its weight multiplied by the phase execution time is equal or greater than 1 percent of the total application runtime.

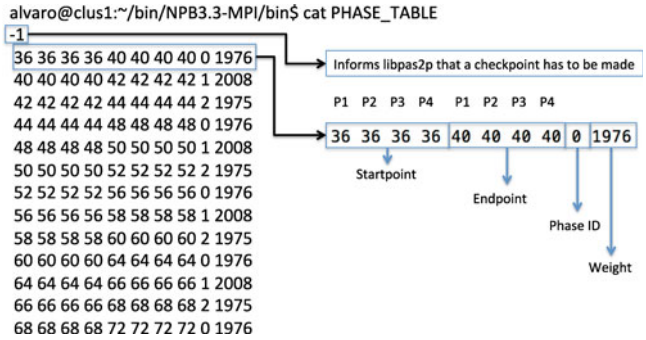


Fig. 7. Table of phases to construct the signature.

3.4 Parallel Application Signature

In this section we show how PAS2P constructs the signature once the phases have been obtained. After analyzing the trace files generated by the PAS2P instrumentation, the phases and their weights are saved in a phase table. Fig. 7 shows an example of this table, obtained from the execution of an application with four processes. It contains the startpoint and endpoint of each phase that will be used to measure its execution time. Each row of the table represents a phase, whose startpoint and endpoint are defined by the number of sends where the phase occurs. The last two columns show the Phase ID and the weights of the phase.

The checkpoint operation is implemented before the starting point of the specific phase to guarantee the correct warm-up time for the machines components (e.g., cache and TLBs) [27] as shown in Fig. 8. When the Startpoint of a phase occurs during execution, it sends a signal to all processes to coordinate and make the coordinated checkpoint [28]. Previous work on PAS2P used the BLCR library [29], which requires installation at the kernel level. We have analyzed different checkpointing libraries and selected the DMTCP library [30], which is a transparent user-level checkpointing library. The DMTCP creates snapshots of the application and the dynamically linked libraries and creates a binary that will contain the statically linked libraries.

To construct the signature, we have to instrument the application (binary); to do this, we use the same library we have been using to create the trace logs. This library interacts with the application, as well as with the external libraries. Another issue is how to detect relevant phases during the execution of the application. So, to construct the signature, we re-run the application loading the Libpas2p library and the phase table to instrument and detect where the phases occur.

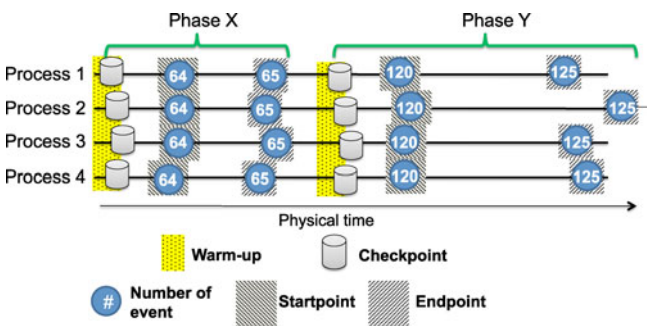


Fig. 8. Create the coordinated checkpoint.

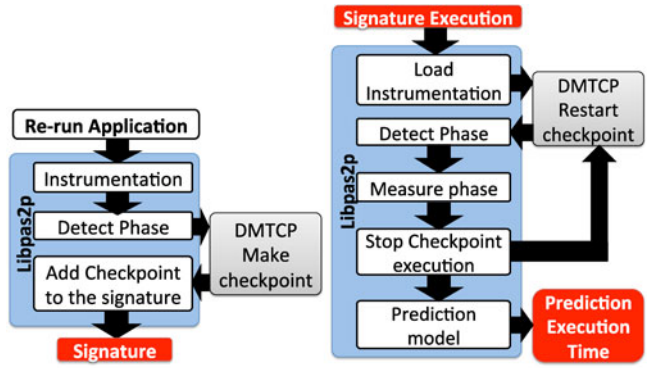


Fig. 9. Libpas2p calls to DMTCP to make and restart the checkpoint.

