Cluster Comput (2016) 19:1105–1137
DOI 10.1007/s10586-016-0615-4

Automated and dynamic abstraction of MPI application performance

Anna Sikora[1] · **Tomàs Margalef**[1] · **Josep Jorba**[2]

Received: 25 September 2015 / Revised: 22 April 2016 / Accepted: 3 August 2016 / Published online: 19 August 2016

Abstract Developing an efficient parallel application is not an easy task, and achieving a good performance requires a thorough understanding of the program's behavior. Careful performance analysis and optimization are crucial. To help developers or users of these applications to analyze the program's behavior, it is necessary to provide them with an abstraction of the application performance. In this paper, we propose a dynamic performance abstraction technique, which enables the automated discovery of causal execution paths, composed of communication and computational activities, in MPI parallel programs. This approach enables autonomous and low-overhead execution monitoring that generates performance knowledge about application behavior for the purpose of online performance diagnosis. Our performance abstraction technique reflects an application behavior and is made up of elements correlated with high-level program structures, such as loops and communication operations. Moreover, it characterizes all elements with statistical execution profiles. We have evaluated our approach on a variety of scientific parallel applications. In all scenarios, our online performance abstraction technique proved effective for low-overhead capturing of the program's behavior and facilitated performance understanding.

✉ Anna Sikora
anna.sikora@uab.cat

Tomàs Margalef
tomas.margalef@uab.cat

Josep Jorba
jjorbae@uoc.edu

[1] Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain

[2] Estudis d'Informàtica,Multimèdia i Telecomunicació, Universitat Oberta de Catalunya, 08018 Barcelona, Spain

Keywords Online performance abstraction · Performance metrics · Automatic performance analysis · Parallel applications

1 Introduction

Developing an efficient parallel application is not an easy task, and achieving a good performance requires a thorough understanding of the program's behavior. In practice, developers must understand both the application and the behavior of the environment. They must often focus more on the resource usage, communication, synchronization and other low-level issues, than on the real problem being solved. There are still several challenges that significantly complicate performance diagnosis of parallel applications. Careful performance analysis and optimization are crucial. There are many tools that assist developers in the process of performance analysis and the detection of performance problems [14]. Graphical trace browsers, such as Paraver [36] or the methodology for interactive 3D vision incorporated in the TAU ParaProf tool [44], offer fine-grained performance metrics and visualizations. However, their accurate interpretation requires a substantial amount of time and effort from highly skilled analysts. Other tools automate the identification of performance bottlenecks and their location in a source code. KappaPI 2 [18], EXPERT [50] and Scalasca [26] perform offline analysis of event traces searching for patterns that indicate inefficient behavior. Paradyn [28] uses runtime code instrumentation to find the parts of a program which contribute significantly to its execution. Periscope [13] uses the agent hierarchy that searches for inefficiencies of large-scale parallel applications based on the APART language.

Although these tools greatly support developers in understanding what is happening and when, they do not automate

the inference process in order to find the root causes of the performance problems. Isolating the root causes is still a manual process that requires a substantial amount of time, effort and skills from qualified professionals. It is crucial to automate the inference process of finding the causes of performance problems by taking advantage of expert information on the operation semantics and parallelism found in parallel computations. As an initial attempt to find problem causes, we propose an online performance abstraction technique which allows for the automated runtime discovery of causal execution paths (instances of execution flows that include contextual data and execution profile), made up of communication and computational activities in message-passing parallel programs preserving happened-before relationships. The application structure and its behavior is automatically abstracted during the execution, and the resulting abstraction can be used for automated and dynamic root-cause analysis [32] or tuning [31].

By following the flow of control and intercepting communication between MPI processes at runtime, the corner stone of this technique is the ability to reflect the application behavior in a compact manner. The resulting abstraction is composed of high-level application structures, such as loops and communication operations and characterizes them with statistical execution profiles. It facilitates the understanding of high-level program behavior and enables an assortment of online diagnosis techniques. Our technique can be deployed in a wide range of unmodified MPI applications with acceptable overhead and scales to thousands of processors. We have evaluated our approach in a variety of scientific parallel applications. In all scenarios, our online performance abstraction technique proved effective for low-overhead capturing of program behavior and facilitated performance understanding.

The remainder of this paper is organized as follows. Section 2 defines the basis that was used for online performance abstraction. Section 3 presents our approach to online performance abstraction. Section 4 describes the prototype tool implementation which can automatically abstract an arbitrary MPI application during runtime. Section 5 presents the results of the experimental evaluation of our tool in real-world applications. The related work is reviewed in Sect. 6. Finally, Sect. 7 concludes the work and points out directions for future research.

2 Definition of online performance abstraction

The goal of our approach is to reflect MPI application behavior by abstracting execution flows through high-level program structures, such as loops and communication operations and characterizing them with statistical execution profiles. We arrange selected primitive events into concepts called activities and then track flows through these activities. The resulting abstraction captures a roadmap of executed flows together with aggregated performance profiles. It gives a compact view of the application behavior, preserving the structure and causality relationships. It then enables a quick online analysis that determines the parts of the program most relevant to its performance.

Our technique is hybrid because it combines features of static and dynamic analysis methods. We perform offline analysis of a binary executable, discover static code structure and instrument heuristically selected loops to detect cycle boundaries. At runtime, we perform selective and incremental event tracing and the aggregation of executed flows of activities. We consider that our technique can be valuable for both non-experienced and expert users. It may ease and shorten the performance understanding process and serve as a base for developing online performance analysis.

The presented performance abstraction technique provides the following three features and improvements:

- It generates event traces, but it consumes them in place, creating performance profiles so that trace files are no longer needed. Our approach for bottleneck identification is equivalent to an automated interpretation of call-path profiles. The basic difference from profiling tools is that classical profiling is function-oriented, while our technique could be thought of as flow profiling, as we analyze profiles of activities that might represent single code blocks or span multiple functions.
- It captures the whole application structure with all the executed paths, together with their statistical execution profiles and aggregating repetitions. Our approach for problem identification is distributed; first, we analyze individual MPI processes independently, and then we merge the results to reflect whole application behavior in a scalable fashion.
- There are tools that automate the identification of performance bottlenecks and their location in the source code. They help a developer in understanding what happens, where, and when, but they do not automate the inference process to find the causes of performance problems. And detecting a bottleneck in a certain place does not indicate why it happens. Only when the root causes of a performance problem are correctly identified is it possible to provide effective solutions to fix or alleviate the issue. Our approach allows for the detection of program phases, the clustering of MPI processes by their behavior, detecting load imbalance by matching loops between communicating processes and comparing their profiles, as well as other observations. Certain properties of our abstraction, such as the causal relationships between activities, can be used to develop tools for root-cause performance problem diagnosis.

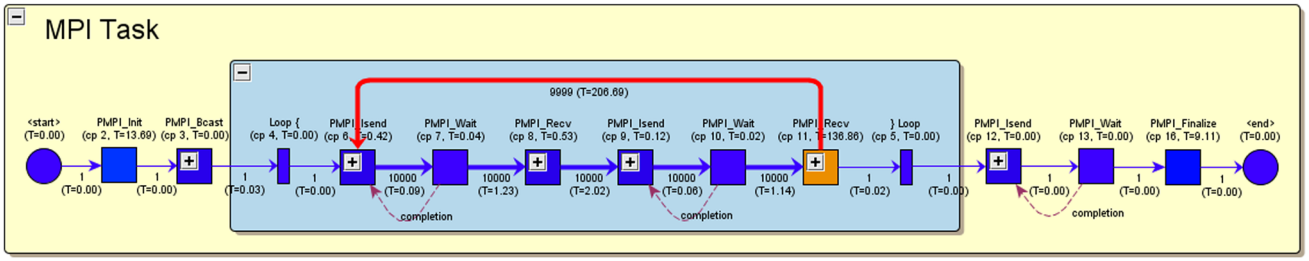

Fig. 1 An example TAG of a single MPI process (*MPI Task*). We use the spectrum of rainbow colors to reflect the relevance of each activity with respect to total time. This graph was generated with yEd

2.1 Abstracting individual MPI process

To abstract a process of a parallel application, we define a *task activity graph (TAG)*, a directed graph that abstracts the execution of a single process. The execution is described by units that correspond to different activities and their causal relationship. We distinguish between two main types of activities: a *communication activity* reflects some message-passing communication operation, and a *computation activity* is a group of statements belonging to the same process, none of which are communication operations. The communication activities performed by a process are abstracted as graph nodes, while computation activities are represented by edges. Additionally, to reflect the semantics of non-blocking operations, we introduce a completion edge that connects the completion node with its start node in the non-blocking operation. Finally, we use *control activities*, represented as nodes, to abstract the punctual program's control flow events, such as process start/termination events, and entry/exit events of selected loops.

The execution of a particular communication activity is identified by an ordered pair of events: *entry* and *exit*, for example entry/exit events of an *MPI_Bcast* call. The end of a communication activity and the start of the consecutive one identify the execution of a computation activity, local to a process. In effect, the sequential flow of execution of a single process, reflected by temporal event ordering, ensures a happens-before relationship between the consecutive activities. Figure 1 shows an example TAG for a single process of a sample MPI application. All graphs were generated post-mortem using the yEd editor [51]. Rainbow colors are used to indicate if a node or edge is a hot/cold activity depending on the percentage of its total execution time, as presented in Fig. 2.

Fig. 2 *Colors* used for indicating the influence of nodes and edges on the application execution time (Color figure online)

We describe the behavior of program activities with execution profiles by adding performance data to nodes and edges. We instrument MPI calls using the MPI profiling interface, so MPI calls are wrapped and then executed through the use of PMPI routines that are actually instrumented. As each activity might be executed multiple times, we aggregate this performance data into statistical metrics. We use two basic metrics for all nodes and edges: a timer that measures the accumulated virtual time and a counter that counts the number of executions. We also calculate min, max, and stddev metrics to statistically track variations. Moreover, the abstraction allows for the addition and removal of arbitrary performance metrics to any activity. This enables us to control what performance data we collect for selected nodes and edges in function of their runtime behavior.

Figure 3 shows details of the visualization of a fragment of a TAG. We can observe a loop composed of a unique execution path (although there could have been various conditional paths). In every iteration, the application executes a sequence of non-blocking send and blocking receive twice, interleaved with some minor calculations, and finally performs some more time-consuming calculations at the end of the iteration. The visualization also shows the performance data. For each edge, we may observe a total number of executions (e.g., 9999) and the accumulated time spent executing the edge. For nodes, the visualization shows the unique identifier (cp) and accumulated node times.

2.2 Abstracting inter-process communications

The TAG reflects all message-passing calls as communication activities. The graph contains nodes for each communication activity (e.g., *MPI_Send, MPI_Recv*), message edges between corresponding send and receive pairs for point-to-point and collective communications, as well as flow edges between nodes that are in consecutive execution order.

To abstract all communications, we intercept the communication routines and capture call attributes including the type of call, source and destination, and other parameters, such as message size. To identify a message edge, we must determine both communicating processes and the particular

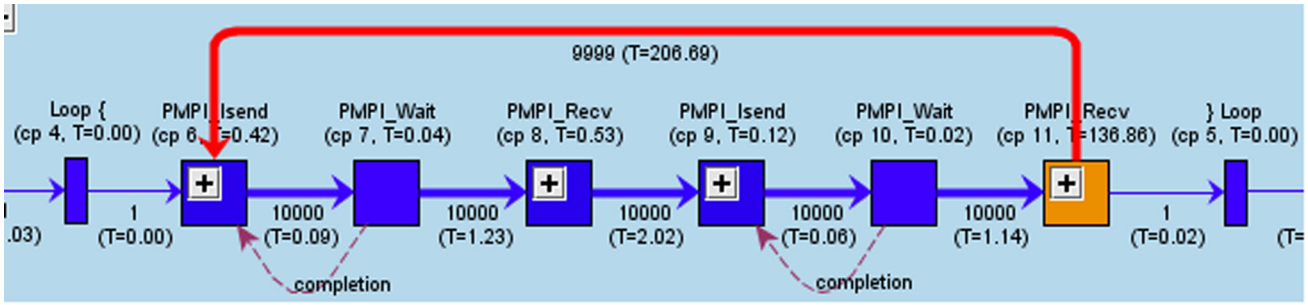

Fig. 3 Visualization of performance data in a TAG

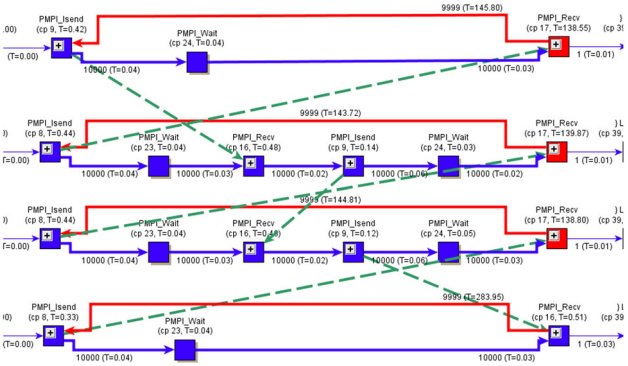

Fig. 4 MPI communication abstraction. *Dashed lines* denote transmitted messages

send and receive activities in those processes. The key idea is to match online a sender call context represented by a node in the sender TAG with a receiver call context represented by a corresponding receive node in the receiver TAG. To accomplish this goal, we piggyback the additional data from the sender to the receiver(s) in every message. We transmit the current send node identifier and store it in the matching receive node as the incoming message edge. This feature enables us to logically connect TAGs, while keeping them distributed. Finally, in order to capture communication profiles, we track the count and size histograms individually for each message edge.

