J Supercomput (2012) 62:1609–1634
DOI 10.1007/s11227-012-0823-5

A framework for efficient performance prediction of distributed applications in heterogeneous systems

Bogdan Florin Cornea · Julien Bourgeois

Published online: 14 September 2012

Abstract Predicting distributed application performance is a constant challenge to researchers, with an increased difficulty when heterogeneous systems are involved. Research conducted so far is limited by application type, programming language, or targeted system. The employed models become too complex and prediction cost increases significantly. We propose dPerf, a new performance prediction tool. In dPerf, we extended existing methods from the frameworks Rose and SimGrid. New methods have also been proposed and implemented such that dPerf would perform (i) static code analysis and (ii) trace-based simulation. Based on these two phases, dPerf predicts the performance of C, C++ and Fortran applications communicating using MPI or P2PSAP. Neither one of the used frameworks was developed explicitly for performance prediction, making dPerf a novel tool. dPerf accuracy is validated by a sequential Laplace code and a parallel NAS benchmark. For a low prediction cost and a high gain, dPerf yields accurate results.

Keywords Performance prediction · Distributed applications · Automatic static analysis · Block benchmarking · Trace-based simulation · dPerf

1 Introduction

The field of parallel and distributed computing has represented the point of interest for many researchers since several decades ago. The constant need for increasing the computing power leads to the development of numerous performance prediction methods and tools. These mainly help researchers in choosing the computing

B.F. Cornea (✉) · J. Bourgeois
UFC/FEMTO-ST Institute, UMR CNRS 6174, Besançon, France
e-mail: bogdan.cornea@univ-fcomte.fr

J. Bourgeois
e-mail: julien.bourgeois@femto-st.fr

architecture that best suits the requirements of their scientific applications. Nowadays, technological improvements make traditional prediction methods completely inefficient, and those that are more recent cannot handle the complexity of today's computing systems.

We present dPerf (**d**istributed **Perf**ormance prediction), a performance prediction method, which is based on previous works [5] and [12]. In dPerf, we implemented our method for performance prediction of High Performance Computing (HPC) applications written for parallel or distributed systems for centralized or decentralized architectures. It supports three of the most intensively used programming languages in the sphere of HPC: C, C++, and Fortran. In addition, the distributed programs can communicate using either P2PSAP [10, 23], or MPI [36]. dPerf uses externally developed frameworks for extending static analysis methods and the trace-based simulation mechanism. dPerf is the result of defining and implementing new methods for block benchmarking and instrumentation such that the outcome of our tool would be the performance prediction of the input code. dPerf addresses homogeneous or heterogeneous systems, it produces fast predictions, it offers support for C, C++ or Fortran code, it gives accurate prediction with a reduced slowdown, it supports distributed applications which communicate using MPI or P2PSAP, and it defines a novel method for taking into account the cache memory effect and the compiler optimization levels.

The work presented in the following has been previously presented in a more briefly manner in [7]. In the current paper, we detail more on the latest related work, we go into detail concerning the methodology and the requirements, and from experimental point of view, we present results from three experimental sets using all compiler optimization levels. We performed experiments with two applications, one sequential and the other parallel. We validate our prediction framework from sequential point of view and then we prove its accuracy on two different distributed computing systems.

Section 2 presents relevant previous work in the field of performance prediction for parallel and distributed applications. Our motivation and the requirements for the presented work are explained in Sect. 3. The results obtained experimentally are presented in Sect. 4, followed by conclusions and perspectives in Sect. 5.

2 Related Work

Performance prediction tools are proposed in parallel with the technological advancement. A great amount of effort is needed for building a tool that would efficiently handle nowadays architectures due to the high complexity of current hardware devices. The performance prediction must be obtained without significant costs so that developers can quickly have an insight on the applications that they are developing for a target platform not available throughout the development process.

We classify performance prediction methods and tools into *analytical*, *profile-based*, and *hybrid*. All methods presented in this section are compared directly to our proposition, dPerf, in Tables 1 and 2.

Table 1 Comparative view between most relevant performance prediction tools and our method—dPerf

Tool	Approach	Target system	Envir.	Input code			Support for compiler optimizations
				Type	Communication	Language	
dPerf	hybrid	heterogen.	P2P	distributed	P2PSAP, MPI	C, C++, Fortran	yes
LogP	analytic	homogen. CoW[f]	centr.[h]	parallel	shmem[a], MP[b], data parallel	any	no
LogGP	analytic	homogen. CoW	centr	Sweep3D	MPI	Fortran and C	no
van Gemund '03	analytic	homogen. CoW	centr.	parallel	MPI	any	no
P^3T	profiling	homogen CoW	centr.	parallel	MP[b]	Fortran	no
Saavedra'96	profiling	–	–	sequential	no comm.	Fortran, any other	no
ChronosMix	profiling	heterogen.	centr.	distributed	MPI	C	no
HAME	hybrid	heterogen.	centr.	parallel	any[c]	any[c]	no
MPI-Sim	hybrid	homogen.	centr.	parallel	MPI	Fortran	no
SimGrid SMPI	hybrid	heterogen.	any	parallel	MPI	C	yes
POEMS	hybrid	homogen.	centr.	distributed	shmem[a], MP[b], data parallel	any[d], C[e], Fortran[e]	no
Snavely'01[i]	hybrid	heterogen. grid, NoW[g]	centr.	distributed	MP[b]	C, C++, Fortran	no
DIMEMAS	hybrid	heterogen. grid, NoW[g]	centr.	distributed	MP[b]	C, C++, Fortran	no

Table 1 (*Continued*)

Tool	Approach	Target system	Envir.	Input code			Support for compiler optimizations
				Type	Communication	Language	
ScalaTrace	hybrid	homogen.	centr.	parallel	MPI	–	no
DeRose'09	hybrid	heterogen. Cray	centr.	distributed	MPI, OpenMP	any	no
PHANTOM	hybrid	heterogen.	centr.	parallel	MPI	Fortran	no

[a]Shared memory

[b]Message passing

[c]Not proven experimentally

[d]For analytical models

[e]For simulation models

[f]Cluster of workstations

[g]Network of workstations

[h]Centralized

[i]Limited by DIMEMAS performance

Table 2 Performance and efficiency. Comparison of dPerf to most relevant performance tools

Tool	Values			
	Slowdown	Speedup	Accuracy	Pred. cost
dPerf	0.2–1.1	–	>80 %[a]	approx. 1[c]
LogP	assumed $\ll 1$	–	assumed high	$\gg 1$ (very high)
LogGP	assumed $\ll 1$	–	>80 %	$\gg 1$ (very high)
van Gemund'03	assumed $\ll 1$	–	50–90 %	$\gg 1$ (very high)
P^3T	>1	–	>98 %	>1
Saavedra'96			75–95 %	
ChronosMix	0.02–0.25	–	>85%	$\ll 1$[e] or 1[f]
HAME[b]	assumed $\ll 1$[b]	–	assumed poor[b]	assumed low[b]
MPI-Sim	–	<12	>80 %	–
SMPI	<1	–	poor	very low
POEMS	assumed $\ll 1$	–	low to high	$\gg 1$ (very high)
Snavely'01[h]	≥ 1	–	>80 %	$\gg 1$
DIMEMAS	≥ 1	–	>80 %	–
ScalaTrace	>1	–	assumed high	–
DeRose'09	$\gg 1$[f]	–	assumed high	average
PHANTOM	assumed $\gg 1$[g]	–	>95 %	assumed low

[a]Accuracy on more than 8 processes

[b]Not proven experimentally

[c]Slightly longer than one full execution

[d]Static analysis

[e]Post-mortem analysis

[f]Several executions are required to build a knowledge base

[g]Execution done on one node, and prediction for N nodes

[h]Limited by DIMEMAS performance

Purely *analytical* methods had been employed in the work of [9] for LogP, [35] for LogGP and [37]. To apply the aforementioned methods, a thorough understanding of the algorithm under evaluation is necessary and this leads to a high cost for obtaining a prediction. With respect to the above mentioned methods, dPerf takes less time to obtain a prediction, addresses decentralized heterogeneous computing systems, works with more than just the message passing implementations, supports most commonly used programming languages in HPC, and takes into account compiler optimization levels.

Profile-based methods are those that make use of hardware counters or instrumented sources to retrieve data-dependency information at execution time. Relevant work in this section has been undergone by [5, 13, 14, 31]. With respect to this category of tools, dPerf brings a few improvements such as the support for a communication protocol other than message passing, accepts more than one programming language at input, and yields accurate results when different compiler optimization levels are used.

An efficient trade-off is represented by *hybrid* methods, combining the analytical model with information obtained through profiling. In our opinion, the supercomputing systems or the applications meant to run on these could be best evaluated and estimated by applying hybrid methods due to the high complexity of the architectures or algorithms under evaluation. HAME [18], a very interesting approach presented by Li et al., predicts program performance in a static manner for application-tuning purposes. Methods based on full simulation [6, 29] provide the most accurate predictions. The results demand the design of an accurate model of the target architecture which involves a high prediction cost. In order to keep development costs at a reasonable level, trace-based simulators become widespread.