As shown in Fig. 9a, Libpas2p detects a phase and calls the DMTCP library to create the checkpoint. Once it is done, the DMTCP returns the control to the Libpas2p to add the checkpoint into the signature. After completing the checkpoint for the last phase, the signature terminates the execution because it is not necessary to continue its execution. Finally we generate the signature, which is an instrumented code that knows where each phase begins and ends. A demonstration about the construction of the signature using the phase table of Fig. 7 is available in Appendix C of the supplementary material.

4 Prediction Methodology

To predict the execution time (PET) of the application on a target machine, Equation (1) is used. When we multiply the execution time of each phase (PhaseETi) by its weight (Wi defined as the number of phase repetitions), we obtain the application execution time

$$PET = \sum_{i=1}^{n} (PhaseET_i)(W_i). \quad (1)$$

Run the signature means executing its constituent phases, the signature restarts the first checkpoint and after the warm-up begins measuring from the point a phase starts until it ends (communication events) as in Fig. 10. When a phase is measured, the signature terminates the checkpoint execution and restarts the next checkpoint. This method is repeated for all phases, as shown in Fig. 9b. Finally, the signature applies the Equation (1) to predict the execution time. A demonstration about execution of the signature is available in Appendix C of the supplementary material.

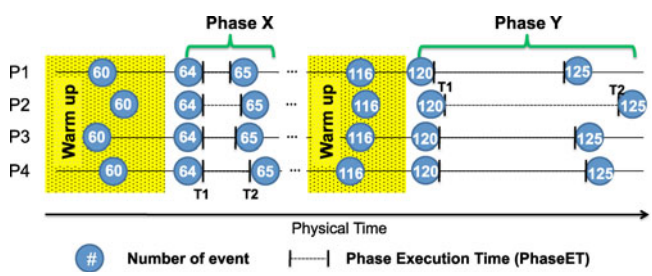


Fig. 10. Measuring phases.

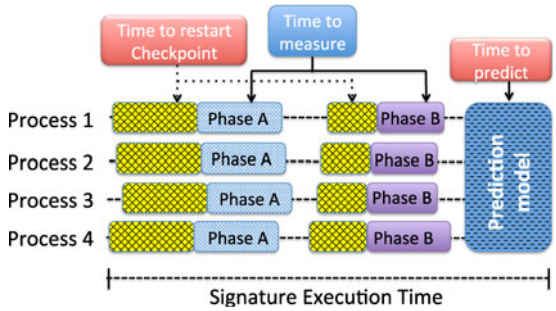


Fig. 11. The signature execution time.

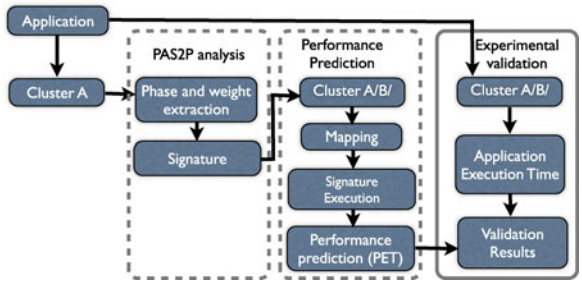


Fig. 12. Experimental methodology.

As Fig. 11 shows, the signature execution time comprises from the time it takes to restart the checkpoint to measure phase A until it finishes the prediction model. When phase A is measured, the signature terminates the checkpoint execution and restarts at the next checkpoint. When phase B is measured, all processes generate MPI messages to inform process 0 that they have finished and process 0 applies Equation (1) to predict the application execution time.