We illustrate the technique in Fig. 4. This example shows a fragment of an execution of an SPMD application composed of four parallel processes that communicate using MPI. The processes are organized in a 1-D mesh topology. Each process executes a given number of iterations of a loop (10,000 in this example). In every iteration, each process exchanges messages with all of its neighboring processes. Our technique detects all communication activities in each process (*MPI_Isend, MPI_Wait* and *MPI_Recv* calls represented as nodes) and tracks executed message edges by correlating sender nodes with their matching receiver nodes.

2.3 Abstracting an entire application

To abstract the execution of an entire parallel application, we merge individual TAGs into a new global graph we will refer to as the *parallel task activity graph (PTAG)*. This merge process can be performed periodically, on demand, or at the end of an execution.

The process is straightforward, as we take advantage of the information stored in the message edges. Each incoming message edge contains data that uniquely identifies sender processes and corresponding send nodes. For point-to-point calls, the edge stores the individual sender data. For collective calls (e.g., *MPI_Gather*, *MPI_Alltoall*), the edge stores a vector of pairs that identifies all sending processes and their corresponding node identifiers.

Figure 5 shows a visualization of an example PTAG for an SPMD application. We may observe four TAGs, where each MPI process is represented by a horizontal line (flow). The

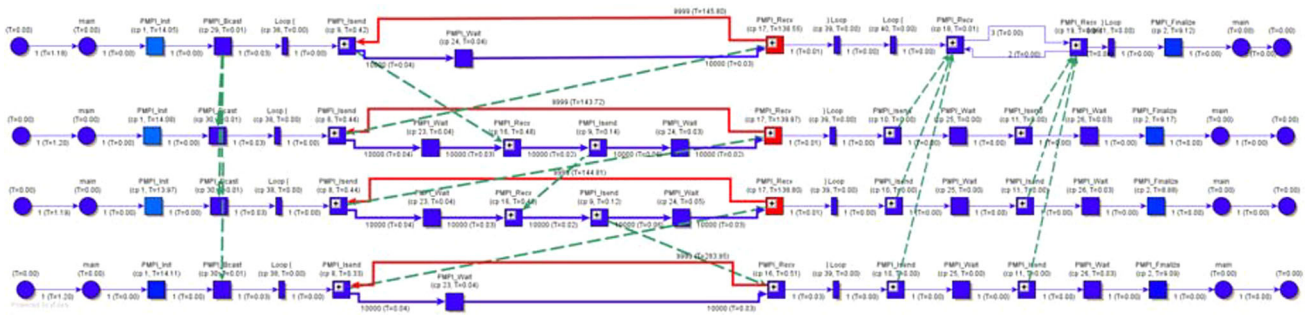

Fig. 5 Parallel task activity graph of a sample MPI application composed of 4 processes

processes intercommunicate, which is reflected with dashed arrows. As explained before, a different color is used to highlight the nodes and edges that take up the highest percentage of total execution time.

2.4 Abstraction definition

Our approach to performance analysis of an individual MPI process is based on a TAG that represents a collection of logically-dependent local events. TAG is extended to the PTAG, which combines sequential abstractions of all individual processes, reflecting exchanged message-passing communications. To formalize our approach, we introduce the following definitions.

Definition 1 (*Call-path*) It determines a sequence of active subroutines that have been called to reach a particular point of execution in a program. The active subroutines are those which have been called, but have not yet completed execution by returning. Call-paths allow a user to identify and characterize performance problems that depend on where an instruction (subroutine) is called from, rather than on the instruction (subroutine) itself. In our approach, call-paths help to distinguish between different instances of the same activities, such as sending or receiving a message. The reason is that an activity may finish quickly for most invocations, but require excessive time when executed along a particular call-path. Obtaining an active call-path during execution from within the program requires the reading of the stack frames from the call stack.

Definition 2 (*Event*) We define an event as an execution instance of a particular instruction in the application when a specified set of constraints is satisfied. The constraints may require the instruction to be executed in a determined call-path, in a determined program state (e.g., a particular variable must have a determined value) or both.

Definition 3 (*Activity*) We define an activity as a unit of execution of a process. Each activity corresponds to a set of instructions that are grouped together to perform a specific action. An activity may correspond to the execution of a sequence of basic blocks in the program or more particularly to the execution of a specific subroutine. An activity is identified by an ordered pair of events: e_{entry} (entry event) and e_{exit} (exit event).

Definition 4 (*Control activity*) It is a subclass of activity that is used to represent a set of control events in the program execution, such as program start and termination, as well as loop start and exit. Control activities are punctual, that is $e_{entry} == e_{exit}$.

Definition 5 (*Communication activity*) Our approach assumes that processes communicate by passing messages. The communication activity represents an act of sending a message, receiving a message or synchronizing with other processes (e.g., barrier). The communication activity might be blocking (it does not terminate until completed) or non-blocking (the activity ends before the operation is completed). We identify communication activities by means of a pre-defined set of subroutines. For example, entry and exit events of an MPI_Send subroutine define the scope of a communication activity.

Definition 6 (*Computation activity*) It is a class of activities that corresponds to a group of statements belonging to the same process, none of which are either communication activities or control activities. Informally, this may include CPU-bound calculations, I/O operations or any other kind of activity. We classify all activities that execute between communication or control activities as a computation activity. For example, the exit event of an MPI_Send subroutine and the entry event to the consecutive communication activity define the scope of a computation activity.

Definition 7 (*Matching communication activity*) For each message-passing activity, we assume the existence of a send communication activity snd_p in process p that transmits the message m, and matching receive communication activity rcv_q in process q that receives message m. The relationship between snd_p and rcv_q can be one-to-one (e.g., send-receive pair), but also one-to-many (e.g., broadcast defines a relationship between one sender snd_p and multiple receivers rcv_{1-N}.), many-to-one (e.g., gather data from multiple senders in a single receiver), many-to-many (e.g., all-to-all collective calls).

Definition 8 (***Causal relationship***) We assume a causal relationship of two events in a parallel program when they are in a *happened-before relationship*. The happened-before relationship (denoted by $\longmapsto$) formulated by Lamport [22] is formally defined as:

- If events e_1 and e_2 occur in the same process, $e_1 \longmapsto e_2$ if the occurrence of event e_1 preceded the occurrence of event e_2.
- If event e_1 is the sending of a message and event e_2 is the reception of the message sent in event e_1, $e_1 \longmapsto e_2$.

This relationship is transitive, that is for three events e_1, e_2, and e_3, if $e_1 \longmapsto e_2$ and $e_2 \longmapsto e_3$, then $e_1 \longmapsto e_3$.

We extend this definition to the causal relationship of activities. An activity u happened before an activity v if and only if the following conditions hold:

- Events u_{entry} and u_{exit} that identify activity u are in a happened-before relationship $(u_{entry} \longmapsto u_{exit})$

- Events v_{entry} and v_{exit} that identify activity v are in a happened-before relationship ($v_{entry} \longmapsto v_{exit}$)
- Events u_{exit} and v_{entry} are in a happened-before relationship ($u_{exit} \longmapsto v_{entry}$)

Definition 9 (*Performance metric*) It is a standard of the measurement of some performance characteristics of a function, activity, process or system. Example metrics include the number of occurrences of some events or activities, wall-clock time elapsed between two events, etc.

Definition 10 (***Execution profile***) It is a measure of performance characteristic (behavior) associated with a particular function, activity or process. Execution profile is expressed by means of a set of statistically aggregated performance metrics.

Definition 11 (***Task activity graph***) We define the task activity graph (TAG) of a program execution to be a DAG *(N, E)* where N is a set of nodes and $E \subset NxN$ is a set of edges. The set of nodes N represents the execution of all selected activities in a process. Each particular activity executed from a distinct call-path is represented exactly by a single node in the TAG, even if it is executed multiple times. Multiple executions of the same activity (with the same call-path) are aggregated in one node. The edge $(u, v) \subset E$ from node u to node v exists if any of the following conditions hold:

- Activities u and v are consecutive in the execution of the same process. In this case, the edge is denominated as *flow edge* and represents the control flow of execution local to a process that was actually executed.
- Activities u and v are matching communication activities where u belongs to one process and v belongs to another process or processes. In this case, the edge is denominated as a *message edge* and represents the inter-process communication. The node u belongs to the TAG of a sending process, while node v belongs to the TAG of a receiver process.
- Activities u and v represent the completion and initiation of a non-blocking communication activity. In this case, the edge is denominated as a *completion edge*, and represents the completion of a communication activity (it connects the completion node with its start node) and is local to a process.

Definition 12 (*Parallel task activity graph*) We define the parallel task activity graph (PTAG) of a parallel program execution to be a sum of the TAG sub-graphs for all the processes of that program.

Similar to the *parallel dynamic program graph (PDG)* approach presented by Choi [10] and later by Mirgorodskiy [30], our abstraction has the property to represent Lamport's happened-before relationship between nodes. In particular, an activity u happened before another activity v if and only if v is reachable from u in the TAG. That is, activity u happened before activity v if there exists a path in the TAG from the u node to the v node.

Adapting observations used in previous works, namely flowback analysis by Choi [10], backward slicing [48], and failure detection by Mirgorodskiy [30], this fact has important consequences. First, the conclusion is that, if activity u happened before an activity v, then u could causally affect v. If the v activity is identified as a performance bottleneck, then its root causes might be one or more activities that happened before v or (obviously) the activity itself. Second, if activity u is the root cause of some performance problem, it is likely to manifest its symptoms after activity u.

3 Online performance abstraction

In this section, we outline our techniques for the online abstraction of individual MPI processes, inter-process communication and whole-application [33]. We construct the PTAG in two steps. First, we collect local information at runtime on each MPI process and construct individual TAGs independently. This includes monitoring the local events of a process, as well as communication events. Second, to abstract the execution of the entire application, we collect snapshots of the TAG of each process and merge them together into a new parallel abstraction. An up-to-date TAG of process behavior is available continuously throughout the process execution as its construction is incremental and performed inside the process space. The PTAG is updated periodically as individual TAG snapshots are collected via the network and merged with programmed frequency.

3.1 Process abstraction

To construct and maintain the TAG of an individual process during its execution, we first analyze the program's executable and perform static code analysis to discover the program's structure. Next, we dynamically instrument the program by injecting tracing statements into selected functions and loops. During execution, these statements generate records of executed events that we use to incrementally build a TAG directly in the process memory space. In addition, we perform measurements and collect statistical execution profiles to reflect the program behavior. Finally, we enable external observers to access the TAG by taking its snapshots on-the-fly.

3.1.1 Offline program analysis

Before the application execution, we determine a set of target functions, a configurable set of functions of inter-

ests that represent program activities to be abstracted as graph nodes. In our work, we focus on abstracting interprocess communication activities, and, therefore, we define all MPI communication routines as target functions. However, a user may declare their own functions of interest, such as computation-specific functions or parallel I/O functions, to be reflected in the resulting abstraction.

We have developed a simple framework that allows the user to provide application-specific knowledge in the form of activity definition. The framework enables the user to flexibly declare all target functions that will be instrumented and abstracted in the TAG, including a set of certain parameters to be recorded (e.g., values of parameters passed to the target function). This definition statically identifies all possible graph nodes, but leaves the definition of edges to be discovered automatically at runtime.

Then we perform static code analysis. We parse the application executable via the Dyninst library [9,47] and extract the static call graph. A call graph refers to a directed graph that represents a calling relationship among subroutines in a program. Although not all possible relationships can be determined statically (for example, calls performed via function pointers are not available for static analysis), these graphs typically give a very good approximation of all possible execution paths, and have proven useful for constructing heuristics [25].

In our studies, we use the static call graph for two purposes. To begin with, we match the definition of activities with the application's executable. For each target function, we traverse the call graph and check if it is being invoked by the application code. If that is not the case, we may discard instrumenting that function. Usually MPI codes do not use all possible MPI primitives and, hence, this step enables us to lower the startup overhead.

Our observation is that, in many parallel scientific codes, the exhibited behavioral patterns are highly correlated to static program constructs, such as loops. This is especially true for stencil codes. A loop is a basic building block that reflects a repetitive sequence of actions. We explore this feature and abstract the execution cycles explicitly. Knowing the boundaries of loops enables us to pre-instrument them, track their execution and reflect them explicitly in the TAG at runtime. This approach differs from the classical pattern discovery based on learning repetitive sequences of actions [35]. In our study, we focus only on loops that contain invocations of communication activities (i.e., target functions). We avoid abstracting computational loops due to possibly high instrumentation overhead. We have developed a search algorithm that detects relevant loops by combining the static call graph traversal with control flow analysis. We assume that any loop in a program that leads (although through conditional control flow) to an invocation of any of the target functions is a relevant loop. We have found that this heuristic enables the detection of communication pattern boundaries at a low cost and with a precision that is reasonably close to the best possible answer.

3.1.2 Dynamic instrumentation of a program

To build a TAG of a process while it is running, we first must be able to monitor its execution. This requires the program to be instrumented in order to collect relevant data. The construction of the abstraction requires two types of data to be collected:

- Events—such as program start or termination, entry or exit of selected functions that represent observed activities, entry or exit of selected loops
- Performance metrics—such as counters (e.g., number of messages sent) or timers (e.g., time spent executing a particular function). Some of the metrics can be derived from events (e.g., event counter); others require the insertion of specific instrumentation (e.g., hardware counters).