The simulator partially describes computing systems and the rest is supplied by trace files. SimGrid [6] is a framework suitable for developing custom simulation tools. SimGrid can perform full or trace-based simulations with its built-in SMPI and MSG module, respectively. POEMS [1] is a very complex system for building end-to-end modules which combines analysis, simulation, and direct measurement. Snavely et al. [34] describes another approach that uses a performance prediction calculated on a single-node computer, which is afterward passed to DIMEMAS [3] in order to simulate the parallelism. ScalaTrace, presented in [24], is a method that regards only the communication aspect in a parallel environment. A point of transition from performance monitoring to performance analysis is emphasized in [30]. Authors present an approach which uses a knowledge base along with a set of performance models. Another relevant contribution in this field was brought by Zhai et al. in [39]. The authors presented PHANTOM, a framework that addresses parallel applications written in Fortran, communicating using MPI, and meant to run on either heterogeneous or homogeneous platforms. By a quick comparison of PHANTOM to dPerf, we state that our method supplies faster predictions, works with three of the most employed languages in HPC, and offers support for distributed applications meant to run in decentralized HPC systems.

With one exception, which is [39], the above mentioned methods address a limited number of aspects of nowadays parallel and distributed applications and systems. Some methods only work for single-processor systems, others only for homogeneous clusters of workstations. Some methods only apply to specific applications or are addressing only message passing parallel programs. Other approaches lack multi-application support or are too dependent on the target platform and network topology. Because researchers demonstrated in [5, 11, 12] that static and semistatic methods give promising results, we propose dPerf, the tool that we have developed, for efficiently combining (i) automatic static analysis based on block benchmarking with (ii) instrumented execution, and with (iii) trace-based simulation of the message-exchange.

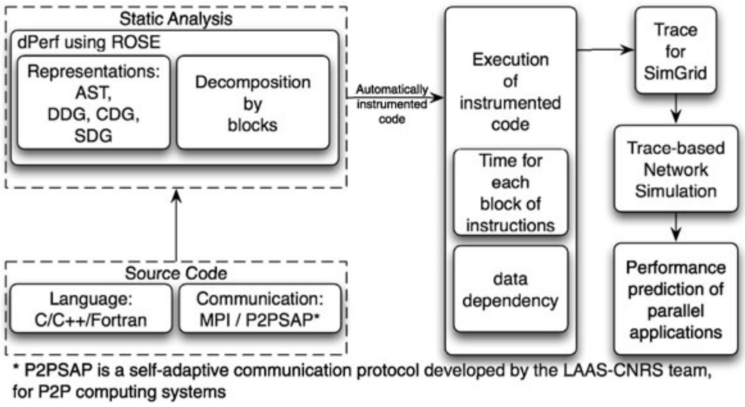

Static Analysis
dPerf using ROSE
Representations: AST, DDG, CDG, SDG
Decomposition by blocks
Automatically instrumented code
Execution of instrumented code
Time for each block of instructions
data dependency
Trace for SimGrid
Trace-based Network Simulation
Performance prediction of parallel applications
Source Code
Language: C/C++/Fortran
Communication: MPI / P2PSAP*
* P2PSAP is a self-adaptive communication protocol developed by the LAAS-CNRS team, for P2P computing systems

Fig. 1 The dPerf framework containing static analyzer (for static analysis and instrumentation of the input source code) and trace-based simulator

3 Methodology and requirements

At the current development state, performance prediction made by dPerf is still dependent on the following factors: (i) the number of nodes, (ii) node architecture, and (iii) application parameters (such as the problem size).

In order to predict application performance, the source code is analyzed, the transformed code is run, instruction execution times are measured, and distributed network topologies are simulated (see Fig. 1). For this, a series of requirements must be met.

3.1 Requirements

For fast and precise measurements that introduce very little noise in the measured system, we make use of *hardware* (or performance) *counters*. The hardware counters are easily accessible in GNU/Linux. Two of the most important measurement infrastructures that can enable the performance counters module are *perfctr* [28] and *perfmon* [27]. Researchers affiliated with the University of Lugano presented a comparative study in [38]. Based on this study, our approach uses *perfctr*.

We access the hardware counters using *PAPI*, the Performance Application Programming Interface [25, 26]. Through PAPI, developers gain access to a wide range of information given by the counters with a minimum noise level introduced into the measured system, thus improving performance analysis results. Two interfaces are available within the PAPI library: a high-level and a low-level interface. The *PAPI high-level interface* is used for performing quick and simple measurements, such as retrieving the total number of available hardware counters or getting the number of nanoseconds (ns) since the previous call to PAPI, and so on. The *PAPI low-level interface* is less restrictive and it provides an advanced interface for our performance prediction tool. From the low-level interface of PAPI, we use *PAPI_get_virt_nsec*. This method accesses the counters and gets the virtual (user) time in nanoseconds. We are interested in capturing this (user) time. As shown in Fig. 2, the user time is the time spent by a CPU only for the user process, without measuring the time spent on

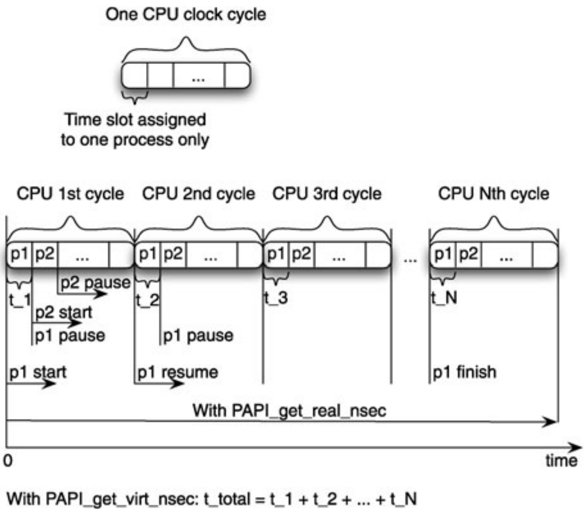


Fig. 2 Measuring the real and virtual time cost with PAPI

CPU cycles of other processes. Before, timing was possible using calls to functions such as *gettimeofday*. Even though it expresses the measurement in microseconds, its real count is not based on microseconds, but it is dependent on the time slot assigned to the process that made the call to *gettimeofday*. Some UNIX systems used to update the *gettimeofday* value every 10 milliseconds [15]. Moreover, the processor cost for calling the *gettimeofday* function itself is quite significant.

One very important requirement is for our tool, dPerf, to accept C, C++, and Fortran, three of the most intensively used languages in distributed programming. At the current development level, the source code used as input must be deterministic. *Rose* [32] is a compiler infrastructure for building custom source-to-source program transformation and analysis tools. It can analyze large scale applications. Since tools based on Rose accept C, C++, Fortran, OpenMP, and UPC programs, they cover most applications running on parallel and distributed systems. Rose is most suited for building tools for static analysis, as well as program and performance analysis. We have chosen to develop our static analyzer for dPerf using Rose compiler mainly due to the following reasons: (i) we aim at analyzing C, C++ and Fortran applications, (ii) no information from the input code is lost during static analysis, (iii) the intermediate representations (IR) in Rose provide the abstract syntax tree (AST) as well as the system dependence graph (SDG), and (iv) the input code, statically transformed, is available at output.

The *Abstract Syntax Tree (AST)* is the fundamental syntactic representation of a single file source code. It can be easily analyzed and based on its traversal any transformation can be performed. dPerf uses the AST built by Rose to identify key elements such as statements, basic blocks, and communication calls.

The *System Dependence Graph (SDG)* [19] is a supergraph containing the data and control dependencies combined into one representation. The SDG is useful in solving variable dependency such as finding the constant value of a conditional statement parameter.

SimGrid [6] is a project that gives developers the possibility to build custom simulators of parallel and distributed computing systems. We use SimGrid's *MSG* module for customizing a trace-based simulator. This simulates the distributed network topology and calculates the total predicted execution time. We extend existing communication primitives in MSG and we define other communication primitives for new protocols such as P2PSAP. Using the output from dPerf, SimGrid's MSG module solves the communication time aspect of our MPI application performance prediction.

3.2 Methodology

Our contribution in this field is focused on providing accurate performance predictions which take into account compiler optimization. This contribution consists of three main phases: (i) the automatic static analysis of an input code, (ii) the execution of the instrumented code issued by the static analyzer, and (iii) trace-based simulation for obtaining the final prediction. These phases are shown in Fig. 1. A demonstration of performance prediction with dPerf can be seen in [8]. The following terminology is employed throughout the rest of the paper:

- $t_{\text{normal execution}}$: reference time; time to run an application; measured from the beginning and until the end of an application execution;
- t_{compute}: processor computation time; measured by the hardware counters in terms of user cycles, i.e., no processor cycles assigned to other processes;
- $t_{\text{communication}}$: communication time; characterized by an exchange of data between participant processes;
- $t_{\text{prediction}}$: necessary time to go through the entire prediction process for obtaining a predicted time (see $t_{\text{predicted}}$ bellow);
- $t_{\text{simulation}}$: time needed to run a trace-based simulation;
- $t_{\text{obtain trace files}}$: time cost for obtaining execution traces; the counter of this duration starts with the static analysis and ends once trace files are obtained; includes static analysis and execution of the instrumented code;
- $t_{\text{predicted}}$: final result of the performance prediction process, i.e., the predicted execution time for the input application; dPerf uses the simple block benchmarking method;
- $t_{\text{threshold predicted}}$: similar to $t_{\text{predicted}}$; dPerf applies the optimized block benchmarking method.