5 Experimental Results

In this section, we validate the prediction methodology demonstrating that the signature works and is able to obtain an accurate prediction of the application in a short time (signature execution time). Section 6 gives more details on the set of experiments we carried out in order to show the overhead generated by the instrumentation, the time required to analyze the tracefiles and the construction time of the signature. We show the signature execution for each application on three clusters, varying the number of cores. We predict their execution times and report the prediction quality of each signature. Table 2 shows the characteristics of these machines.

To evaluate the prediction quality and validate the proposed methodology, we performed an experimental evaluation on target machines, labeled A, B, C and for the experiments with different ISA (Cluster D) we show them in Appendix E, available in the supplemental material. We present the results obtained for the CG, BT and SP from the NPB, Sweep3D, POP, and SMG2000.

The method used to obtain the results involves executing each application, as shown in Fig. 1. We add a new block named experimental validation, illustrated in Fig. 12, where we execute the application on base machine to extract the signature and then we execute the signature in target machine changing the mapping policies to obtain the predicted execution time (PET). Finally, we execute the whole application on target machine to compare the predicted execution time with the application execution time to obtain the prediction execution time error (PETE).

To present how we obtain the predicted execution time, Table 3 shows information about the execution carried out in the base machine, as well as how we select the relevant phases based on the total number of them there, and once the signature has been constructed, Table 3 shows information about the execution of the signature, each phase's execution time and the signature prediction results. We executed the application MD Moldy with 256 processes to analyze and extract the phases to construct the signature. Table 3 also shows the size of the trace file and time required to analyze the trace file produced by the instrumentation, as well as its size (5.2 GB). When we analyzed the trace file, we extracted the 13 phases that compose the total application behavior. From here, to construct the signature we selected the most relevant phases, those ID's that represent 1 percent or more of the application execution time.

To obtain the prediction execution time (PET), it is necessary to execute the signature on the target machine to measure each phase execution time. This time also includes the computation and communication time of each phase on the target machine. Finally, using Eq. (1), we obtain the predicted execution time. In the same Table 3, we perform the same procedure, that is, we execute the application LU from

TABLE 2
Clusters Characteristics

Cluster	Characteristics
Cluster A 128 cores	Processor: Dual-Core Intel Xeon 5150 2.66 GHz, Memory: L2 4 MB, 8 GB RAM DIMM, Network: Gigabit Ethernet.
Cluster B 64 cores	Processor: 2 x Quad-Core Intel Xeon E5430, 2.66 GHz Memory: L2 2 x 6 MB, 16 GB RAM DIMM Network Gigabit Ethernet.
Cluster C 256 cores	Processor: 4 Intel Xeon Quad-core E7350 2.66 GHz Memory: L2 2 x 4 MB 16 cores, 48 GB RAM SDRAM Network: ConnectX IB Mellanoxcard
Cluster D 169 cores	Processor: 16 Itanium Montvale SMP NUMA Memory: 128 GB RAM Network: Infiniband 4 x DDR at 20 Gbps.

TABLE 3
Extraction and Execution of Phases on Cluster C

MD Moldy analysis			
Number of processes: 256, Input data: tip4p Size of log trace: 5.2 GB Time to analyze the log trace: 336.78 Sec Total of phases: 13, Relevant phases: 4			
Relevant Phase ID	PhaseET (Sec)	Weight	(PhaseET)*(Weight) (Sec)
1	0.003018	100,000	301.80
2	0.006131	89,976	551.64
3	0.000949	199,998	189.79
4	0.009387	9,998	93.85
Application Execution Time (Sec):			1169.31
Signature Execution Time (Sec):			1.69

TABLE 4
Parameter of the Applications

Application	Processes	Workload
CG, BT, SP	64	Class C
Sweep3d	32	Sweep.250 13 iterations
SMG2k	64	−n 200 solver 3
POP	64	Synthetic with 150 iterations

NAS Parallel Benchmarks, and GROMACS with a different number of processes. See Appendix D, which is available in the supplemental material. Appendix D, available in the online supplemental material, presents information about the analysis and its relevant phases, as well as their weights used to construct the signature and ultimately predict the execution time of each application.