To collect all the necessary data, we use Dyninst [9,47], a library that provides a platform-independent dynamic instrumentation capability. We use three types of instrumentation:

- Control event instrumentation—to monitor the occurrence of control events, such as process start and termination (both normal and abnormal), we instrument predetermined program locations at the main(), exit() and abort() functions.
- Function instrumentation—we instrument the entry and exit points of each target function. The entry instrumentation executes two actions (Fig. 6). First, it performs a low-overhead stack walk to capture the actual callpath and then determines its unique identifier. Second, it reads and stores the values necessary to calculate the requested performance metrics, for example, the current virtual CPU time that is used to measure the inclusive time spent executing the instrumented function.
- Loop instrumentation—we distinguish here the loop instrumentation, as we abstract a behavioral pattern from

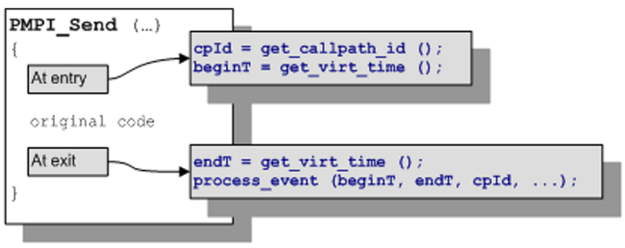

Fig. 6 Event generation instrumentation inserted into an example target function

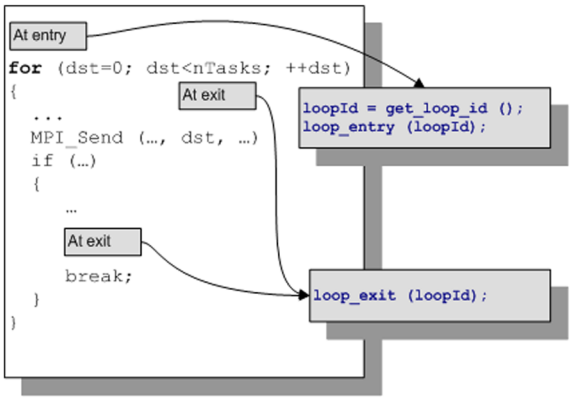

Fig. 7 Event generation instrumentation inserted into an example loop

each process and provide certain information to the causal execution paths (to discover causal execution paths that have the highest execution time in a loop). We insert instrumentation into the loop entry and exit points (Fig. 7). The instrumentation executes two actions. First, it determines the current loop's unique identifier. This identifier combines the call-path identifier of a current function (loop call-context) with a loop entry point address to reflect the loop's location inside the function. If the call-path has not yet been determined in the scope of an actual function that contains the loop, a complete call-path identification process is necessary. Otherwise, we reuse the call-path identifier that has already been determined in the current scope. This avoids an unnecessary overhead of calculating the identifier more than once. Second, the instrumentation updates the TAG, adding two nodes that reflect loop-entry and loop-exit. For each loop, we track the number of iterations and execution time. This instrumentation is inexpensive, as it only assures the existence of a corresponding loop marker node in the TAG.

To distinguish the location of events, we abstract each communication function considering a call-path taken to reach the function. This information is particularly useful as the function behavior in MPI programs often varies widely depending on the caller's chain on the stack. For that reason, an activity invoked from distinct call-paths (e.g., main/f1/foo/MPI_Isend and main/f2/foo/MPI_Isend) is represented by separate nodes in the graph.

We determine and uniquely identify actual program call-paths in the following way. First, adopting the iPath approach [5], we perform a cost-effective stack walk to determine the current call-path. Second, while walking the stack, we calculate the call-path hash signature. Third, using this signature, we perform an inexpensive lookup in the call-path hash table to determine the actual call-path unique identifier. If an actual call-path is not found, that is, it is executing for the first time, we assign it a new unique identifier and add a new entry to the hash table.

To capture the actual process call-path, we use the stack walking technique. Stack walking is the ability to read the actual process call stack. A call stack is composed of stack frames (sometimes called activation records). These are machine-dependent data structures containing subroutine state information (e.g., subroutine return address). Each stack frame corresponds to a call to a subroutine which has not yet terminated with a return. The stack frame at the top of the stack is for the currently executing routine.

In our approach, when walking the stack, we build a vector of call-site addresses that represents the actual call-path. We identify each call-path with a unique, numeric identifier that can be used to determine the actual location in the TAG. The unique identifier technique allows constant time look-up speed, since the identifier can be used as an index on an array of nodes. Given a vector of call-site addresses, we calculate the call-path hash signature. For that purpose, we use a combined hashing function:

$$CPSignature = \sum_{i=0}^{cpLen} addressHash(cpIP_i) \quad (1)$$

where

- *CPSignature* a calculated call-path signature
- *cpLen* call-path vector length
- $cpIP_i$ a callsite return address stored at index i
- *addressHash()* an XOR-based integer hashing function selected because of its speed and acceptable level of key conflicts.

Second, we use a hash table that associates call-path signatures with call-path entries. A call-path entry stores a vector of addresses and a sequential number used as a unique call-path identifier. The hash table serves as a call-path repository and supports an efficient, constant-time lookup and the efficient insertion of new entries. In addition, we apply the recently-used path heuristic to augment the efficiency of the look-up. Figure 8 illustrates the call-path identification technique.

Our approach to identifying the call-path in place has several advantages over the existing profiling tools. It can be injected or removed by means of dynamic instrumentation into selected program locations at run-time without requiring any other modifications to the program. This technique is able to act only in particular functions (e.g., selected MPI functions), and only these functions are instrumented. In effect, the overhead is incurred only when instrumentation is executed. In our technique, we insert the call-path identifica-

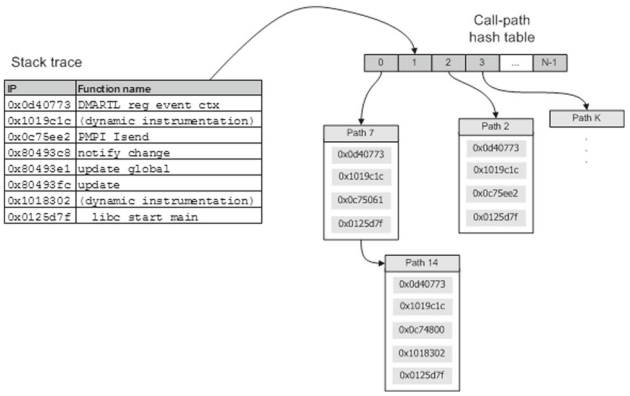

Fig. 8 Call-path identification

tion functionality mainly into MPI communication routines, which are much more expensive in terms of execution time and so the overhead is kept low with relation to the cost of the invoked routine (the absolute overhead is the same, no matter which routine is called). The results of this technique are available immediately at run-time, making it possible to perform a location-aware analysis.

3.1.3 Incremental abstraction

To enable efficient TAG construction for a single MPI process, we maintain a local, partial execution graph in the process's memory and update it incrementally by consuming the incoming flow of events. There are a number of alternatives for representing graphs in memory. For TAG purposes, we have chosen the adjacency list approach that represents each node as a data structure that contains a list of all adjacent nodes. It is the most appropriate, because the constructed graph is sparse. This observation results from the natural characteristics of the programs. The graph reflects the execution flow through high-level program constructs such as loops and communication operations, which are abstracted as nodes connected by edges. The degree of each node is usually one, as each node is simply followed by another sequential instruction (node) except branches that have a higher degree (usually two). Switch statements usually have higher degrees, but they are much less frequent.

We maintain a data structure that contains a collection of nodes and a collection of edges. For each node, besides storing its properties, we maintain a list of outgoing edges that reflect possible flows that actually got executed. Additionally, for each node and edge, we maintain a list of associated performance metrics. The construction procedure is the following. We process each event record that has been generated. Starting with an empty graph, we add a new node that represents a currently executed activity when it is executed the first time or we update the node if it already exists. The mapping between the activity and the graph nodes is direct, as we use a unique call-path identifier as an index into the array of nodes. To abstract the execution flow, we reflect the transition from a previous activity to a current activity by adding or updating an edge from the previous node to the actual node. The instrumentation also updates the execution profile of the affected node and edge by evaluating and aggregating desired performance metrics. If the node represents a communication activity, we create (or update if it exists) a message-edge and store the associated attributes.

Another requirement that we address with our technique is the ability to examine a partial TAG at runtime. This requirement leads to additional complications:

- To allow an external process (an observer) to access the TAG data, we allocate and store it in a shared memory segment. An external observer may then attach to this segment in a read-only mode and then directly read the memory to access the TAG data. The observer may read the TAG in place or periodically take its snapshots, i.e. sample the actual state of the abstraction and store its copy, without stopping the process. This is illustrated in Fig. 9.
- To minimize the impact of taking a snapshot we use a compact representation of the abstraction in memory avoiding fragmentation and complex data layouts. In particular, we use pointer-less data structures to enable zero-cost data relocation. That is, after copying the TAG to a different memory location, it is valid and may be used immediately. We avoid using pointers as shared memory segments are usually mapped at different addresses for each process, and each pointer would require an explicit translation (i.e. moving by an offset) thus increasing the sampling overhead.
- Finally, there are some synchronization issues to solve. As the TAG snapshot can be taken at any arbitrary instant, for example in the middle of an update, we must ensure its consistency. Since the instrumentation code inserted

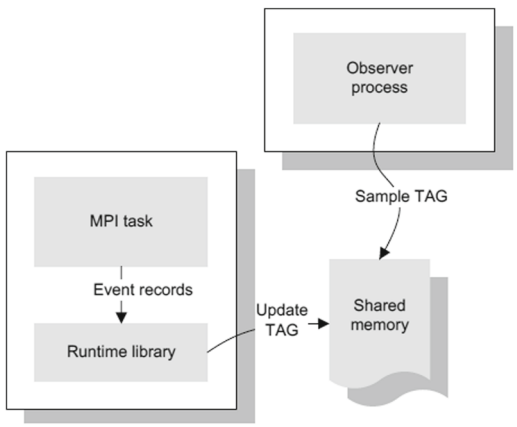

Fig. 9 Sampling TAG via shared memory

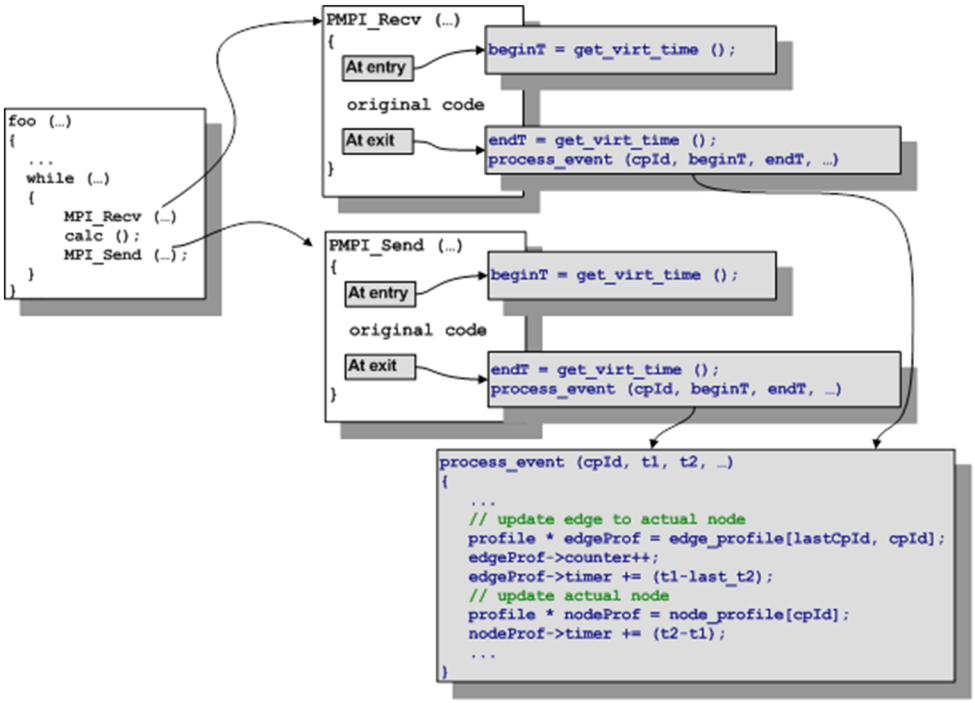

Fig. 10 Instrumentation that captures basic performance metrics for nodes and edges

into an application may write data just as it is being sampled (read) by an external observer, care must be taken to ensure that consistent values get sampled. However, the overhead must be kept low. To achieve both goals, we use optimistic sampling with lock-free synchronization. This solution is conceptually close to the software transactional memory (STM) approach [16]. STM is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. It works as an alternative to lock-based synchronization and is typically implemented in a lock-free way. A transaction, in this context, is a piece of code that executes a series of reads and writes to shared memory. STM is optimistic in the sense that a process completes modifications to shared memory without regard for what other processes might be doing. Instead of placing the responsibility on the writer to make sure it does not adversely affect others, it is placed on the reader, who, after completing an entire transaction, verifies that other threads have not concurrently made changes to memory that it accessed in the past. If a transaction cannot be committed due to conflicting changes, it is typically aborted and re-executed from the beginning until it succeeds.

The benefit of such an approach is increased concurrency: the observed process does not need to wait for access to the TAG, and an application process can safely and simultaneously modify disjointed parts of a data structure that would normally be protected under the same lock. Despite the overhead of retrying samples that fail, in most realistic programs conflicts arise rarely enough that there is an immense performance gain over lock-based protocols.