Automatic static analysis dPerf uses Rose methods to obtain the intermediate representations AST and SDG for the given input code. Using the AST, dPerf decomposes the input code into instruction blocks. Each block is verified for MPI function calls. At this point in the analysis process, two different techniques can be employed: (i) a simple block benchmarking and (ii) an optimized block benchmarking.

The *simple block benchmarking* technique only marks the end of an instruction block and prepares the output of communication parameters for the creation of trace file later-on at execution time. Later in the prediction process, this leads to output traces as in the following listing:

```
p0 compute 5385
p0 Isend p0 8429748
```

```
p0 Recv p0
p0 compute 46447205
```

Lines shown above correspond to computation and communication lines of code, as they occur in the source file. Computation cost is expressed in nanoseconds, while communication is expressed in bytes. For instance, *p0 compute 5385* means that on process 0, a block of instructions has computed for 5385 nanoseconds, measurement taken by the hardware counters. Similarly, *p0 Isend p0 8429748* means that process 0 performs an Isend toward itself (process 0) and an amount of 8429748 bytes are transferred.

Optimized block benchmarking, which we also named *the Threshold iteration rule*, is a method which we propose and which we implemented in dPerf. It extends the simple block benchmarking by reducing loop boundaries to a minimum but representative number of iterations which preserves computation and communication characteristics. This method can only be used on loops with independent iterations. The decision if loops have data-dependent or independent iterations is taken based on SDG analysis. If loops are data-dependent, the Threshold iteration rule cannot be applied. This approach is a block benchmarking method that takes into account the instruction prefetching effect. We observed that after a certain number of iterations, the time per iteration is constant within a small error interval. Let the reference number of iterations of a block be denoted by *th*, or threshold, with t_{th} being the time in nanoseconds where *th* is reached. For a block with a single iteration, the Threshold iteration rule is expressed as follows:

$$t_{\text{avg}_{\text{iteration}}} = \frac{t_{\text{th}}}{th} \tag{1}$$

where $t_{\text{avg}_{\text{iteration}}}$ is the average time for one block iteration, value that takes into account the time for loading data into memory. If the estimated time for the entire loop is denoted by $t_{\text{loop estimated}}$, for the above mentioned we have

$$t_{\text{loop estimated}} = t_{\text{avg}_{\text{iteration}}} \times 1 = \frac{t_{\text{th}}}{th} \tag{2}$$

In the case of blocks belonging to loop statements with n cycles, formula (2) becomes:

$$t_{\text{loop estimated}} = t_{\text{avg}_{\text{iteration}}} \times n = \frac{t_{\text{th}}}{th} \times n \tag{3}$$

Above *th*, $t_{\text{avg}_{\text{iteration}}}$ is constant or within an ε_{th} error interval. Let x be the number of iterations of a block, and t_x its average execution time, then

$$\varepsilon_{\text{th}} = t_x - t_{\text{th}}, \quad \forall x > th \tag{4}$$

The curve $t_{\text{avg}_{\text{iteration}}}$ in Fig. 3 is obtained from running a sample instruction block. This sample consists of a loop, the number of iterations being indicated on abscissa. During the first few iterations, data prefetching implies a great time cost. As the loop reiterates, it can be noticed that costs related to data being loaded into memory are decreasing. From the shape of the $t_{\text{avg}_{\text{iteration}}}$ curve, we state that for most loops, the time cost per loop iteration decreases with the increase in number of iterations. Based on the above observation, an error interval ε can be defined. Let ε_{th} be the error percentage level defined in formula (4) such that for any x number of iterations, the

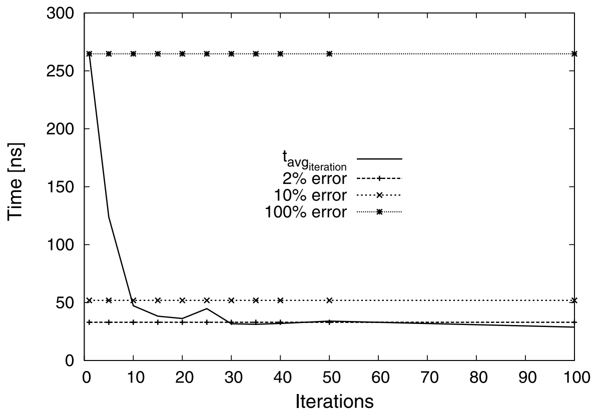


Fig. 3 Threshold and possible error levels for a sample instruction block consisting of a loop

time per iteration $t_{\mathrm{avg}_{\mathrm{iteration}}}$ is always smaller than $\varepsilon_{\mathrm{th}}$. In Fig. 3, if 2 % is the maximum accepted error ($\varepsilon_{\mathrm{th}} = 2\ \%$), then a minimum of 30 out of 100 iterations are required. For $\varepsilon_{\mathrm{th}} = 10\ \%$, only 10 out of 100 iterations are necessary to correctly estimate the loop overall execution time.

We extend this principle to the various optimization levels available with G++. We observed that for the same sample block of instructions as the one in the Fig. 3, the execution time decreases as the number of reiterations of the same block takes place.

For Fig. 3, for a chosen $\varepsilon_{\mathrm{th}}$, we propose the threshold method which only iterates a minimum number of times that provides sufficient information for estimating the time cost for the complete number of iterations. The estimation is done, as expressed in formula (3), by measuring the first *th* iterations, computing an average $t_{\mathrm{avg}_{\mathrm{iteration}}}$ multiplied by the number of iterations n of the original loop.

There are two practical cases that may occur when applying the threshold iteration rule:

- the loop does not contain calls for communication, or
- the loop contains communication.

In both cases, the trace files obtained later in the prediction process will only contain $t_{\mathrm{loop\ estimated}}$. Optimized block benchmarking when *loops do not contain communication calls* means reducing the loop upper boundary to a minimum significant level. For *loops containing communication calls*, in addition to the previous case, the traces contain a succession of computation times and communication parameters. For this reason, at the first iteration of the loop under evaluation, the communication parameters are stored aside. For the next iterations, we no longer evaluate the communications, but only apply the remaining of the threshold rule, until we obtain $t_{\mathrm{avg}_{\mathrm{iteration}}}$.

Dynamic analysis In this second prediction phase, a transformed code previously obtained from the static analyzer is built and executed for different compiler optimization levels: 0, 1, 2, 3, and s. Each level provides an optimization degree that we take into account when comparing dPerf predictions to the execution of the original code. The use of compiler optimization places dPerf predictions in real execution conditions. Executing the instrumented code generates trace files corresponding to each participant process.The traces contain only computation times and the relevant parameter for each communication. Computation is measured using hardware counters and is expressed in nanoseconds.

Trace-based simulation and prediction result The trace files obtained through dynamic analysis are used for simulating the network and for computing $t_{\text{predicted}}$, the overall prediction of the test application. Among the innovations brought by the method that we present in this paper we mention a reduced *slowdown* due to the two block benchmarking techniques that we propose: (i) the simple and (ii) the optimized block benchmarking techniques. The slowdown is the unit of measurement for the efficiency of performance prediction tools. It is defined as the ratio of the time for obtaining a performance prediction to the normal application execution time. The slowdown is expressed per process, based on measurements taken on the same architecture.

$$slowdown = \frac{t_{\text{prediction}}}{t_{\text{normal execution}} \times Nb\_Processes} \tag{5}$$

where

$$t_{\text{prediction}} = t_{\text{obtain trace files}} + t_{\text{simulation}} \tag{6}$$

In general, performance prediction tools have a *slowdown* greater than one, i.e., most tools take longer to predict performances than to execute an application. dPerf is characterized by a reduced slowdown. This means that in our case, dPerf has a *gain*, and not a slowdown. The gain is the inverse of the slowdown. Per simulated process, dPerf yields prediction results faster than the execution time, except for some very rare cases. For example, if a prediction tool has a slowdown s, then it means that calculating the performance prediction for an application ($t_{\text{prediction}}$) is slowed down s times (per simulated process) with respect to the normal execution time ($t_{\text{normal execution}}$).

$$t_{\text{prediction}} = s \times t_{\text{normal execution}} \tag{7}$$

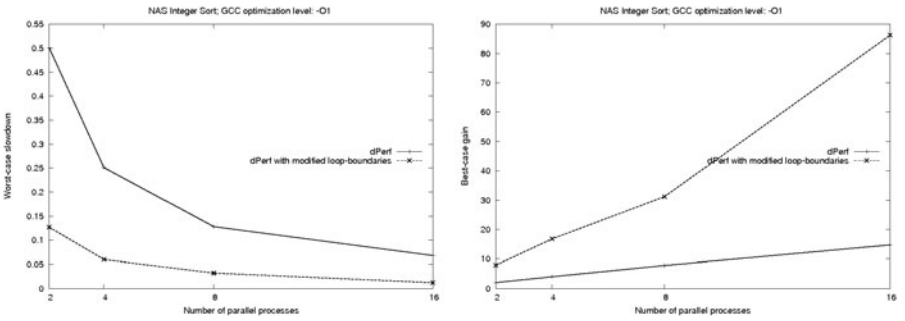
If a performance prediction is calculated for an architecture consisting of N_p processes, then the time for obtaining the prediction is

$$t_{\text{prediction}} = s \times t_{\text{normal execution}} \times N_p \tag{8}$$

$t_{\text{obtain trace files}}$ is the time taken to statically analyze the input source code plus a one time execution of the code transformed by dPerf in order to obtain the trace files. $t_{\text{simulation}}$ is the time taken by SimGrid MSG to perform a trace-based simulation.

dPerf support for modifying loop boundaries is undergoing, thus not fully ready yet. At this point, loop boundary modifications are based on the information available through the IR available with Rose.