We applied our prediction methodology to the above applications to extract phases and obtain the application signatures. After running the signatures from all applications, we obtained the execution time for each phase and the signature execution time, the SET is the sum of the execution times of all constituent phases. To obtain the predicted execution time, we multiply the execution time of each phase by the weight vector given by the PAS2P tool and add the times obtained.

5.1 Construct the Signatures on Cluster A (Base Machine) and Predict the AET for Target Machine

In this section we have constructed the signature on base machine (cluster A), aiming to validate our methodology being able to make predictions changing the mapping policies. Information about the executions on the base machine is shown in Appendix F, available in the online supplemental material. Once we have the signature, we take the signature to predict the performance for target machine (cluster B). We executed the applications shown in Table 4, which also shows the number of processes and the workload used for the application analysis to extract the signature.

TABLE 5
Predictions for Cluster B (Target Machine)

Appl.	Cores	SET (Sec)	SET versus AET(%)	PET (Sec)	PETE(%)	AET (Sec)
CG-64	32	8.42	0.29	2793.42	1.90	2847.42
	64	4.87	0.32	1504.66	0.48	1511.91
BT-64	32	13.47	0.80	1652.65	0.9	1667.64
	64	10.19	0.77	1302.76	0.55	1309.91
SP-64	32	2.04	0.24	808.76	1.28	819.17
	64	2.08	0.51	388.367	3.05	400.55
SMG2k	32	16.75	2.63	633.23	0.38	635.61
64	64	8.37	10.15	162.87	2.32	166.74
Sweep	16	4.32	0.17	2494.36	0.06	2492.74
3d-32	32	3.01	0.22	1328.04	0.40	1322.62
POP-64	32	22.79	1.41	1608.85	0.17	1611.59
	64	18.36	1.79	1016.01	0.61	1022.28

SET: Signature Execution Time, SET versus AET: 100(SET/AET).
PET: Predicted Execution Time, AET: Application Execution Time.
PETE: Prediction Execution Time Error.

TABLE 6
Parameter of the Applications

Application	Processes	Workload
CG, BT, SP	256	CLASS D
SMG2k	256	−n 200 solver 3 1200 iterations
Sweep3d	256	Sweep.200 13 iterations

Table 5 shows the results from cluster B (target machine) with different number of cores, indicating that the application runtime may vary depending on the number of cores, the core architecture (i.e., CPU speed, CPU cache, and interconnection between the cores) and the interconnection network. When comparing columns 3 (SET) and 7 (AET), it can be seen that the SET is notably shortened compared with the AET. Column 4 shows the percentage value obtained by dividing the SET by the AET, showing that the signature represents a small fraction of the application execution time. Column 5 shows the predicted execution time. Finally, column 6 presents the prediction execution time error.

These results show that the signature execution time decreased by 98.26 percent, meaning that the signature represents 1.74 percent of the application execution time. We also see that the prediction quality has an average accuracy of over 97.55 percent.

5.2 Construct Signatures on Cluster C (Base Machine) and Predict the AET for Target Machine

We used the same applications and changed the number of processes and workload for cluster C. As shown in Table 6, we instrumented each application to generate a tracefile to be used as input for the PAS2P tool giving phases and weights, which we use to build the signatures on the base machine. The information about the execution of the signature on the base machine is presented in Appendix F, available in the online supplemental material.

After generating signatures on cluster C, we moved them to cluster A to predict the time that the application takes to run. In this case, we have two different interconnection networks: cluster C has an Infiniband network, and cluster A has a Gigabit Ethernet network; each rack in cluster A has two dual-core processors, and cluster C has four quad-core processors.