3.1.4 Execution profiles

We describe the behavior of program activities with execution profiles. We construct profiles by adding performance measurements to nodes and edges. As each activity (node or edge) might be executed multiple times, we aggregate the collected performance data into statistical metrics. Graph nodes are usually associated with particular functions (e.g., MPI_Bcast). Therefore, the execution profile of a node contains function-level performance metrics aggregated over the runtime of an application. This is different for graph edges that reflect an arbitrary control flow between two consecutive activities. An edge might represent a short sequence of basic blocks or even an invocation of a function call between two consecutive communication activities that hides a complex numerical algorithm. In this context, the execution profile of an edge has a varying level of granularity, depending on what actually got executed. We use two basic metrics for all the nodes and edges: a timer which measures the elapsed virtual time and a counter which counts the number of executions. We also calculate min, max, and stddev metrics to track variations. Additionally, we can add/remove arbitrary performance metrics to/from any activity.

Consider the example illustrated in Fig. 10. The body of the while loop inside function foo() is abstracted in the TAG as a sequence:

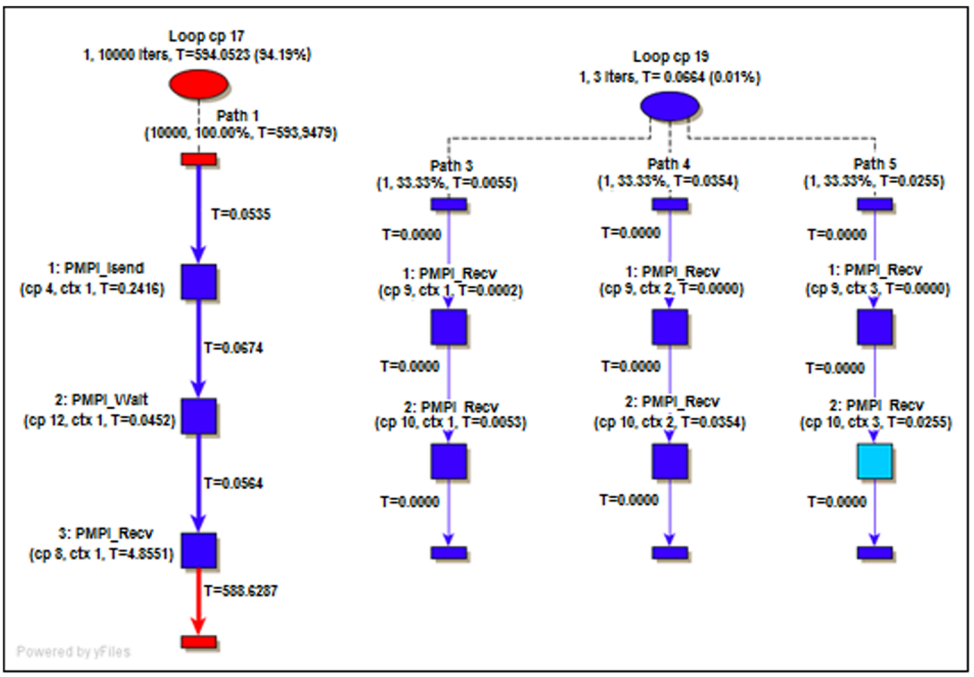

Fig. 11 Causal execution paths collected for example loops

MPI_Recv node → calc edge → MPI_Send node.

We insert instrumentation into the entry and exit of target functions PMPI_Recv and PMPI_Send. This instrumentation measures virtual time at the entry (beginT) and exit (endT) of each function and finally updates the TAG by invoking a process_event function. The latter updates the graph structure (omitted in this example), and then the performance metrics of the edge from a recent to an actually executed node, and, finally, the metrics of the node itself. We increment the number of times an edge has been executed and a timer that sums up the total time spent executing the edge. The edge execution time is calculated as the difference between the actual node's entry time and the recent node's exit time, and it does not require any additional measurements. The node profile is updated by adding the total time spent executing the node (t2–t1) to the timer. Finally, bookkeeping actions are performed, such as updating the recent node and its last exit time.

3.1.5 Causal execution path collection

To enable lower-level analysis, we introduce causal execution paths: temporary ordered sequences of activities [22]. Causal execution paths are instances of execution flows, and they are in fact recorded event traces that include contextual data (such as sender/receiver rank) and an execution profile. The major difference is that causal execution paths aggregate repeated instances of the same execution paths, while event traces store each repetition independently. Repeated executed paths are subject to an on-the-fly aggregation of performance metrics. Path profiles include counter and total time spent per path, giving insight for analysis. For example, we may request causal execution path tracking for a selected loop. Loop entry/exit events determine path boundaries, and each unique sequence of activities is identified as a separate path. These paths contain the comprehensive detail of performance data, giving a good base for the root-cause analysis.

Consider the example illustrated in Fig. 11. We have requested the collection of causal execution paths for loops 17 and 19, previously identified using the TAG of some application. We may observe that loop 17 has been executed 10,000 times, but is characterized only by a unique sequence of events, represented by path 1. Every iteration consists of asynchronously sending a message to process with rank 1 (ctx 1), then receiving a response message from the same process, and, finally, performing some local calculations. The execution profile shows that the process spends 94.2 % of its time on this loop, and most of this is taken up by the last computing edge (marked with a different color). A different example is loop 19, which has executed three iterations. However, each iteration took a different execution path (i.e., paths 3, 4 and 5). Although performed activities in the execution flow are the same (receiving two consecutive messages), the paths are differentiated by a sender rank.

Our approach considers an online, on demand collection of causal execution paths executed between selected activation and deactivation events. The causal execution path collection begins when the process executes a user-provided or tool-provided activation event. This event corresponds to some semantic activity, such as sending a particular message or is related to entering a specific code region, such as a loop. This mechanism has the ability to be deployed on demand

and to collect detailed runtime information while minimizing the execution overhead. Once activated, the mechanism records the sequence of executed activities that are identified by the TAG, for example, all communication activities, in a ring buffer. Together with each activity, we store contextual attributes, such as sender/receiver rank and timestamps. The recording finishes when the execution reaches a deactivation event, such as a loop exit. Once completed, we aggregate the path to the path repository using a hash table. The repository stores all distinct causal execution paths that have been executed together with their accumulated profiles. When the user or analyzer tool decides to stop the collection procedure, the mechanism can be removed from the running process.

The collection mechanism is based on dynamic instrumentation. We insert instrumentation into the running process at points that correspond to activation and deactivation events, and we also augment the TAG update function that is already inserted for purposes of TAG construction. The activation instrumentation evaluates the necessary conditions, and, if they hold, it marks the path recording flag and returns control to the application. The update instrumentation records each executed activity in the path buffer when the recording flag is active. Finally, the deactivation instrumentation adds the path to the repository and returns control to the application.

3.2 Communication abstraction

Our approach uses local tracing to capture causally ordered events and abstract local execution flow in each application process. However, when the inter-process communication is involved, this approach must be extended. To connect TAGs of individual processes into one PTAG, it is necessary to track the execution flow from the sender to receiver(s) processes by following individual messages that cross process boundaries.

We have found that by determining and intercepting all inter-process communications, we can obtain the data necessary to dynamically construct a PTAG that reflects executed communications. The key idea is to match a sender call context, represented by a node in the sender process TAG, with a receiver call context, represented by a corresponding receive node in the receiver process TAG. To achieve that, we attach a small amount of additional information to every message that is transmitted. Additionally, as we abstract the application online, the matching must be performed dynamically during application runtime. To accomplish this goal, we piggyback the additional data from sender to receiver(s) in every MPI message. We transmit the current send node identifier, and we store it in the matching receive node as the incoming message edge. This feature enables us to logically connect TAGs while keeping them distributed. Finally, in order to capture communication profiles, we track the count and time histograms for each message edge individually.

The big concern of the MPI piggyback mechanism is the overhead. To choose the most suitable technique with minimal overhead, we carefully examined different canonical solutions and validated their characteristics [42]. The obtained results, finally and similarly to those presented by Schulz in [41], led us to the conclusion that the selection of the best technique depends on the message size and the type of communication operations. We then decided to develop a hybrid MPI piggyback technique that combines different mechanisms depending on the communication type (point-to-point or broadcast) and message size (large or small). For small messages, we take advantage of the data-type wrapping. The major steps of our approach are:

- Identify communication routines that are invoked by the application process
- Intercept these communication routines with send and receive wrappers that detect the initiation of communication
- In the send wrapper, attach an executed send node call-path identifier (cpId) to every message sent by a process
- In the receive wrapper, detect and receive the identifier at the receiver's process
- Store the identifier in the receiver process TAG as an attribute of the receive node that has actually been executed

For large messages, we send an additional message, which we found to be much cheaper than wrapping. Moreover, we interleave the original operation with an asynchronous transmission of piggyback data. This optimization partially hides the latency of the additional send and lowers the overall intrusion.

3.2.1 Abstracting communication activities in a TAG

To reflect actual communications, we introduce message edges. A message edge describes an act of communication and identifies all of its participants. We also define an execution profile for each message edge. Consider passing a message from a sender to a receiver, as illustrated in Fig. 12. A sender node, 17, at the process called $Task_0$ sends a message that is received by node 21 at process $Task_1$. The piggyback data attached to the message identifies a sender as node 17 at $Task_0$. From the perspective of $Task_0$, we abstract this communication by creating a new message edge attached to node 17. We classify the message edge as outgoing to express the direction of message flow. We store the destination rank ($Task_0$) in the message edge and indicate that the remote receive node is unknown. On the receiver side, we create a corresponding message edge to reflect the reception of the message and attach it to node 21. This edge is incoming and contains the source rank ($Task_0$) and the sender

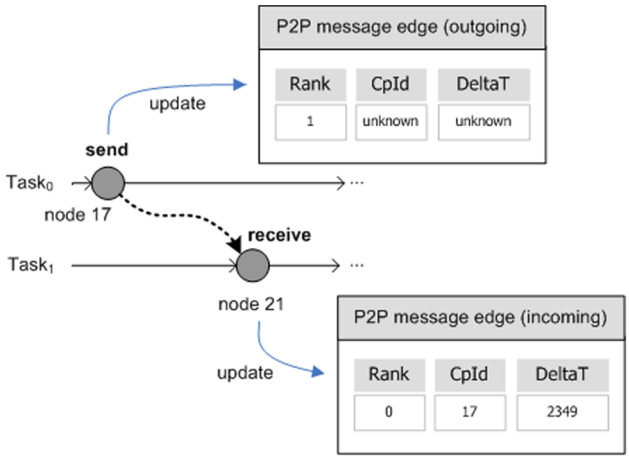

Fig. 12 Message edges for point-to-point operation

node identifier (17) delivered as piggyback data. In addition, we introduce a time difference metric (DeltaT). The DeltaT metric is calculated as the difference between a timestamp taken at the entry of the operation at a given process and the entry of the corresponding operation at a reference process (i.e., the receiver process) and adjusted by an estimated clock difference between the nodes. This metric might be used to estimate the wait time caused by the temporal asynchrony of communicating processes.

Considering types of MPI communication primitives, we distinguish three different types of communications:

- Blocking point-to-point operations
  - *Incoming* message receives
  - *Outgoing* message sends
  - *Exchange* message exchanges (send/receive)
- Non-blocking point-to-point operations
  - *Incoming* message receives
  - *Outgoing* message sends
- Collective operations

It is straightforward to abstract *point-to-point blocking communication* calls, because all the information that is required for abstracting is available at the exit of the call and there are no dependencies on other operations.

Abstracting *non-blocking operations* adds more complications. The instrumentation of non-blocking calls is illustrated in Fig. 13. To abstract these operations, we correlate the start of each non-blocking operation (e.g., MPI_Isend) with its completion operation (e.g., MPI_Wait). To do that, we need to keep track of opaque MPI request handles and update the TAG in a slightly different way. First, when the operation starts, we update the start node that indicates the initiation of the non-blocking operation (e.g., MPI_Isend) and store the associated request handle. We maintain an associative array of pairs (MPI_Request; call-path ID) with constant insertion and lookup times. In addition, we apply recently used optimization to minimize the lookup cost even further. Second,