A dependency analysis was performed and we obtained the same precision for a gain in dPerf prediction greater than 1, being close to 90 (see Fig. 4b). The corresponding slowdown is presented in Fig. 4a.



(a) Example of slowdown for dPerf (b) Example of gain for dPerf

Fig. 4 The slowdown and the gain for dPerf when using simple and loop-boundaries modification

4 Experiments

This section describes in detail the experimental work done with dPerf. At first, we use a sequential implementation of the Laplace transform in order to prove the accuracy of the approach used in dPerf. We denote this scenario by *Seq. Laplace*. Our approach is afterward tested on a distributed system with low heterogeneity, denoted by *NAS IS 1*. A third validation is done on another distributed architecture highly heterogeneous referred to as *NAS IS 2*,. The input code for *NAS IS 1* and *NAS IS 2*, scenarios is the NAS Integer Sort benchmark [4, 22], and their system topologies are shown in Fig. 6 and Fig. 7, respectively. Laplace is simple sequential code, but opposed to this, NAS IS is communication-intensive code for parallel and distributed systems.

4.1 Laplace transform—sequential application

By using the C implementation of the Laplace transform [17], we show the precision of dPerf on a single process.

Reference values and the compiler We use the GCC compiler for building the code. The validation implies comparing the prediction obtained with dPerf to a reference time. This reference time is the actual execution duration of the Laplace transform on a real platform. We denote this time by t_{exec}, $t_{\text{real exec}}$, or $t_{\text{normal execution}}$. The code is build, in turn, using GCC optimization level 0, 1, 2, 3, and s. We aim at remaining accurate in our prediction when we use a simple or threshold-based block benchmarking method.

The Seq. Laplace computing system consists of one computing machine having the following characteristics: Intel Core 2 Duo @ 2.26 GHz, 3 MB cache.

Acquiring reference time begins by compiling the original Laplace code, in turn, with every relevant optimization level available in GCC and then run each binary in order to have the real execution time. The measurement of the reference time is taken with the *time* command. We store the *user* time and we compare it to the predicted time with respect to the optimization level.

Calculating the predicted time The Laplace code is passed at input to our tool dPerf. An AST is obtained and since Laplace is sequential, no communication calls exists, hence dPerf only applies the simple block benchmarking technique. Then it prepares the code for instrumentation by inserting calls to PAPI at the beginning and at the end of the Laplace transform. The static analyzer in dPerf yields a slightly modified code (see [16]). When using the optimized block benchmarking technique, dPerf applies the simple block benchmarking and, in addition, it analyzes out-most loops and tries to reduce their boundaries to a minimum threshold. This shortens the time to obtain a prediction with dPerf while taking into account the cache memory effect. This provides a good accuracy while reducing prediction cost. The resulting code is compiled, in turn, with GCC optimization levels 0, 1, 2, 3, s, and by running each binary we obtain execution traces. The traces contain execution times of instruction blocks in the Laplace code. The traces are passed to SimGrid MSG, but as the code is sequential, the network time is zero, hence the measurement of the instrumented code is equal to the prediction with dPerf.

Comparison of reference and predicted times In Fig. 5, the reference execution time is compared to a prediction obtained by dPerf with simple block benchmarking, and then by using dPerf with loop boundaries modification. Both predictions made by dPerf are close to the real execution time. This is confirmed by the corresponding error levels presented in the same figure. We state that dPerf can apply either one of the two block benchmarking techniques while preserving accuracy of the results.

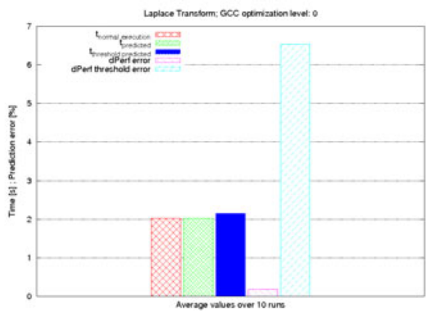
Efficiency of our approach The experiments give an accurate result due to the two block benchmarking techniques implemented in dPerf. The prediction error remains acceptable for sequential programs and the slowdown is approximately one for any optimization level used at compilation time. For this reason, the slowdown is not presented graphically. A more interesting value for slowdown is obtained for parallel and distributed applications.

4.2 NAS Integer Sort—communication-intensive, distributed application

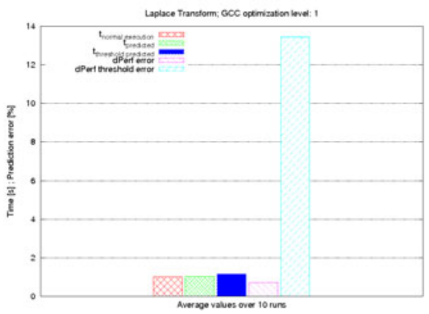
A C/MPI application is used for verifying the accuracy of dPerf on distributed systems. We present two experimental set-ups and the corresponding prediction results. Our framework is validated by two sets of experiments denoted by *NAS IS 1* and *NAS IS 2*,, the only difference between the two scenarios being the architecture heterogeneity.

Input source code The application to be evaluated and whose performance will be predicted is the Integer Sorting code of the NAS Parallel Benchmark suite. The code was written in C and the message exchange is done through MPI. This benchmark is available under a single source file. Throughout the experiments, the problem size of class A is used, this being suited for HPC systems.

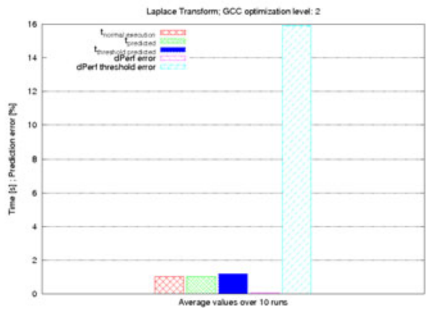
Reference values and the compiler Similar to *Seq. Laplace*, we use the GCC compiler and the reference time is measured for compiler optimization levels 0, 1, 2, 3, and s.



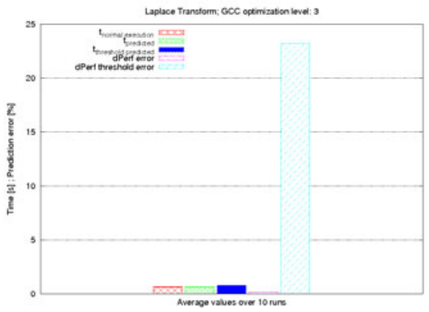
(a) Times and errors; GCC optimization 0



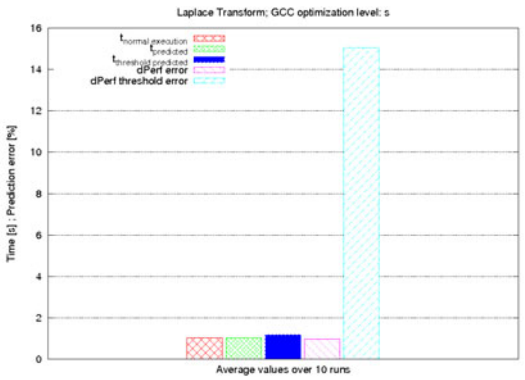
(b) Times and errors; GCC optimization 1



(c) Times and errors; GCC optimization 2



(d) Times and errors; GCC optimization 3



(e) Times and errors; GCC optimization s

Fig. 5 *Seq. Laplace*; Reference time compared to (i) prediction using the simple block benchmarking and (ii) prediction using outer-loop boundaries modification. The last two values in each figure is the prediction error

We denote the execution time of IS on a real platform by t_{exec}, $t_{\text{real exec}}$, or $t_{\text{normal execution}}$, and we use this time as a reference.

Number of computing nodes The number of computing nodes (for simplicity referred to as *nodes*) is set to 2^n. $n \in \{1, 2, 3, 4\}$, i.e., we use 2, 4, 8, and 16 nodes.