We used cluster A as the target machine and executed the signature to obtain the prediction execution time for the application. As Table 7 shows, a yielding maximum error of 6.4 percent appears when we execute the entire application due to the execution of the relevant phases and not the total number of phases that the application has.

We have generated signatures with 256 processes on the base machine using 256 cores, the target machine (cluster A) has 128 cores; therefore, on cluster A, we mapped the signature assigning two processes to share the same core. The data shown in Table 7 allows us to analyze the signature executions on target machine. As we can see, once we have migrated the signature to a target machine that has fewer cores than the base machine on which it was created, the prediction error is still low and the SET compared with the AET is maintained below 8 percent.

TABLE 7
Predictions for Cluster A (Target Machine)

Appl.	Cores	SET (Sec)	SET versus AET(%)	PET (Sec)	PETE(%)	AET (Sec)
CG-256	128	59.52	2.03	2971.10	1.6	2922.24
BT-256	128	17.78	1.48	1182.67	1.5	1200.85
SP-256	128	17.53	0.77	2411.35	6.4	2265.40
SMG2k 256	128	120.17	1.75	6783.47	1.0	6858.17
Sweep 3d-256	128	82.28	7.62	1043.01	3.5	1079.13

SET: Signature Execution Time, SET versus AET: 100(SET/AET).
PET: Predicted Execution Time, AET: Application Execution Time.
PETE: Prediction Execution Time Error.

Finally, we ran the selected applications on the three clusters with different characteristics, generating signatures from 32 to 256 processors. In this case, we obtained an overall prediction error of 3 percent.

As a summary of our experimental results, we conclude that the proposed prediction methodology using signatures to predict the application execution time has enabled us to achieve two main objectives.

The first objective was to create a signature that had the same behavior as the application with an average prediction error of 3 percent. A portion of this error depends on the number of relevant phases that form the signature. If we take all the application phases, including both the relevant and not-relevant phases (initialization phases and finalization phases), this prediction error is reduced because the application execution time is the sum of the execution times of all phases.

The second objective was to make the prediction execution time fall within a bounded time using the signature execution. In this experimental validation, the signature execution time represents 1.74 percent of the total application execution time.

6 Performance of the PAS2P Tool

In this section, we show the overhead and the time PAS2P requires to instrument, analyze and construct the signature. We carried out an additional set of experiments using Sweep3d and sweep.150 as input. We have also used the CG, BT, SP, FT and LU applications from NAS using the class D as input. In SMG2K we used as input -n 200 -solver 3 -iterations 550 compiled with 128 processes. The applications mentioned above were executed on cluster C, then we proceeded to analyze and construct the signature to execute it afterwards, using this same cluster as a target machine.

Once PAS2P finished instrumenting the applications, we obtained the tracefiles. The size of the tracefiles is shown in column TFSize of Table 8. Column TFAT shows the time PAS2P requires to be able to analyze, create the application model and extract the phases. In the same Table 8, the Total Phases column gives the number of phases detected on every application used and from which we select the most relevant. The last column signature construction time (SCT) gives the time required to construct the signature. Being able to construct the signature requires to re-running the application in order to make the checkpoints, which we need to execute the relevant phases. To get the SCT we have to measure from the re-execution application until the last checkpoint saved, adding the time needed to make checkpoints.

TABLE 8
Performance of the PAS2P Tool in Order to Extract the Phases and Construct the Signature

Appl.	TFSize	TFAT (Sec)	Total Phases	Relevant Phases	SCT (Sec)
CG	593 MB	45.73	7	5	130.42
BT	292 MB	22.82	14	8	216.21
SP	617 MB	52.59	16	10	149.59
LU	5.2 GB	393.01	25	2	142.24
FT	512 KB	0.76	5	4	518.23
Sweep3d	1.8 GB	105.64	12	5	52.00
SMG2K	32 MB	10.27	7	3	43.20

TFSize: Tracefile Size, TFAT: Tracefile Analysis Time.
SCT: Signature Construction Time.