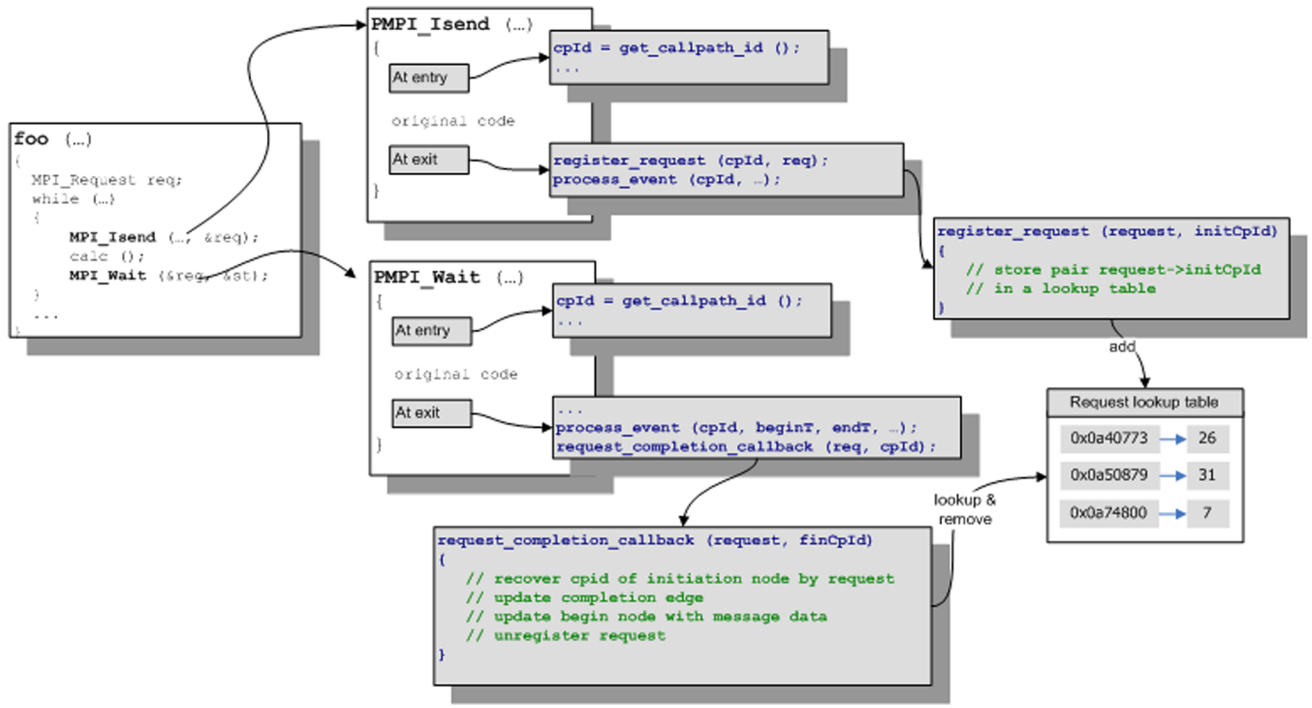

Fig. 13 Abstracting non-blocking communication calls

when the operation is completed, we update the completion node (e.g., MPI_Wait). Third, we recover the corresponding start node by its request handle, and add or update the completion edge between the completion node and the start node. The purpose of completion edges is to reflect the scope of the non-blocking operations. Finally, we update the message edge. When sending a message, all the data necessary to update the message edge is available at the entry of the send call. However, for non-blocking receive, we must defer the update until the operation is completed and we can access the valid MPI_Status structure.

Handling *collective communication* primitives is similar to blocking point-to-point primitives, with two exceptions. First, depending on the type of call, there are different scenarios of distribution of piggyback information. In particular:

- *Broadcast/scatter* each process stores its local piggyback data and the data received from the root process as a link structure. There is no central location where the piggyback data from all the processes is available at once. Instead, each participating node maintains its local data and a reference to the root process (Rank), the node that initiated the call (CpId) and the relative time difference to the root process (DeltaT).
- *Gather/reduce* the situation is inverted for these operations. A root node receives the piggyback from all the processes and maintains a list of tuples with participating processes (Rank), their corresponding nodes (CpId) and time differences (DeltaT) relative to the root process.
- *Rooted collective communication with large messages* for all types of operations we send an additional small message with piggyback data using non-blocking primitives.
- *All-to-all collective communication with small messages* for all-to-all message transmissions, we apply a mixed approach. We designate a single process as a root of the operation (linearly) and make it send a piggyback data to other processes. The rest of processes do not attach any data, but store its local piggyback data and received data from the root process.
- *All-to-all collective communication with large messages* similarly to rooted operations, we use a strategy based on sending an additional message. We designate a single process as a root (linearly) and invoke gather operation to collect all piggyback data in one place. We prefer the gather operation over non-blocking set of point-to-point calls because of scalability issues on massively parallel computers. Typically, the collective operations are optimized for good scalability, while point-to-point calls may suffer from linear scalability limitation.
- *Barrier* this synchronization call is special in that it does not transmit any data. Therefore, the only solution is to send an additional message. For this purpose, we use gather operation from all processes to a designated root (linearly).

The second issue that must be addressed results from non-deterministic communication calls. In MPI programs, non-determinism is introduced by so-called wild-card receives, i.e., MPI_Recv or MPI_Irecv calls combined with the MPI_ANY_SOURCE option, and wait calls, such as MPI_Waitsome. For example, a program becomes non-deterministic as soon as there is at least one wild-card receive in it, where at least two messages can be accepted. In such a case, the order in which the messages will be received is not predictable. This problem is known as message racing [21]. The non-deterministic repetitions of MPI calls, such as those triggered by, e.g., MPI_Recv (MPI_ANY_SOURCE), might present a challenge. In our approach, the TAG reflects the execution of, e.g., MPI_Recv activity and message links that indicate all possible senders. In this case, the non-determinism of the order in which the messages are received is not addressed because we do not preserve the message ordering at that level. In fact, the receive node maintains all possible message edges, but ignores the order in which they are added. We build causal execution paths to capture a detailed flow of events that preserves both program structure and message ordering. However, we do not capture all possible orderings, but focus only on those that are the most frequent or take up significant execution time.

For example, consider a loop that terminates upon receiving messages from all the processes in a group. We take into account the following scenarios:

- *Deterministic sequence of senders* the receive operation is used to collect messages from a fixed number of senders in a fixed order. For example, a typical construction is a loop that terminates upon receiving messages from all of the tasks in a group. In this case, the receive operation is deterministic, as the code forces the order in which the messages are to be received. Each sequence can be abstracted as a distinct causal path.
- *Non-deterministic set of senders* the messages are received from a determined number of senders, but in an arbitrary order. One example is a loop with an MPI_Recv (MPI_ANY_SOURCE) operation or a pair of MPI_Irecv() and MPI_Waitall() calls. This results in an explosion of possible combinations of sequences of senders. Although non-deterministic, it is common for the actual number of orderings in an execution to be limited, especially for short sequences. Therefore, we intend to capture a limited number of distinct sequences and their statistics (i.e. number of occurrences and total time). This enables us to detect dominant sequences in some

cases. However, addressing all possible cases remains as future work.

To characterize the communication behavior, we gather aggregate statistical metrics for each individual message edge. This provides data for statistical communication analysis. We define basic performance metrics for each message edge as follows:

- *Timer* time accumulated sending/receiving over this edge
- *Message counter* number of messages sent/received
- *Byte counter* accumulated number of transmitted bytes
- *DeltaT* accumulated time difference (deviations from pair wise temporal synchrony of communicating processes). This metric is used for latency analysis.
- *Message size histogram (optional)* counts the number of messages of a given size class. We use a simple logarithm-based histogram *log2(size)* that divides the message size space into a fixed-number of buckets.
- *Message time histogram (optional)* breaks down timer metric into a fixed-number of buckets with accumulated time spent on each message size class.

In addition, dynamic instrumentation enables on-the-fly activation/deactivation of histograms, and, more generally, the insertion and removal of arbitrary metrics for each individual message edge.

Our approach is similar to that of mpiP [17], which collects metrics separately for each communication activity identified by a unique call path. However, we collect more fine-grained statistics as we distinguish between distinct communication pairs (sender-receiver) or groups (for collective operations). In general, the advantage of our approach over other statistical ones is that it preserves the program structure and it may be extended to preserve a temporal ordering of events (with causal paths), which improves its ability to identify the causes of problems more precisely. On the other hand, the statistical approach (even augmented with causal paths) is lossy and its precision for analysis might be lower when compared with full communication traces. However, full tracing comes at the cost of high data volume and scalability problems. Therefore, the selection of the most appropriate technique depends on the trade-off between overhead and preserved level-of details.

3.3 Parallel application abstraction

To abstract the execution of an entire MPI application, we collect TAG snapshots from all the processes and merge them into a new global graph that we call the Parallel Task Activity Graph (PTAG). This process can be performed periodically, on demand, or at the end of the execution. The merge process is straightforward, as we take advantage of the information stored in the message edges. We process all message edges applying the following procedure:

- Each incoming point-to-point message edge contains data that uniquely identifies the sender process and send node. We add an edge that connects the send node to the matching receive node in the PTAG to reflect the communication.
- We ignore outgoing, point-to-point message edges, as they are redundant.
- Each collective message edge contains a vector of pairs that identifies all send processes and their corresponding node identifiers or a local node and a root node data. In both cases, we add a corresponding group of point-to-point edges to the PTAG to reflect the collective communication and its participants.

The limitation of the basic approach to PTAG construction is its inherent lack of scalability. To address this issue, we have developed a scalable approach for PTAG construction. To retain the benefits of TAGs while reducing the volume of collected data, we classify processes into behavioral equivalence classes. For that purpose, we propose an algorithm to merge two TAGs (clustering).

First, we decompose each graph into a set of hierarchically organized sub-graphs called sections. We define the following set of basic rules to identify a section:

- A sequence of events that does not contain a loop (e.g., from program start to the entry of the first loop).
- Each loop is a section.
- Each nested loop becomes a sub-section.

This approach is equivalent to converting the graph into a tree by eliminating all loop back edges. The use of trees instead of directed graphs leads to the generation of a hierarchical set of sections.

Next, we determine the behavioral equivalence of two corresponding sections from each TAG. We do not assume SPMD applications, but we compare two sections, searching for their similarity. In this sense, two sections are equivalent if they have a similar behavior and SPMD-like structures. We use the following rules:

1. The compared sections must be isomorphic. That is, we require both sub-graphs that represent the compared sections to be structurally identical. This means each sub-graph must be composed of the same set of nodes and edges. We consider two nodes to be structurally equal if they represent the same activity that has been executed in the same call-path. If a node represents a communication activity, we also compare message edges. We compare

two message edges, taking into account type (p2p or collective, incoming or outgoing) and source and destination ranks. We use relative ranks to compare source and destination processes. For example, if two processes, 2 and 6, exchange messages with their neighbors to the left, 1 and 5, the relative rank is −1 in both cases, which we interpret as equivalent communication behavior. Finally, two edges are structurally equal if they connect equivalent nodes. The structural equivalence enables us to find groups of sections and whole processes that execute the same activities. For example, the master process of a master-worker MPI application behaves substantially differently from the worker nodes. Workers perform units of work, while the master distributes the units among the workers and gathers the results.

2. The compared sections must have equivalent runtime behavior. We apply two rules to decide on behavioral equivalence of executed activities. First, we impose a strict rule that requires the counter metric, e.g., number of executions of each node and edge to be equal. This prevents treating loops with similar execution times but different numbers of iterations as equivalent. Second, we use a distance metric to compare the execution profiles of the compared sections. This rule enables us to distinguish between sections that execute the same activities but with different performances. The distance metric estimates the similarity of two execution flows: a long distance implies that flows have different behaviors; a short distance suggests that the flows are similar. We use distance metric instead of strict equality to tolerate insignificant variations between execution flows.

We define a distance metric between section p and section q as follows:

$$Distance(p,q) = \sum_{i=1}^{N} (T(p,n_i) - T(q,n_i))^2 + \sum_{j=1}^{E} (T(p,e_j) - T(q,e_j))^2 \quad (2)$$

where N is the number of nodes in the sub-graph, E is the number of edges in the sub-graph, n_i is the i-th node (from 1 to N), e_j is the j-th edge (from 1 to E), $T(section, n_i)$ is the time profile function of i-th node in a given section, $T(section, e_j)$ is the time profile function of j-th edge in a given section

The distance metric between sections is defined as the sum of Euclidian distances between the time profiles of each node and each edge. We use an accumulated timer metric (e.g., total time spent executing node/edge in a given time window) as a time profile function, T. If sections p and q behave similarly, each node and edge will consume similar amounts of time in both processes and their distances will be low, as will the sum of all the distances.

Finally, we merge equivalent sections into a single section and add it to the output graph. We label each section with identifiers of the processes that execute that section.

3.4 Abstraction-based analysis techniques

Our approach enables the assortment of online performance analysis techniques. The TAG allows for distributed reasoning about the behavior of individual processes online. It provides a high-level view of execution and enables the easy detection of performance bottlenecks and their locations in each process. As this information is available at runtime, the monitoring could be refined in order to provide more in-depth views of each problem. By merging individual TAGs, we gain a global application view, which provides the opportunity to analyze the whole application while it runs. This allows for the detection of program phases, the clustering of processes by their behavior and the detection of a load imbalance by matching loops between processes and comparing their profiles, as well as other observations. Some of our abstraction properties, such as the causal relationships between activities and the on-demand detection of causal execution paths, can be used to develop tools for root-cause problem diagnosis. Finally, the resulting abstraction can be displayed using visualization tools.

Actually, we have used the presented technique to build a set of techniques for online performance analysis that can be deployed in arbitrary MPI applications running in large-scale parallel systems. We have developed a three-step iterative approach called root-cause performance analysis (RCA) [43]. First, we identify the most severe performance bottlenecks and their locations in the application. Second, we perform an in-depth analysis of each individual problem. For that purpose, we apply a knowledge base to drive the detailed analysis of well-known parallel inefficiencies, such as those resulting from inter-process communication and synchronization. Finally, for each problem that has non-local causes, we perform a root-cause search by comparing concurrent execution paths followed by communicating processes and inferring the differences. This enables us to correlate different performance problems in causal relationships and distinguish the generation of inefficiencies (root causes) from their causal propagation (symptoms).

To identify these problems, we periodically collect snapshots of the PTAG that summarize the execution in a specified time window. Each snapshot is then used to locate communication or computational activities (nodes or edges) that contribute significantly to a total execution time. For each severe problem, we perform an in-depth analysis by investigating its possible causes by exploring a knowledge-based cause space. We focus on determining the causes that con-

tribute most to the total problem time. Our knowledge base provides a catalog of well-known performance inefficiencies and methods with which to interpret them. We focus on typical problems for the message-passing paradigm resulting from communication (e.g., limited bandwidth, a large number of small messages) or synchronization (e.g., late sender, late receiver, wait at collective operation). Moreover, for computational bottlenecks, we diagnose where the edge-constrained code spends time by applying code-range profiling. This technique aids the developers in discovering and locating performance inefficiencies.