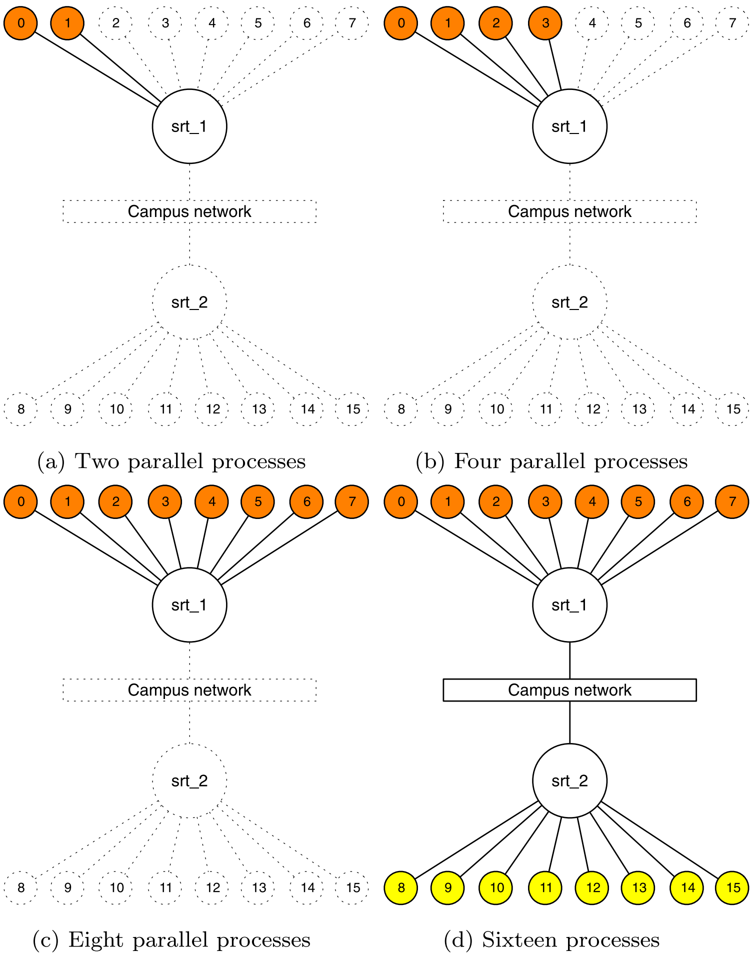
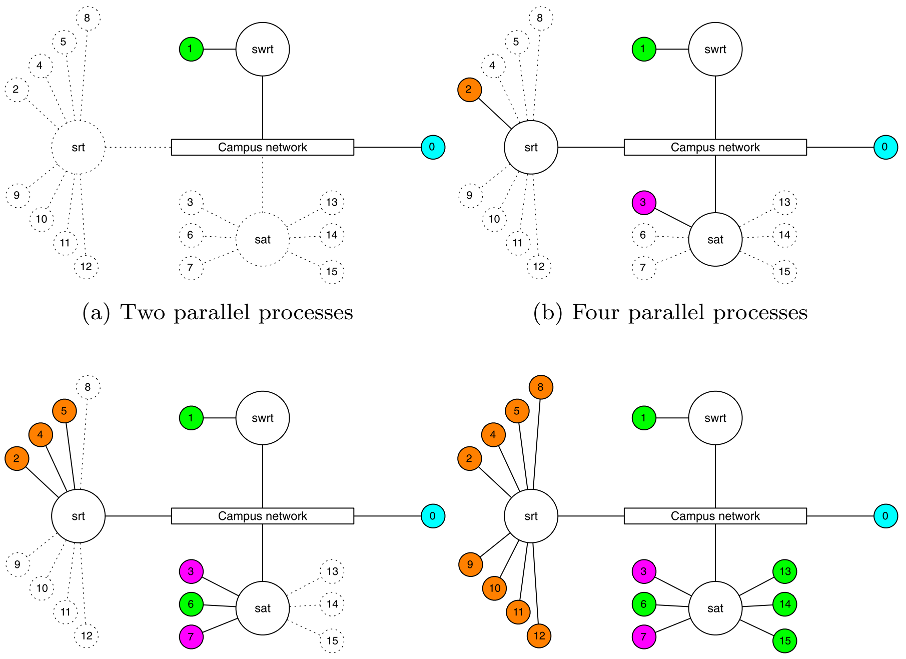

0
1
2
3
4
5
6
7
srt_1
Campus network
srt_2
8
9
10
11
12
13
14
15
(a) Two parallel processes
(b) Four parallel processes
(c) Eight parallel processes
(d) Sixteen processes

Fig. 6 *NAS IS 1* active nodes during a parallel run and the network complexity when increasing the number of nodes

NAS IS 1 computing system and network topology The experimental set-up *NAS IS 1* (see Fig. 6) consists of a heterogeneous system composed of 16 machines (or nodes) spread over two sites. The network complexity with respect to the number of parallel processes in use is detailed in Fig. 6.

– nodes 0–7 (orange in Fig. 6): Intel Pentium D @ 2.8 GHz, 1 MB cache, 1 Gbps network adapters;
– nodes 8–15 (yellow): Intel Core 2 Duo @ 2.33 GHz, 4 MB cache, 1 Gbps network adapters.

We emphasize that at this point we only consider one available processor core per machine. All communications are internode ones and as soon as we switch from 8 to 16 nodes, the architecture becomes heterogeneous. The network consists of the 16 nodes connected using:



(a) Two parallel processes (b) Four parallel processes

(c) Eight parallel processes (d) Sixteen parallel processes

Fig. 7 *NAS IS 2*, active nodes during a parallel run and the network complexity when changing the number of nodes

- *srt_1*, an HP Procurve 2848 switch, supporting 1 Gbps on each port;
- *srt_2*, a Cisco Catalyst 2900XL, with Ethernet ports of 100 Mbps;
- *Campus network*, a set of routing switches on a ring-based topology, connecting srt_1 and srt_2, with a bandwidth of 100 Mbps.

NAS IS 2, computing system and network topology The experimental set-up *NAS IS 2*, (see Fig. 7) consists of a system with a higher degree of heterogeneity than *NAS IS 1*, composed of 16 machines (nodes 0 to 15) spread over four sites. In *NAS IS 2*, the network degree of complexity reaches a maximum level from four processes and up (see Fig. 7).

- node 0 (cyan in Fig. 7): Intel Bi-Xenon @ 2.8 GHz, 512 KB cache, 3 GB RAM, 1 Gbps network adapters;
- nodes 1, 6, 13–15 (green): Intel Core 2 Duo @ 3 GHz, 6 MB cache, 1 GB RAM, 1 Gbps network adapters;
- nodes 3,7 (purple): Intel Pentium 4 @ 3 GHz, 1 GB cache, 1 GB RAM, 1 Gbps network adapters;
- nodes 2, 4, 5, 8–12 (orange): Intel Core 2 Duo @ 2.33 GHz, 4 MB cache, 1 Gbps network adapters.

As in the case of *NAS IS 1*, we only consider one available processor core per machine. The network consists of the 16 nodes connected using:

- *swrt*, Linksys WRT54GL router, with Ethernet ports of 100 Mbps;
- *srt*, FORE Systems ES 2810 switch, with Ethernet ports of 100 Mbps;
- *sat*, Allied Telesyn AT-FS708LE switch, with Ethernet ports of 100 Mbps;
- Campus network, a set of routing switches on a ring-based topology, connecting swrt, srt, sat, and node 0, with a bandwidth of 100 Mbps.

4.2.1 *Validating our framework by NAS IS 1*

In the following, we describe the method for calculating $t_{\text{predicted}}$ (or $t_{\text{simulated}}$), an estimation of IS execution time. All values used for computing the predicted time are an average of ten measurements.

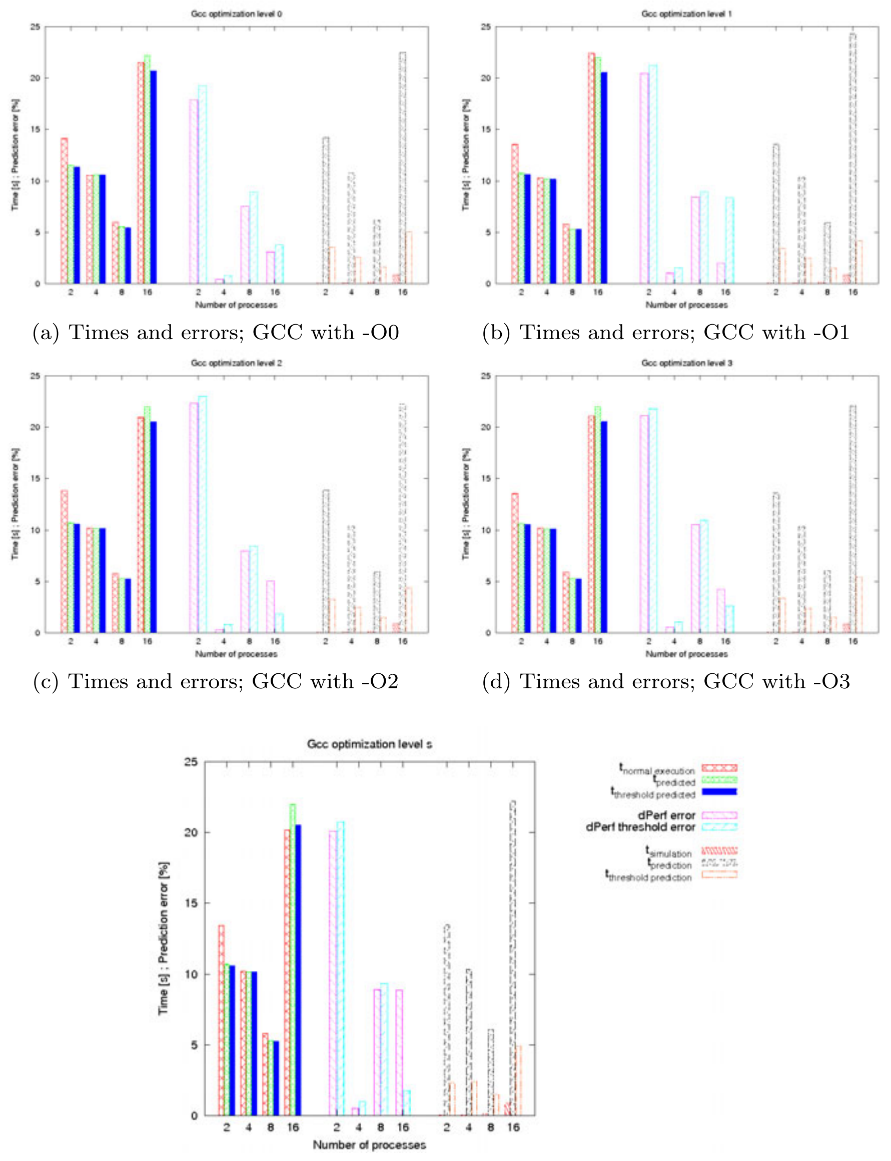
Acquiring reference time A first step is to compile the original unaltered code of IS, in turn, with every optimization level available in GCC and then run IS on 2, 4, 8, 16 nodes in order to have the real execution time. The values stored at this point are compared against the predicted time with respect to the optimization level and the number of parallel processes.