The PAS2P tool instruments the application generating an overhead as shown in Table 9. We had previously executed the applications without any instrumentation (AET) to be able to compare them with the execution of the application instrumented by PAS2P (AETPAS2P). The overhead caused by PAS2P instrumentation depends on the number of communication events that the applications have to perform. In this case, of all the applications we tested, LU has the highest overhead.

The same Table 9 shows the SET, which is the time required to measure the phases and predict the execution time of the application. Comparing both columns (AET versus SET) we can see that there is a considerable reduction in the AET. The last column shows the overhead given by total time it takes to generate and run the signature to achieve a performance prediction on target machine. We must point out that the construction of the signature is created only one time on base machine, then, to predict the performance, we migrate the signature to the target machines without the need to analyze the application again. Is an important note that the application FT has the highest overhead. When we analyzed the phases of the application (Table C.2 of the Appendix C, available in the online supplemental material), we found out that the highest weight was 20, meaning the application has little repetitiveness. The overhead in FT is due to the high SCT, this is, if we have to construct the

TABLE 9
Time Required to Obtain the Signature and Predict

Appl.	AET (Sec)	AETPAS2P (Sec)	SET (Sec)	Overhead
CG	512.10	522.29	11.4	1.37X
BT	846.42	848.09	35.41	1.31X
SP	1816.58	1831.08	37.38	1.13X
LU	623.41	668.44	24.64	1.96X
FT	371.03	387.38	68.66	2.62X
Sweep3d	439.28	455.81	43.48	1.49X
SMG2K	788.24	794.59	22.47	1.10X

AET: Application Execution Time.
AETPAS2P: Application Execution Time with PAS2P.
SET: Signature Execution Time.
Overhead: AETPAS2P+TFAT+SCT+SET/ AET.

signature we have to make a checkpoint to the phase but at the same time we have to guarantee the warm up of machine, therefore, the checkpoint is made after the phases have occurred a series of times, which is why the SCT is greater than the AET. Anyway, the signature is constructed only once in base machine. To predict the performance in difference machines only we have to execute the same signature in a very bounded time.

In case the application does not have a repetitiveness (communication repetitiveness), PAS2P can extract the phases, but the time to execute the phases will be similar as to execute the whole application. Another example is the master/worker pattern, where the master sends the job to the workers, then the workers compute and when they end the job send their results to the master. In this kind of applications PAS2P detects one phase with a weight of 1 and executing this phase will be the same as to execute the whole application.

7 Conclusions and Future Work

The PAS2P methodology allows us to generate a model of a parallel application and automatically extract its most significant phases to create a signature whose execution lets us predict the application's performance on different parallel computers. We tested our methodology over different clusters using a set of scientific applications, varying the number of cores or CPU type. We used dual core, 2 $\times$ quad-core and 4 $\times$ quad-core. In addition, we used different interconnection networks, such as Gigabit Ethernet and Infiniband, obtaining an average 97 percent prediction quality. As a utility, using the signature as meta-information allows us to evaluate the execution time of programs in service queues in order to provide useful information for schedulers.

To predict the performance of an application, we construct a signature that depends on the number of processes the application itself has executed. Nevertheless, the signature is able to execute using different mappings, increasing or decreasing the number of CPUs to know the application execution time using different system resources. The prediction that the signature gives would only be useful for the data set employed in the construction of the application signature. To predict with a different data set, we would have to re-execute the application for each one of them, in order to analyze and extract the phases that will compose the signature.

PAS2P has some limitations when it comes to extracting phases of applications with very little communication repetitiveness. When this happens, the runtime of the phases would be very similar to the execution time of the entire application. Additionally, PAS2P cannot predict I/O applications. Another limitation is that we cannot port the signature to the target machine since the target machine has a different ISA than the base machine. In this case, we can just construct the signature again, using the information from the phases and weight extracted in the base machine.