Detection and alleviation of individual performance inefficiencies is often misleading. An overhead originating at a certain point in the application's process can causally propagate through the process control flow and then through the message flow to another process and cause other inefficiencies at other points. To address this issue, we propose a cause-effect analysis technique that intends to differentiate root causes from one or more of their symptoms and produce an explanation that characterizes each problem found in a parallel program. With our automated, abstraction-based performance analysis approach, we are able to easily identify the most severe performance problems during application execution and locate their root causes without previous knowledge of application internals.

The key differences of our technique from existing tools, such as EXPERT [50], Scalasca [26] and KappaPI 2 [18] are that it can be performed online, does not require full traces, scales to thousands of nodes and provides similar results. Programmers often have no idea where to begin searching for possible bottlenecks. Our tool allows the programmer to get a quick overview of the program's execution and provides guidance to problems, their causes and corresponding locations in the source code. It would also be very profitable to go one step further and provide the programmer with the actionable recommendations that could help solve the encountered problems. We believe such recommendations could be valuable for both non-experienced and expert users and could shorten the performance tuning process.

4 Prototype for dynamic abstraction of MPI applications

We have developed a prototype tool that is able to build a performance abstraction of an MPI application online, in a distributed way, and without access to the application source code [32]. The tool collects and processes the local event traces at runtime in each process, building individual TAGs. Then, it periodically collects the TAG snapshots and, using a hierarchy of intermediate processes, merges them into a global PTAG for online analysis and visualization.

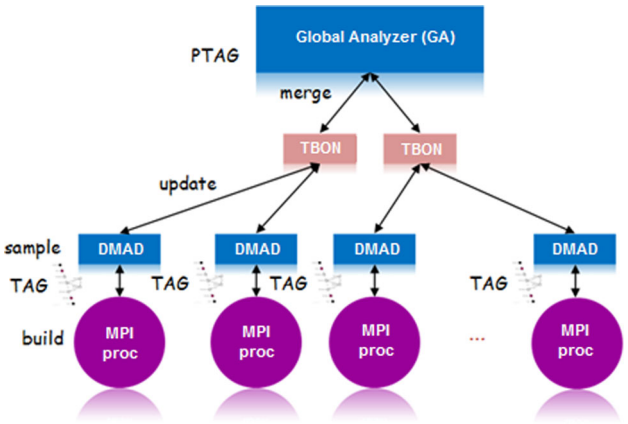

Fig. 14 Prototype tool architecture

The architecture of the prototype tool is illustrated in Fig. 14. It is based on the MRNet infrastructure [39], a software overlay network that provides efficient multicast and reduction communications for parallel and distributed tools. MRNet uses a tree of processes between the tool's front and back-ends to improve group communication performance. Our prototype tool is composed of four main components: the front-end (global analyzer—GA), a hierarchy of intermediate processes (TBON), the tool daemons (dynamic monitoring and analyzer Daemon—DMAD), and the runtime performance abstracting library (RTLib). The front-end coordinates the tool daemons and collects the TAG snapshots of the individual processes by merging them into a PTAG with the help of MRNet. It is also responsible for processing the PTAG, i.e., behavioral clustering. Finally, the front-end exports the PTAG into an open graph format GraphML [6]. The detailed flow of the PTAG creation using MRNet is presented in Fig. 15.

Each DMAD is a light-weight daemon based on the Dyninst library [9] that implements the following functionalities: static code analysis, loading of the RTLib library into the application process, interception of the MPI routines, instrumentation insertion to trace events and to collect performance metrics, process start, periodical capture of TAG snapshots, while the program is running, and TAG propagation to the front-end. The RTLib library is responsible for the incremental construction of a local TAG. It provides the implementation of graph manipulation routines optimized for fast insertion and constant-time look-ups. The graph and the associated performance metrics are stored in a shared memory segment as a compact, pointer-free data structure to allow the daemon to take its snapshots periodically without stopping the process. The frequency of taking TAG snapshots can be configured by a user.

Before the application starts, the tool determines the target functions, a configurable set of functions which identifies program activities. By default, we configure all MPI com-

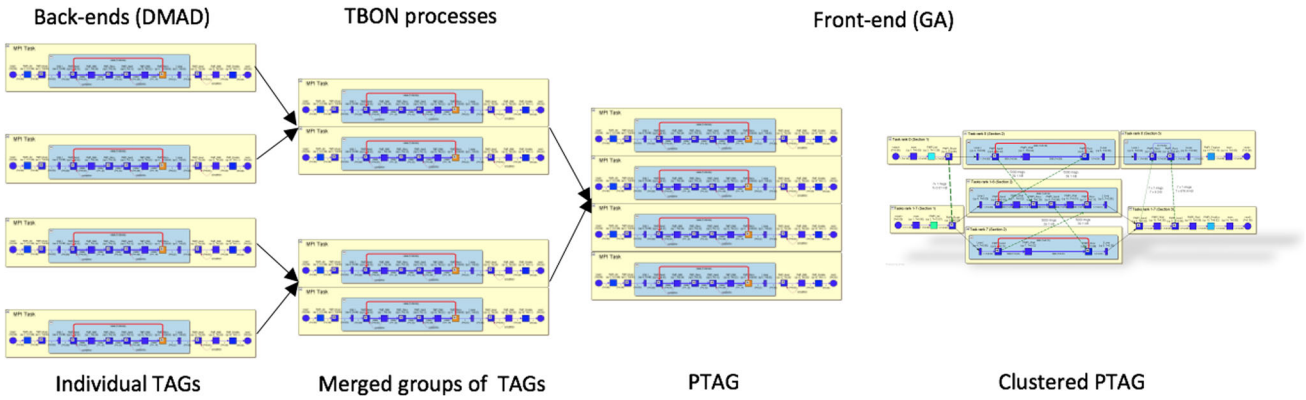

Fig. 15 Flow of the PTAG creation using MRNet

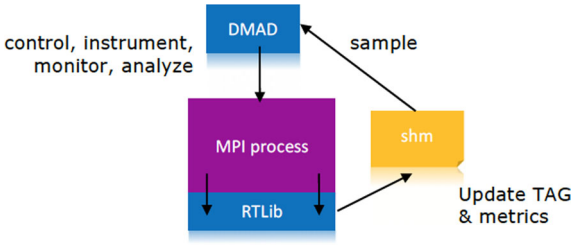

Fig. 16 Building TAGs by DMAD using shared memory

munication functions as communication activities. However, users may configure their own functions of interest to be reflected in the abstraction. Next, the daemon traverses the program static call graph and control flow graphs for selected functions to select loops which lead to the invocation of target functions.

To build a TAG, the DMAD instruments the entry and exit points of each target function. This instrumentation captures the record of the executed event. Then, it performs a low-overhead stack walk using the unwind library [1] to determine the actual call-path, calculates its signature and looks up a hash table to recover the node identifier. Next, the instrumentation invokes the RTLib library routine to process the event record. A local, partial execution graph (TAG) is maintained in each process. The library updates the graph structure by adding, if necessary, the node which represents a currently executed activity and the edge from the previous activity. Finally, it updates the execution profile of an affected node/edge by aggregating the desired performance metrics. The flow followed by the DMAD to build TAGs in shared memory is presented in Fig. 16.

The scalability of the cluster approach depends on the MPI application type: the more different types of processes there are in the application, the less compact the PTAG is. To avoid a huge overhead, the prototype tool supports the PTAG creation periodically or at the end of the execution (a user can configure when a PTAG is created and the period frequency). Moreover, as the tool is based on MRNet and uses a hierarchy of TBON processes, it provides efficient multicast and reduction communications. In the future, the clustering may be performed on different levels of the hierarchy, not just in the front-end (Global Analyzer).

The tool is implemented in C++ and targets MPI applications. We built our prototype tool using OpenMPI [15], Dyninst API [47] and MRNet [46]. It has been tested on x86/Linux, Intel IA64/Linux, and partially on PowerPC-64/Linux.

5 Experiments

To evaluate our approach, we have conducted a number of experiments with real-world MPI parallel applications. Our experiments were conducted in two different environments:

- UAB cluster (x86/Linux platform). This is a 128-CPU (32-node) cluster located at the Universitat Autònoma de Barcelona (UAB).
- MareNostrum (PowerPC-64/Linux platform). This supercomputer is located at the Barcelona Supercomputing Center (BSC) [3].

It must be pointed out that each MPI process is located on a separate CPU and, moreover, on different physical nodes to assure the same computation and communication conditions.

We experimented with applications that employ the most common styles of message-passing parallel programming, in particular:

- *SPMD* This technique is particularly appropriate for problems with regular, predictable communication patterns. We have experimented with four SPMD programs that apply different data decompositions:

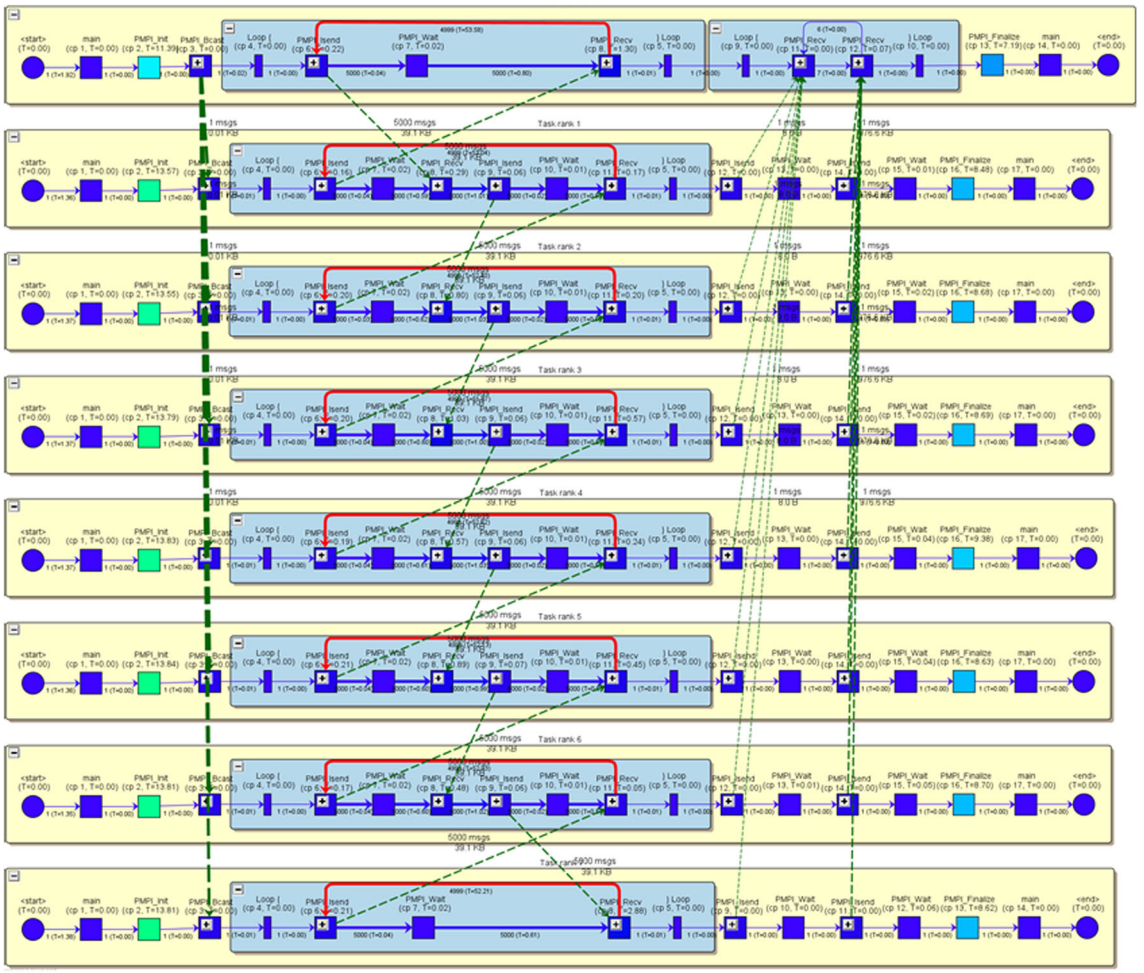

Fig. 17 PTAG for WaveSend program executed on 8 CPUs (UAB cluster)

- *WaveSend* 1D data decomposition, concurrent wave equation (C code)
- *NAS LU Benchmark* 2D data decomposition, CFD application (Fortran)
- *NAS IS Benchmark* 2D data decomposition, parallel sort over small integers (C code)
- *SMG2000* 3D data decomposition, parallel semi-coarsening multigrid solver (C code)

- *Master/worker* We have experimented with the XFire application, a large-scale forest fire propagation simulator code.

5.1 WaveSend (SPMD)

This program implements the concurrent wave equation as described in [11]. A vibrating string is decomposed into a vector of points. Since the amplitude of each point depends on its neighbors, a contiguous block of points is assigned to each MPI process ("block" decomposition). One of the processes (so-called master) reads the input vector and distributes the data to all of the processes. Then, all of the processes contribute to the calculation (including the master). Finally, the master process collects the updated points from all of the processes and saves the result, that is, the final vector of amplitudes from the last iteration. Each process is responsible for updating the amplitude of a number of points over time. At each iteration, each process exchanges boundary points with its nearest neighbors (left or right). WaveSend applies 1-D data decomposition and represents a typical SPMD. The program uses MPI point-to-point, non-blocking send and blocking receive primitives to exchange boundary points.