Calculating the predicted time IS is passed at input to dPerf. If dPerf is set to use the simple block benchmarking technique, then it identifies all MPI routines and prepares the blocks for instrumentation. If dPerf is configured to use the optimized block benchmarking technique, then it identifies all MPI routines and prepares the blocks for instrumentation with respect to the threshold iteration rule. When loops without communication are found, the threshold iteration (*th*) is calculated and the loop upper boundary is set to *th*. The transformed IS code obtained at the end of the static analysis is much smaller than the original IS. When blocks containing communication are found, dPerf computes *th*, then it inserts the necessary calls to the PAPI library before and after each MPI communication call.

After static analysis, the transformed IS code is compiled using each optimization level, for 2, 4, 8, 16 parallel processes, and set the problem size to $CLASS = A$. Similar to the acquisition of the reference time, the recently built variations of the transformed IS code are run and upon each execution one trace file for every parallel process is obtained. In turn, the trace files corresponding to each instrumented execution run are passed at input for SimGrid MSG. We emphasize that at this point, SimGrid's MSG module, the one responsible for the trace-replay mechanism, simulates the communication over any chosen network topology. The platform description file can be found in [8]. SimGrid MSG replays the traces and outputs the predicted time for each scenario. The results are compared in the remaining part of this section.

Comparison of reference and predicted times The first set of results to draw our attention is presented in Fig. 8. We observe that our prediction framework yields results that are very close to the actual execution time of IS; this being validated for all optimization levels. NAS IS is a communication-intensive benchmark. For this reason, the error bars in Fig. 8 show that for a low number of employed nodes (2 in our case), the error can reach levels up to 23 %. For the same reason, the prediction error decreases below 10 %, as more messages are exchanged between processes.

At this point, we can state that dPerf can apply either one of the two block benchmarking techniques while preserving accuracy of the results.



(a) Times and errors; GCC with -O0

(b) Times and errors; GCC with -O1

(c) Times and errors; GCC with -O2

(d) Times and errors; GCC with -O3

(e) Times and errors; GCC with -Os

Fig. 8 *NAS IS 1*. dPerf prediction; reference execution time; prediction error; simulation time

Efficiency of our framework For one heterogeneous cluster with fixed nodes and N different network topologies, the cost to obtain a performance prediction is

$$t_{\text{prediction}_1} = t_{\text{obtain trace files}} + t_{\text{simulation}} \tag{9}$$

for the very first topology, and

$$t_{\text{prediction}_i} = t_{\text{simulation}} \quad \mathit{with}\ i = 2..N \tag{10}$$

for the 2nd to Nth topologies, since $t_{\text{obtain trace files}}$ remains unchanged and is already known. This results in having

$$t_{\text{per prediction}} = \frac{\sum_{j=1}^{N} t_{\text{prediction}_j}}{N} \tag{11}$$

From formulae (9), (10), and (11) we obtain

$$t_{\text{per prediction}} = \frac{t_{\text{obtain trace files}} + t_{\text{simulation}} \times N}{N} \tag{12}$$

$$t_{\text{per prediction}} = \frac{t_{\text{obtain trace files}}}{N} + t_{\text{simulation}} \tag{13}$$

Figure 8 denotes the importance of using execution and trace-based simulation. A first experiment with the given input parameters is represented in the figure by $t_{\text{prediction}}$ bars. If we want a prediction for the same input parameters but for a different network configuration, then we rerun the trace-based simulation. This operation is represented by $t_{\text{simulation}}$ in Fig. 8. The change in network topology and the rerun of the simulation output a completely new prediction result with an insignificant time cost. The more we test other network topologies, the more $t_{\text{per prediction}}$ decreases.

Figure 9 shows that dPerf has a gain instead of a slowdown. Both block benchmarking methods implemented in dPerf work with a slowdown inferior to *1.1*. The gain corresponding to dPerf predictions at each relevant optimization level in GCC is, on average, between 10 and 65, knowing that a gain of 1 means a prediction cost equal to a normal execution cost.

When choosing one method or another, dPerf searches for the outer most loops with independent iteration variables. If such loops are found, the optimized block benchmarking is applied. If not, then dPerf uses simple block benchmarking.

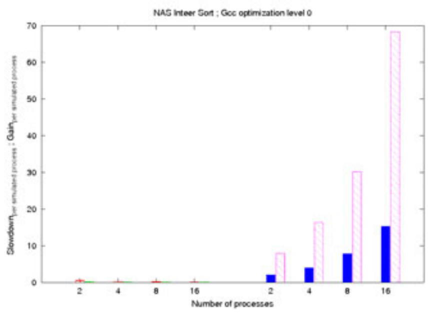
4.2.2 Validating our framework by NAS IS 2,

The validation steps were previously explained for *NAS IS 1* and, therefore, we only present the second set of results which corresponds to the second experimental set-up (see Fig. 7).

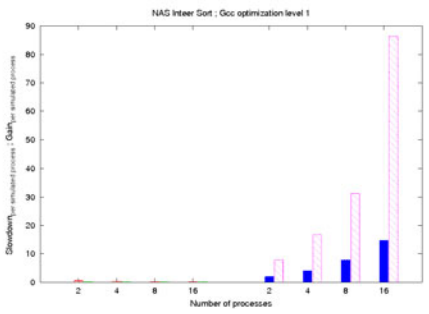
Comparison of reference and predicted times These results are of particular interest due to the high heterogeneity of the system and network. The system has a medium complexity when IS is executed on two processes (see Fig. 7a), and becomes of high complexity as soon as IS starts turning on four processes or more, as seen in Fig. 7(b, c, d). The first results depicted in Fig. 10 show the prediction obtained with dPerf with respect to the reference execution time $t_{\text{normal execution}}$, and the error in prediction. The accuracy in prediction has a slightly different behavior than the one in *NAS IS 1* due to the increased heterogeneity level of the system. However, the predicted time yields a smaller error as the number of processes increases above the value four, meaning that our estimation becomes more accurate for 8 and 16 processes.

The accuracy in the case of *NAS IS 1* is high, but the precision of our framework for *NAS IS 2*, also remains high, for all optimization levels, given its degree of complexity (see Fig. 7).

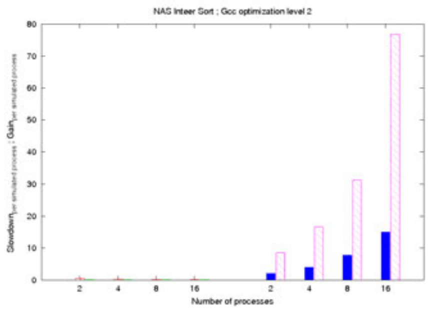
For *NAS IS 2*,, the comparison between reference, prediction, and threshold-prediction times is depicted in Fig. 10.



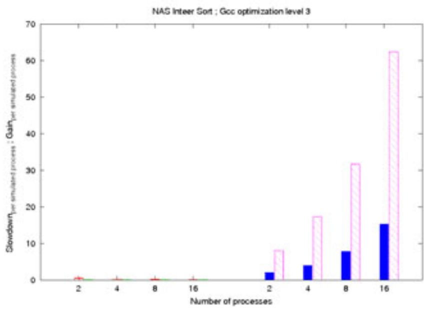
(a) Slowdown and gain; GCC -O0



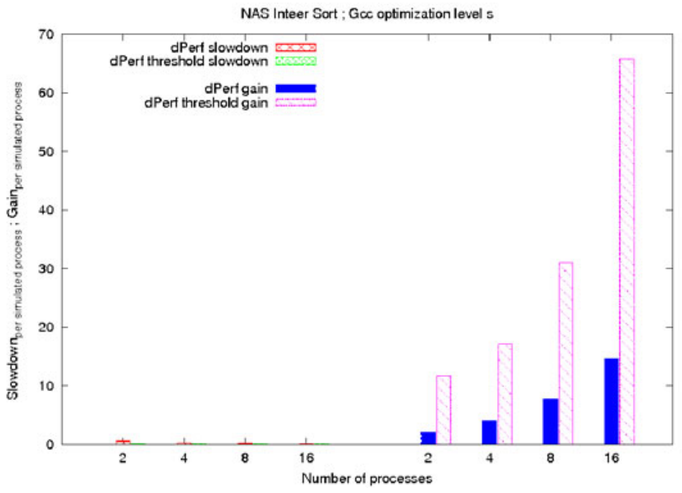
(b) Slowdown and gain; GCC -O1



(c) Slowdown and gain; GCC -O2



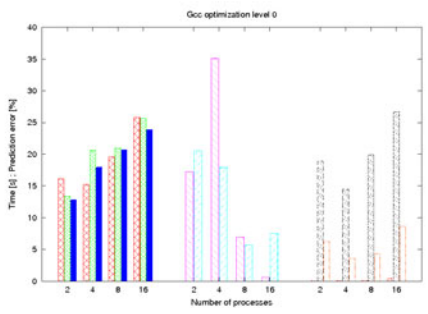
(d) Slowdown and gain; GCC -O3



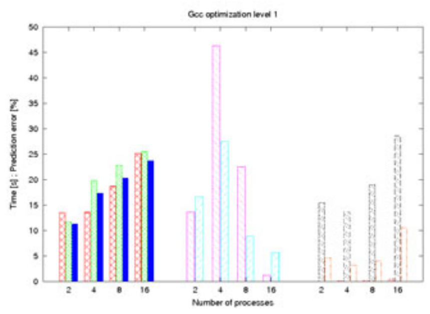
(e) Slowdown and gain; GCC -Os

Fig. 9 *NAS IS 1*; Slowdown and gain; The legend of (**e**) also applies to figures (**a** to **d**)