We propose that the PAS2P tool be used to help programmers identify performance issues when designing scientific algorithms, enabling the generation of a parallel application model. PAS2P allows us to easily create a signature that can be executed on other machines by simply extracting the application phases. It also enables us to predict how the application's performance will be on target machines by its execution. We have tested our methodology on a set of scientific applications, making variations on the number of nodes and the interconnection networks, obtaining a good prediction quality.

Acknowledgments

This research was supported by the MEC-MINECO-MICINN Spain, under contracts TIN2007-64974 and TIN2011-24384.

References

[1] J. McCalpin and C. Oakland, "An industry perspective on performance characterization: Applications vs benchmarks," *Proc. 3rd Annu. IEEE Workshop Workload Characterization*, Sep. 2000.

[2] J. Canillas, A. Wong, D. Rexachs, and E. Luque, "Predicting parallel applications performance using signatures: The workload effect," in *Proc. 9th IEEE/ACS Int. Conf. Comput. Syst. Appl.*, Dec. 2011, pp. 299–300.

[3] L. Lamport and C. Time, "The ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[4] D. Bailey, E. Barszcz, J. Barton, and D. Browning. (1991, Jan.). The NAS parallel benchmarks, *Int. J. High Perform. Comput.* [Online]. Available: http://hpc.sagepub.com/cgi/content/abstract/5/3/63, pp. 158–165.

[5] A. Hoisie, O. Lubeck, and H. Wasserman. (2000, Jan.). Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional. *J. High Perform. Comput. Appl.* [Online]. Available: http://hpc.sagepub.com/cgi/content/abstract/14/4/330, vol. 14, no. 4, pp. 330–346.

[6] J. Vetter, "Performance analysis of distributed applications using automatic classification of communication inefficiencies," in *Proc. 14th Int. Conf. Supercomput.*, New York, NY, USA, 2000, pp. 245–254.

[7] P. N. Brown, R. D. Falgout, J. E. Jones, Jim, and E. Jones, "Semicoarsening multigrid on distributed memory machines," *SIAM J. Sci. Comput.*, vol. 21, pp. 1823–1834, 2000.

[8] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," *J. Chem. Theory Comput.*, vol. 4, no. 3, pp. 435–447, 2008.

[9] K. Refson, "Moldy: A portable molecular dynamics simulation program for serial and parallel computers," *Comput. Phys. Commun.*, vol. 126, no. 3, pp. 310–329, 2000.

[10] J. Gustafson. (2012, Jun. 23). A new approach to computer performance prediction [Online]. Available: http://hint.byu.edu/documentation/Gus/France/France.html

[11] A. S. Laura, L. Carrington, N. Wolter, and T. San, "A framework for performance modeling and prediction," in *Proc. ACM/IEEE Supercomput.*, 2002, pp. 1–17.

[12] S. Sodhi, J. Subhlok, and Q. Xu. (2008, Jan.). Performance prediction with skeletons. *Cluster Comput.*, vol. 11, no. 2, pp. 151–165, 2008 .

[13] X. Wu, V. Deshpande, and F. Mueller, "Scalabenchgen: Auto-generation of communication benchmarks traces," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 1250–1260.

[14] S. Girona, J. Labarta, and R. M. Badia, "Validation of dimemas communication model for MPI collective operations," in *Proc. 7th Eur. PVM/MPI Users' Group Meeting Recent Adv. Parallel Virtual Mach. Message Passing Interface*, London, U.K., 2000, pp. 39–46.

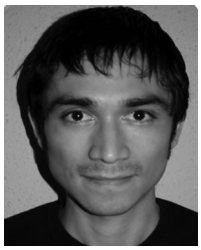
[15] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong, "Detecting phases in parallel applications on shared memory architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2006, p. 68.

[16] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang, "Mambo: A full system simulator for the powerpc architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 8–12, 2004.