Figure 17 presents the visualization of the PTAG of the WaveSend program executed on 8 CPUs (million point wave and 5000 iterations). The space limitation prevents us from showing executions on a larger number of CPUs. The snapshot was collected after the program termination and shows the complete execution history of the program.

As we can observe, the abstraction visualization gives a quick and easy-to-understand overview of the program's behavior. We start by sketching a few intuitive interpretations

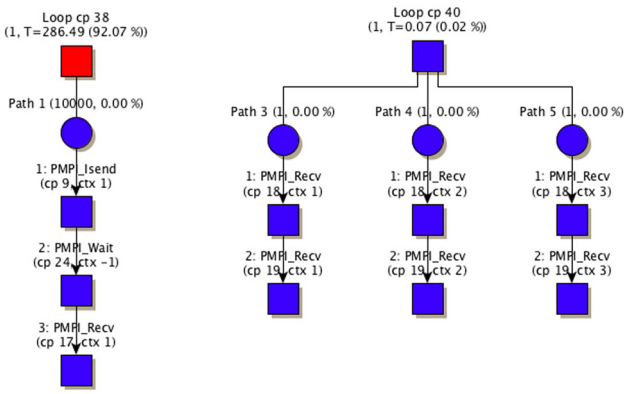

Fig. 18 An example causal execution path for the WaveSend application

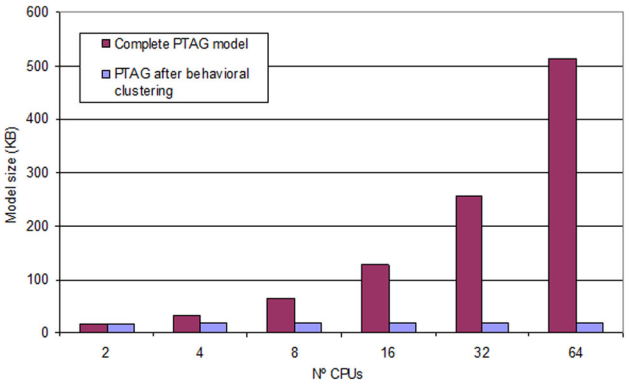

Fig. 19 PTAG snapshot size, varying numbers of CPUs

without actually requiring any knowledge of the program's internals. The first observation is that the program exhibits three main phases of execution. The first phase we may interpret as initialization: the master process broadcasts initial data to other processes. In the second phase, each process enters its main loop and executes 5000 iterations, exchanging data with its neighbors. The communication pattern clearly reveals 1-D decomposition: we can identify border processes with a single neighbor (left or right) and processes with two neighbors (left and right). Finally, in the last phase, all processes but the master process send results using two point-to-point messages to the master process that collect the results.

In the presented example, the performance metrics indicate that the program is CPU-bound and is well-balanced. More than 70 % of the time is spent on calculations in the main loop, and there are no observable significant communication inefficiencies except for a relatively high cost of MPI_Init and MPI_Finalize calls (28 %). We can also observe that the interleaving non-blocking sends with calculations (see MPI_Isend and MPI_Wait calls in the main loop) do not offer any significant benefits. If configured in the DMAD, we can obtain a list of causal execution paths that detail the application behavior. An example causal execution path is presented in Fig. 18 and we can use it for analysis purposes, as they contain contextual data (such as sender/receiver rank) and execution profile.

The size of a complete PTAG snapshot depends linearly on the number of MPI processes. And the size of each individual snapshot of the TAG depends mainly on the program's code size. Considering a tiny program like WaveSend, the size of each TAG expressed in XML is about 8 KB. Accordingly, for 1024 MPI processes, the complete PTAG reaches approximately 8 MB. This relationship between the size and the number of CPUs has been illustrated in Fig. 19.

It is worth noting, however, that stencil codes are usually regular and the level of repetitiveness between processes is usually high. To simplify the analysis and visualization of the PTAG, we took advantage of this property and applied our behavioral clustering technique. We have performed clustering in two stages. First, we exploited cross-node repetitiveness. Except for border processes, all other processes are characterized with very similar behavior. To detect these similarities, we have applied a graph transformation algorithm that merges equivalent TAGs, where equivalence is based on graph isomorphism. The result of this transformation is shown in Fig. 20. There are three classes of TAGs, namely the border processes (A and B) and the internal process (C). The abstraction of processes in each class are isomorphic and their execution profiles are similar (assuming a 2 % tolerance threshold). The property of our clustered abstraction is that its size depends on the number of distinct behavioral classes rather than on the number of processes.

Second, we have refined our clustering and exploited code-section level repetitiveness across the nodes. Instead of clustering the whole PTAG, we divide each TAG into code sections and merge equivalent sections between the processes. The result of this transformation is shown in Fig. 21. In comparison to TAG-level clustering, we were able to discover two subclasses of behavior during initialization, namely a master process that distributes the data and the other processes that receive the data. The second section, namely the main loop, has three subclasses: one for each border process and the processes that communicate with the two neighbors. Finally, the third section also has two subclasses: the master that collects the results and the other processes that submit them to the master.

Furthermore, we demonstrate the ability for other PTAG transformations that are useful for better program understanding. Consider the visualization with communication phases as presented in Fig. 22. We have applied a transformation that preserves message edges between intercommunicating processes, but reduces each process TAG to a single node. By visualizing snapshots of the PTAG taken after each code-section, we can discover three main communication phases:

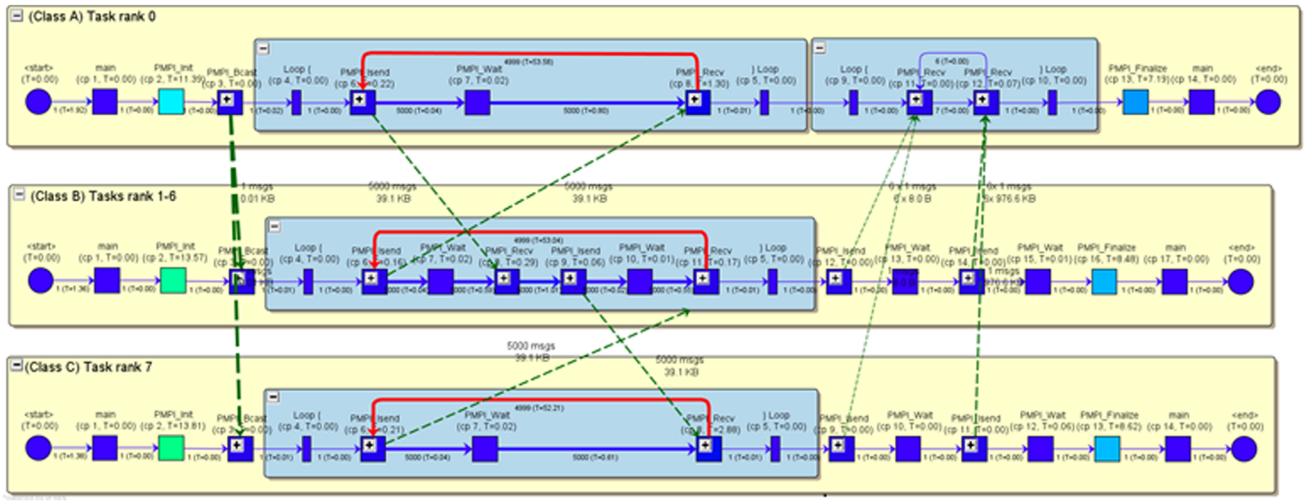

Fig. 20 Improved visualization of WaveSend PTAG clustered by similar processes. We merge all TAG sub-graphs that are isomorphic and have equivalent execution profiles

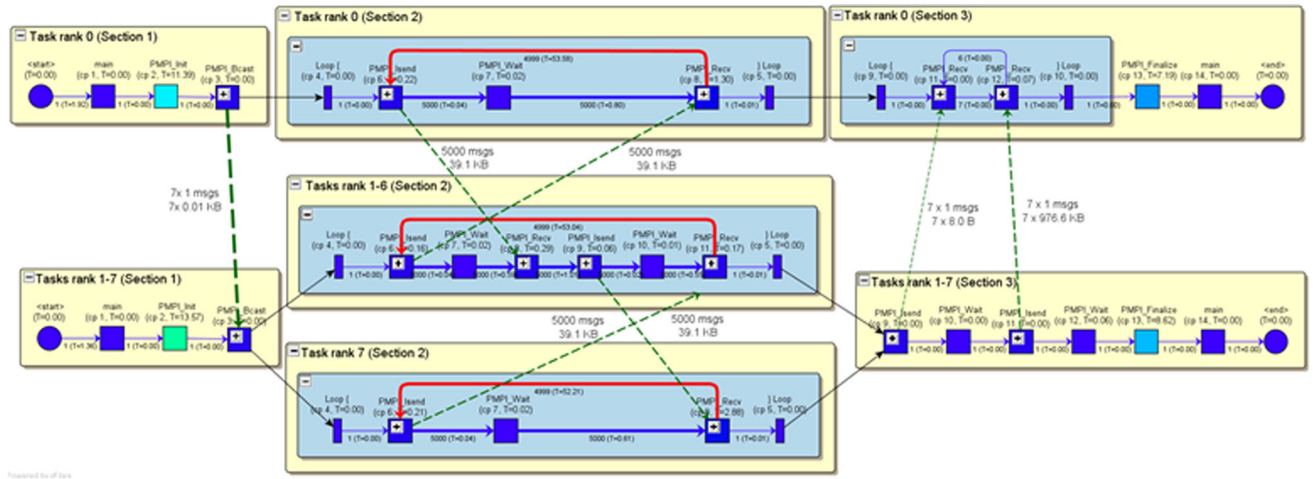

Fig. 21 Scalable visualization of WaveSend PTAG after behavioral clustering of code sections

initial data distribution, 1D nearest neighbor exchange pattern and final reduction. Each phase is characterized with a statistical communication profile that includes the number and size of the transmitted messages.

To evaluate the overhead introduced by our prototype tool, we have conducted a number of experiments. For this application, the tool instrumented 10 functions (e.g., entry and exit points of main, MPI_Isend, MPI_Recv and other used MPI calls) and 2 loops (i.e., entry and exit points of the main loop and result collection loop). This startup overhead was constant and independent from the number of processes, as all the processes start in parallel on different CPUs. We do not include this startup overhead when presenting program execution times.

We compared execution times of the WaveSend code run with and without control of our prototype tool, varying the number of CPUs. Figure 23 presents a comparison of large problem size and small-scale executions, with CPUs varying from 2 to 32 on the UAB cluster. Detailed data are presented in Table 1. We measure the total overhead the presented technique imposes on the executed application as the difference between the wall clock completion time of an application executed under the DMAD tool and the completion time of the same application without the tool. We can observe that the introduced overhead is kept low (0.21–2.55 %) when varying the number of CPUs from 2 to 32. Moreover, it is worth noticing that the overhead is nearly constant although the number of CPUs varies (3.39–4.55 s). This behavior is expected, as the overhead introduced at runtime depends linearly on the number of invocations of communication calls. Since each process executes roughly the same code (except border processes that differ slightly), the number of communication calls is constant and is thus the overhead.

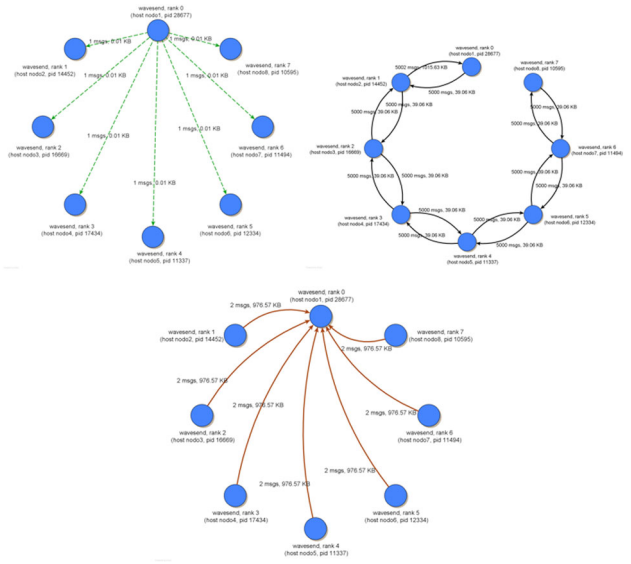

Fig. 22 Visualization of WaveSend communication phases: initial data broadcast (*left*), main exchange phase (*right*), and final result collection (*bottom*)

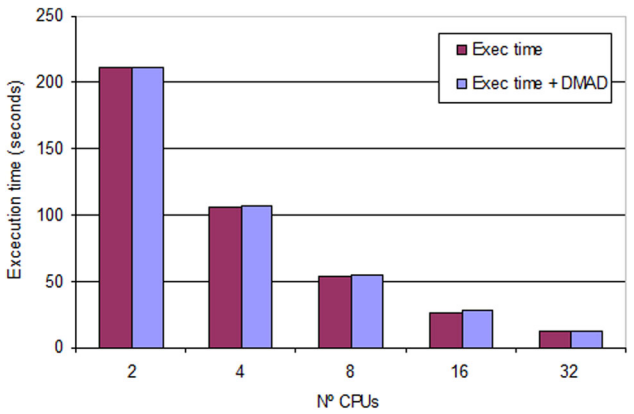

Fig. 23 WaveSend 1-D stencil execution times: large problem size (UAB cluster)

The size of the PTAG (see Table 1) saved in XML format (without behavioral clustering) is very small and scales quasi-linearly with the number of CPUs.