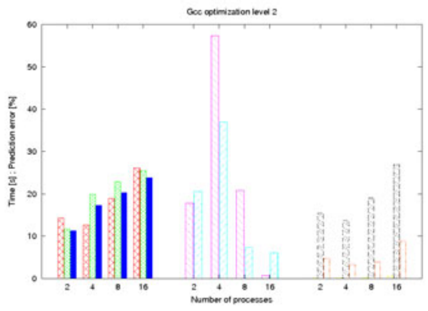
Efficiency of our framework For the experimental set-up *NAS IS 2*,, we calculated the slowdown for a simple prediction with dPerf and compared it to the prediction when dPerf uses loop-boundaries modification. The slowdown varies from one com-



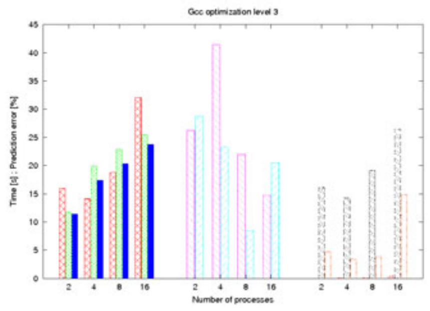
(a) Times and errors; GCC with -O0



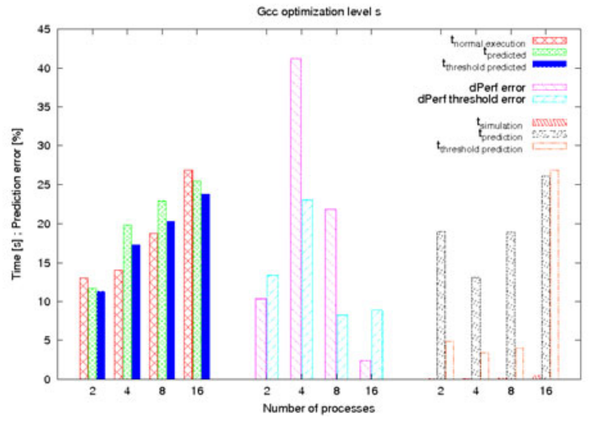
(b) Times and errors; GCC with -O1



(c) Times and errors; GCC with -O2



(d) Times and errors; GCC with -O3

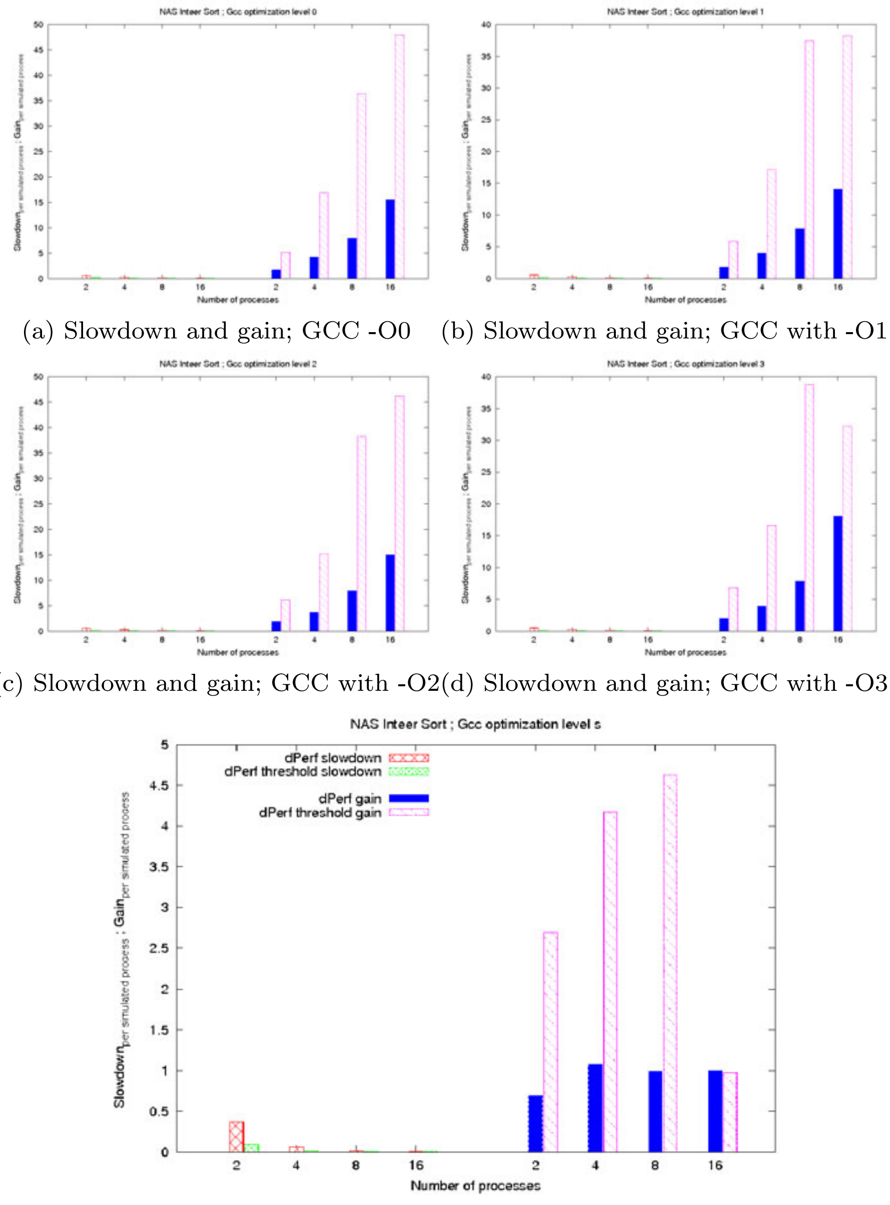


(e) Times and errors; GCC with -Os

Fig. 10 Results for *NAS IS 2*,. The legend shown in figure (**e**) also applies to figures (**a** to **d**)

piler optimization to another, as it can be seen in Fig. 11. The slowdown was calculated according to formulae (5) to (8).

For all experimental results presented above, we state that our performance prediction method yields accurate results with a time cost that decreases proportionally to the number of network topologies tested.



(a) Slowdown and gain; GCC -O0 (b) Slowdown and gain; GCC with -O1

(c) Slowdown and gain; GCC with -O2(d) Slowdown and gain; GCC with -O3

(e) Slowdown and gain; GCC with -Os

Fig. 11 The slowdown and gain for *NAS IS* 2,. The legend of figure (**e**) also applies to figures (**a** to **d**)

5 Conclusion and future work

In this paper, we presented our approach for predicting performance for distributed applications running in a heterogeneous environment. This approach was imple-

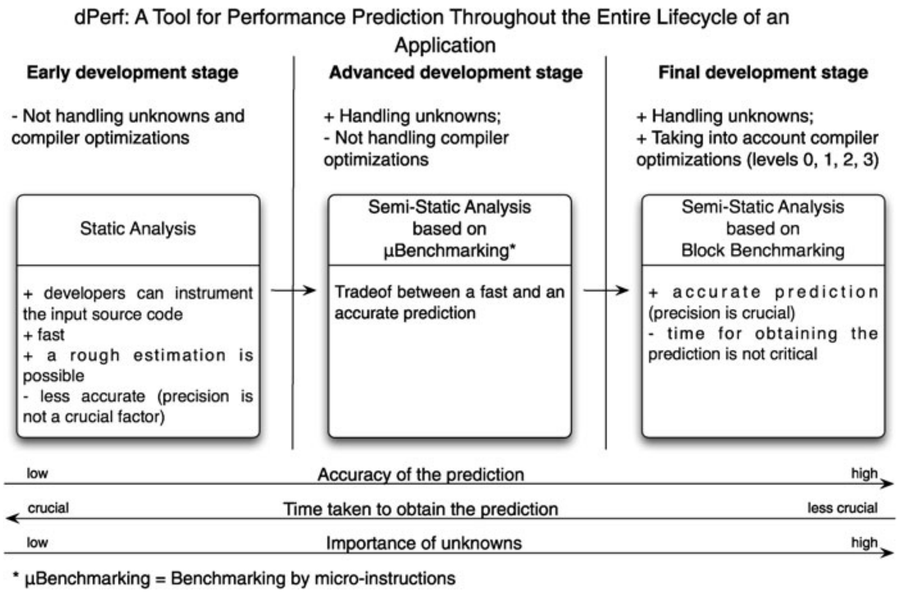

dPerf: A Tool for Performance Prediction Throughout the Entire Lifecycle of an Application
Early development stage
- Not handling unknowns and compiler optimizations
Static Analysis
+ developers can instrument the input source code
+ fast
+ a rough estimation is possible
- less accurate (precision is not a crucial factor)
Advanced development stage
+ Handling unknowns;
- Not handling compiler optimizations
Semi-Static Analysis based on µBenchmarking*
Tradeof between a fast and an accurate prediction
Final development stage
+ Handling unknowns;
+ Taking into account compiler optimizations (levels 0, 1, 2, 3)
Semi-Static Analysis based on Block Benchmarking
+ accurate prediction (precision is crucial)
- time for obtaining the prediction is not critical
low Accuracy of the prediction high
crucial Time taken to obtain the prediction less crucial
low Importance of unknowns high
* µBenchmarking = Benchmarking by micro-instructions

Fig. 12 Performance prediction at any point in an application life-cycle

mented in dPerf, having three main components: (i) the automatic static analyzer, which uses our techniques for benchmarking instruction blocks in a simple or optimized manner, (ii) the runtime dependency solver through execution, and (iii) the trace-based simulator which computes the performance prediction. The current development state and the accuracy of our method was tested using NAS Integer Sort.