[17] L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Proc. IEEE/ACM Supercomput.: High Perform. Netw. Comput. Conf.*, 2005, p. 40.

[18] M. Casas, R. M. Badia, and J. Labarta, "Automatic phase detection and structure extraction of MPI applications," *Int. J. High Perform. Comput. Appl.*, vol. 24, no. 3, pp. 335–360, 2010.

[19] H. Brunst, D. Kranzlmüller, M. S. Muller, and W. E. Nagel, "Tools for scalable parallel program analysis: vampir ng, marmot, and dewiz," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 3, pp. 149–161, Jul. 2009.
[20] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalatrace: Scalable compression and replay of communication traces for high-performance computing," *J. Parallel Distrib. Comput.*, vol. 69, no. 8, pp. 696–710, Aug. 2009.
[21] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proc. Int. Conf. Parallel Archit Compilation Tech.*, Jan. 2001, pp. 3–14.
[22] E. Perelman, M. Polito, J. yves Bouguet, J. Sampson, B. Calder, and C. Dulong, "Detecting phases in parallel applications on shared memory architectures," in *Proc. Int. Parallel and Distributed Process. Symp.*, 2006, pp. 25–29.
[23] A. Wong, D. Rexachs, and E. Luque, "Parallel application signature," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, Aug. 31, 2009–Sep. 4, 2009, pp. 1–4.
[24] H. Brunst, D. Kranzlmüller, M. S. Muller, and W. E. Nagel, "Tools for scalable parallel program analysis: vampir ng, marmot, and dewiz," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 3, pp. 149–161, Jul. 2009.
[25] A. Wong, D. Rexachs, and E. Luque, "Parallel application signature for performance prediction," in *Proc. Int. Conf. Parallel Distrib. Process. Tech. Appl.*, 2010, vol. 2, no. 408–414.
[26] A. Wong, D. Rexachs, and E. Luque, "Pas2p tool, parallel application signature for performance prediction," in *Proc. 10th Int. Conf. Appl. Parallel Sci. Comput.—Volume Part I*, 2012, pp. 293–302.
[27] G. Hamerly, E. Perelman, and B. Calder, "How to use simpoint to pick simulation points," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 25–30, 2004.
[28] J. Hursey, J. M. Squyres, and A. Lumsdaine. (2006, Jul.). A checkpoint and restart service specification for open MPI, Indiana Univ., Bloomington, IN, USA, Tech. Rep. TR635 [Online]. Available: http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR635
[29] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," *J. Phys. Conf. Series*, vol. 46, no. 1, pp. 494–499, 2006.
[30] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–12.



Alvaro Wong is an associate researcher at the Computer Architecture and Operating System Department, University Autonoma of Barcelona, Spain. He has worked in performance prediction of HPC applications in the ITEA 2 European Project No 09011, research centers, and industries. He has coauthored a total of 11 full-reviewed technical papers in journals and conference proceedings.

Dolores Rexachs is an associate professor at the Computer Architecture and Operating System Department, University Autonoma of Barcelona (UAB), Spain. She has been the supervisor of seven PhD thesis and has been invited lecturer in Universities of Argentina, Brazil, Chile, and Paraguay. The research interests include parallel computer architecture, parallel I/O subsystem, fault tolerance in parallel computers, tools to evaluate, predict, and improve the performance in parallel computers. She has coauthored more than 50 full-reviewed technical papers in journals and conference proceedings.

Emilio Luque is a professor at the Computer Architecture and Operating System Department, University Autonoma of Barcelona, Spain. Invited lecturer at universities in US, South America, Europe, and Asia, key note speaker in several conferences and leader in several research projects funded by the European Union (EU), Spanish government, and different industries. His major research areas are: parallel and distributed simulation, performance prediction, and efficient management of multicluster-multicore systems and fault tolerance in parallel computers. He has supervised 19 PhD thesis and coauthored more than 230 technical papers in journals and conference proceedings.