The approach presented is a continuous effort to obtain a performance prediction method with scalable and architecture-independent results. We grant special attention to the use of the System Dependence Graph, a representation that we intend to exploit in such a manner as to entirely solve all data-dependencies. Our near-future development plans aim at reducing as much as possible the dependence of dPerf predictions on the computing system. We intend for dPerf to provide prediction results throughout the entire development, thus the entire life-cycle, of a distributed application (see Fig. 12). Regarding network simulation, we are interested in adding P2P support to SimGrid MSG and estimate the performance of distributed applications in the P2P environment. We are constantly looking into related work in order to improve our method and to integrate support for multicore machines. We envisage analyzing the source code in assembler, inspired by the work of [20, 21, 33] to provide scalable architecture-independent traces. Our model could considerably increase its precision, and it would be an important feature for dPerf when predicting performance for regular and P2P systems.

Acknowledgements This work is funded by the French National Agency for Research under the ANR-07-CIS7-011-01 contract [2].

References

1. Adve VS, Bagrodia R, Browne JC, Deelman E, Dube A, Houstis EN, Rice JR, Sakellariou R, Sundaram-Stukel DJ, Teller PJ, Vernon MK (2000) POEMS: End-to-end performance design of large parallel adaptive computational systems. IEEE Trans Softw Eng 26:1027–1048
2. ANR CIP project web page. http://spiderman-2.laas.fr/CIS-CIP
3. Badia RM, Escalé F, Gabriel E, Gimenez J, Keller R, Labarta J, Müller MS (2004) Performance prediction in a grid environment. In: Grid computing. Lecture notes in computer science, vol 2970. Springer, Berlin/Heidelberg, pp 257–264
4. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS, Simon HD, Venkatakrishnan V, Weeratunga SK (1991) The NAS parallel benchmarks—summary and preliminary results. In: SC'91: proceedings of the 1991 ACM/IEEE conference on supercomputing. ACM Press, New York, pp 158–165
5. Bourgeois J, Spies F (2000) Performance prediction of an NAS benchmark program with Chronos-Mix environment. In: Euro-Par'00: the 6-th international Euro-Par conference on parallel processing. Springer, Berlin, pp 208–216
6. Casanova H, Legrand A, Quinson M (2008) SimGrid: a generic framework for large-scale distributed experiments. In: UKSIM'08: proceedings of the 10th int conference on computer modeling and simulation. IEEE Computer Society, Los Alamitos, pp 126–131
7. Cornea BF, Bourgeois J (2011) Performance prediction of distributed applications using block benchmarking methods. In: PDP'11, 19-th int Euromicro conf on parallel, distributed and network-based processing. IEEE Computer Society, Los Alamitos
8. Cornea B, Bourgeois J (2012) http://lifc.univ-fcomte.fr/page_personnelle/recherche/136
9. Culler D, Karp R, Patterson D, Sahay A, Schauser KE, Santos E, Subramonian R, von Eicken T (1993) LogP: towards a realistic model of parallel computation. ACM Press, New York, pp 1–12
10. El Baz D, Nguyen TT (2010) A self-adaptive communication protocol with application to high performance peer to peer distributed computing. In: PDP'10: proceedings of the 18th Euromicro conference on parallel, distributed and network-based processing. IEEE Computer Society, Los Alamitos, pp 327–333
11. Ernst-Desmulier JB, Bourgeois J, Spies F, Verbeke J (2005) Adding new features in a peer-to-peer distributed computing framework. In: PDP'05: proceedings of the 13th Euromicro conference on parallel, distributed and network-based processing. IEEE Computer Society, Los Alamitos, pp 34–41
12. Ernst-Desmulier JB, Bourgeois J, Spies F (2008) P2pperf: a framework for simulating and optimizing peer-to-peer-distributed computing applications. Concurr Comput 20(6):693–712
13. Fahringer T (1996) On estimating the useful work distribution of parallel programs under the P3T: a static performance estimator. Concurr Pract Exp 8:28–32
14. Fahringer T, Zima HP (1993) A static parameter based performance prediction tool for parallel programs. In: ICS'93: proceedings of the 7th international conference on supercomputing. ACM Press, New York, pp 207–219
15. Finney SA (2001) Real-time data collection in Linux: a case study. Behav Res Methods Instrum Comput 33:167–173
16. Laplace transform instrumented with dPerf; simple block benchmarking method. http://bogdan.cornea.perso.neuf.fr/files/journal_files/laplace_dperf.c
17. Laplace transform. http://www.physics.ohio-state.edu/~ntg/780/c_progs/laplace.c
18. Li J, Shi F, Deng N, Zuo Q (2009) Performance prediction based on hierarchy parallel features captured in multi-processing system. In: HPDC'09: proc of the 18th ACM int symposium on high performance distributed computing. ACM Press, New York, pp 63–64
19. Livadas PE, Croll S (1994) System dependence graphs based on parse trees and their use in software maintenance. Inf Sci 76(3–4):197–232
20. Marin G (2007) Application insight through performance modeling. In: IPCCC'07: proceedings of the performance, computing, and comm. conf. IEEE Computer Society, Los Alamitos
21. Marin G, Mellor-Crummey J (2004) Cross-architecture performance predictions for scientific applications using parameterized models. In: SIGMETRICS'04/Performance'04: proceedings of the joint international conference on measurement and modeling of computer systems. ACM Press, New York, pp 2–13
22. NAS parallel benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html
23. Nguyen TT, El Baz D, Spiteri P, Jourjon G, Chau M (2010) High performance peer-to-peer distributed computing with application to obstacle problem. In: IPDPSW'10: IEEE international symposium on parallel distributed processing, workshops and Phd forum, pp 1–8

24. Noeth M, Marathe J, Mueller F, Schulz M, de Supinski B (2006) Scalable compression and replay of communication traces in massively parallel environments. In: SC'06: proceedings of the 2006 ACM/IEEE conference on supercomputing. ACM Press, New York, p 144
25. PAPI project website. http://icl.cs.utk.edu/papi/
26. PAPI SC2008 handout. http://icl.cs.utk.edu/graphics/posters/files/
27. Perfmon project webpage. http://perfmon2.sourceforge.net/
28. Pettersson M (2012) Perfctr project webpage. http://user.it.uu.se/~mikpe/linux/perfctr/
29. Prakash S, Bagrodia RL (1998) MPI-SIM: using parallel simulation to evaluate mpi programs. In: WSC'98: proceedings of the 30th conference on winter simulation. IEEE Computer Society Press, Los Alamitos, pp 467–474
30. Rose LD, Poxon H (2009) A paradigm change: from performance monitoring to performance analysis. In: SBAC-PAD, pp 119–126
31. Saavedra RH, Smith AJ (1996) Analysis of benchmark characteristics and benchmark performance prediction. ACM Trans Comput Syst 14(4):344–384
32. Schordan M, Quinlan D (2003) A source-to-source architecture for user-defined optimizations. In: Modular programming languages. Lecture notes in computer science, vol 2789. Springer, Berlin/Heidelberg, pp 214–223
33. Skinner D, Kramer W (2005) Understanding the causes of performance variability in HPC workloads. In: IEEE workload characterization symposium, pp 137–149
34. Snavely A, Wolter N, Carrington L (2001) Modeling application performance by convolving machine signatures with application profiles. In: WWC'01: IEEE international workshop on workload characterization. IEEE Computer Society, Los Alamitos, pp 149–156
35. Sundaram-Stukel D, Vernon MK (1999) Predictive analysis of a wavefront application using LogGP. In: 7th ACM SIGPLAN symposium on principles and practice of parallel programming, vol 34(8). ACM Press, New York, pp 141–150
36. The message passing interface standard. http://www-unix.mcs.anl.gov/mpi
37. van Gemund AJC (2003) Symbolic performance modeling of parallel systems. IEEE Trans Parallel Distrib Syst 14(2):154–165
38. Zaparanuks D, Jovic M, Hauswirth M (2009) Accuracy of performance counter measurements. In: ISPASS'09: IEEE international symposium on performance analysis of systems and software, pp 23–32
39. Zhai J, Chen W, Zheng W (2010) Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In: PPoPP'10: proceedings of the 15th ACM SIGPLAN symposium on principles and practice of parallel programming. ACM Press, New York, pp 305–314