To verify how our tool behaves on a larger scale, we have conducted experiments on the MareNostrum supercomputer. However, the lack of availability of the Dyninst API on PPC32/PPC64 platform (at the moment of conducting these experiments) prevented us from using dynamic instrumentation. Nevertheless, we have developed a statically linked support library that uses PMPI interface to intercept communication calls. Although we were not able to detect and instrument loops and insert/change instrumentation on-the-fly, we were able to test it on a large scale. Our simplified tool intercepts MPI calls, builds a TAG at each process and uses MPI piggyback to connect the TAGs. Finally, the tool dumps the abstraction at the end of the execution and a post mortem tool merges them into the PTAG.

Figure 24 presents the execution results of this simplified tool for large problem sizes on the MareNostrum supercomputer, with CPUs varying from 2 to 512 (note logarithmic scale on Y axis). See Table 2 for details. Our observation is that the overhead of our tool is low (0.2–5.1 %), except for execution on 128 and more CPUs (9.6–35.8 %). However, these executions were extremely short (seconds), and the overall instrumentation cost with relation to the application code was significant. This relationship would change with bigger problem sizes. The important fact is that the introduced overhead was maintained nearly constant when adding more CPUs, which confirms the scalability of our implementation.

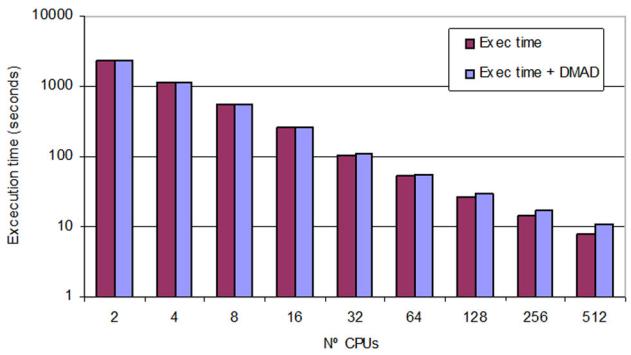

Fig. 24 WaveSend 1-D stencil execution times: large problem size (MareNostrum)

Table 1 Comparison of WaveSend execution times with and without DMAD: large problem size (UAB cluster)

No. CPUs	Exec time (s)	Exec time + DMAD (s)	Overhead (s)	Overhead (%)	PTAG size (KB)	No. snapshots
2	2073.51	2077.89	4.38	0.21	9.36	70
4	1038.74	1043.29	4.55	0.44	20.96	36
8	530.38	534.55	4.17	0.79	45.11	19
16	267.57	271.54	3.97	1.48	92.71	10
32	132.76	136.15	3.39	2.55	195.16	5

Table 2 Comparison of WaveSend execution times with and without DMAD: large problem size (MareNostrum)

No. CPUs	Exec time (s)	Exec time + DMAD (s)	Overhead (s)	Overhead (%)	PTAG size (KB)
2	1898.91	1902.42	3.51	0.18	9
4	1122.91	1127.21	4.3	0.38	21
8	545.19	548.94	3.75	0.69	45
16	258.82	262.18	3.36	1.30	93
32	106.90	109.29	2,39	2.23	195
64	53.08	55.76	2.68	5.06	421
128	27.21	29.82	2.61	9.60	927
256	14.46	17.16	2.7	18.69	2092
512	8.08	10.98	2.9	35.84	4822

5.2 NAS LU benchmark (SPMD)

NAS LU is a well-known simulated CFD application benchmark that employs a symmetric successive over-relaxation (SSOR) scheme to solve a regular-sparse, lower and upper triangular system. This application is written in Fortran and represents a typical SPMD code [2].

This code requires a power-of-two number of processors. A 2-D partitioning of the grid onto processors occurs by dividing the grid into two equal parts repeatedly in the first two dimensions, alternately X and then Y, until all power-of-two processors are assigned. This results in vertical pencil-like grid partitions on the individual processors. The ordering of the point-based operations that constitute the SSOR procedure proceeds on diagonals, which progressively sweep from one corner on a given Z plane to the opposite corner of the same Z plane, and then proceed on to the next Z plane. Communication of partition boundary data occurs after the completion of computation on all of the diagonals that contact an adjacent partition. This constitutes a diagonal pipelining method called a wavefront method [4]. This benchmark is very sensitive to the small-message communication performance of an MPI implementation, as it sends large numbers of very small messages.

We have executed a B-class NAS LU benchmark on 4 processors (UAB cluster). Our tool needed ≈ 17 s at startup and caused less than 2 % of runtime overhead. The PTAG was exported into the XML format and its size was 274.8 KB. The first observation is that the program exhibits three main phases of execution. In the first phase, the process called *Task rank 0* broadcasts initial data to other processes (invoking an MPI_Bcast call several times). The volume of distributed data is very low (≃0.01 KB). Next, the wave-like communication takes place when each process exchanges data with its neighbors (1 exchange per neighbor pair, ≃406 KB). The pattern confirms 2D decomposition (2x2 grid), and we can identify the coordinates of each process. Afterwards, all of the processes synchronize on the barrier. There are no sightings of imbalance at this stage. The second phase is the main loop, composed of 250 iterations. In each iteration, all of the processes execute two loops to exchange the data with corresponding 2D neighbors and perform local calculations. The analysis of communication patterns reveals the communication scheme, composed of 5 steps:

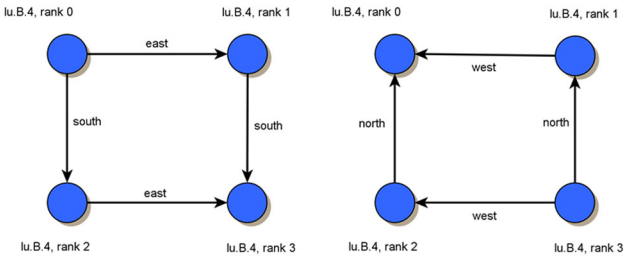

Fig. 25 NAS LU main loop communication patterns 1 and 2. South-east wavefront (*left*), and north-west wavefront (*right*). Each link transmitted 25,000 small messages (≈ 2 KB data each)

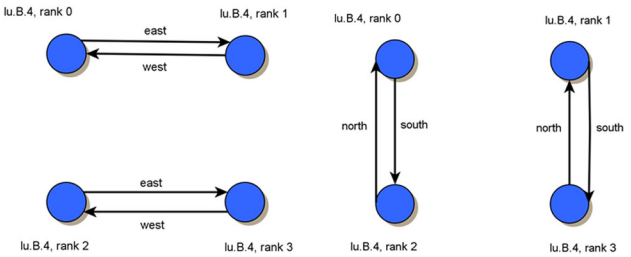

Fig. 26 NAS LU main loop communication patterns 3 and 4. East-west exchange (*left*), and south-north exchange (*right*). Each link transmitted 250 messages (≈ 377 KB data each)

1. South-east wavefront executed 100 times per iteration (see Fig. 25, left image)
2. North-west wavefront executed 1000 times per iteration (see Fig. 25, right image)
3. East-west exchange executed once per iteration (see Fig. 26, left image). This step might be preceded by a conditional all-to-all reduction

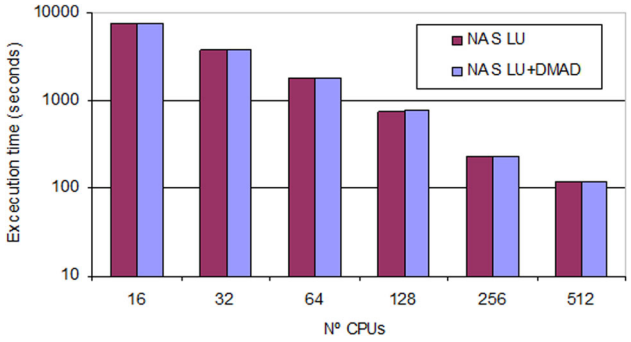

Fig. 27 NAS LU execution times: class D problem size (Marenostrum)

4. South-north exchange executed once per iteration (see Fig. 26, right image)
5. Final all-to-all reduction.

The behavior of each process depends on its location in the grid. With a 2 × 2 grid, all of the processes are border processes, and each is characterized with a different pattern. For larger grids (e.g., 4 × 4 and more), there would be multiple processes with all four neighbors (west, north, east, and south) and the same behavior. In the final phase, the application performs a number of reductions and terminates. The performance metrics show a relatively good load balance, as the majority of the time is spent on calculations and there are no sightings of communication inefficiencies.

To evaluate the overhead introduced by our tool, we have conducted a number of experiments, varying the number of CPUs in the Marenostrum supercomputer. We used class D problem size (grid 408 × 408 × 408, 300 iterations) to take advantage of its capacities. The comparison of execution times with and without the DMAD when varying the number of CPUs from 16 to 512 is shown in Fig. 27. Detailed data are presented in Table 3. Although this application sends large amounts of small messages, its computation to communication ratio is high. The resulting abstraction reveals that most of the time is spent on well-balanced calculations. This benefits our tool and causes the total overhead to be lower than 2 % in all cases. This indicates that our MPI piggyback scheme implementation is efficient for such scenarios.

5.3 NAS IS benchmark (SPMD)

The NAS Integer Sort (IS) kernel is a well-known benchmark [2] that is used for testing both integer computation speed and inter-process communication. The IS kernel ranks a large array of small integers as fast as possible using a bucket-sort method that is useful for particle method codes. Bucket sort [8] is the fastest sorting method because it does not perform any key comparisons. However, there are significant limitations to its usage, and it can be applied sufficiently only in uncommon situations. To do a bucket sort, a temporary array must be used in which the elements to be sorted are distributed based on their key fields. The distribution of n numbers requires n steps and, thus, the performance of bucket sort is O(N).

In the IS benchmark, the process called *Task rank 0* (master) generates a vector of integer data (keys) to be sorted using the pseudorandom number generator and Gaussian distribution. The sorting is performed according to the following scheme. The master process divides all existing keys in the number of keys/number of nodes parts. Each part must be distributed to a single process. First, the master sends each worker a message with the information that specifies the range and number of the keys. Next, it sends the data. Each process receives data from the master and samples it to arrive at a good load balance. It communicates with other processes in order to know their ranges. Then, it keeps all of the keys which fall within its range and sends the other keys to the appropriate nodes. Finally, it sorts all the keys in its range. In this benchmark, the communication costs are high (up to about 50 %). This is because the benchmark is dominated by all-to-all data exchange messages, since each processor sends the data which falls within the range of the recipient to all of the others.

We have executed a C-class IS benchmark on 8 processors (UAB cluster). The PTAG was exported into the XML format and its size was 75 KB. The abstraction reveals a simple structure of IS code. The execution is divided into three main phases. In the first phase, initialization, we observe that the MPI_Init call cost (≃14 s) is relevant with respect to the total execution time, which itself is short. Once connected, the processes perform local calculations (≃10 s)

Table 3 Comparison of NAS LU execution times with and without DMAD: class D problem size (Marenostrum)

No. CPUs	Exec time (s)	Exec time + DMAD (s)	Overhead (s)	Overhead (%)	No. snapshots	Mop/s
16	7555.98	7651.40	95.42	1.26	253	5280
32	3726.63	3776.52	49.89	1.34	125	10,706
64	1822.64	1848.56	25.92	1.42	62	21,890
128	754.12	765.46	11.34	1.50	26	52,906
256	228.82	232.47	3.65	1.59	9	174,366
512	118.44	120.70	2.26	1.91	5	336,872

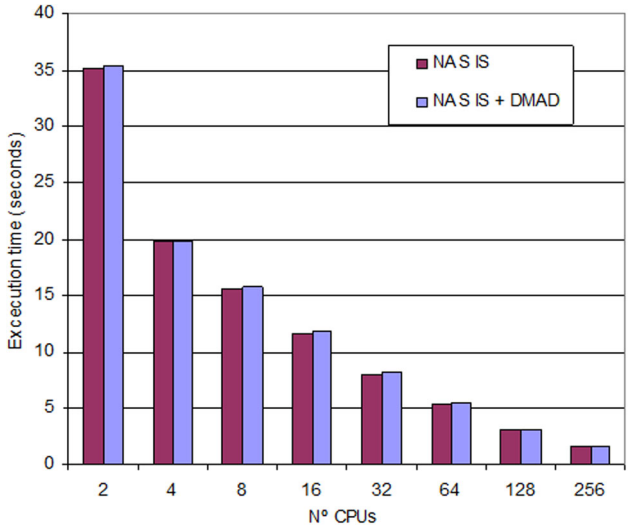

Fig. 28 NAS IS execution times: class C problem size (Marenostrum)

and then perform global initialization by performing global data reduction, all-to-all exchange, and, finally, personalized all-to-all exchange of ≃129 MB of data (MPI_Alltoallv). Nevertheless, all of the processes terminate this phase equally and with no sign of imbalance. The second phase is the main loop, composed of 10 iterations. In each iteration, all of the processes execute the same scheme, composed of global reduction, all-to-all exchange, personalized all-to-all exchange of data, and, finally, local calculation. The iteration time is dominated by communication activity due to relevant data volume (all-to-all with 129 MB of data in each iteration). However, the calculations and communications are well-balanced, and we did not observe any significant inefficiencies. In the final phase, the application reduces a small amount of data on the process *Task rank 0*, then propagates messages from *Task rank 0* sequentially up to the last process, performs the ultimate data reduction and terminates.

The comparison of execution times with and without the DMAD when varying the number of CPUs from 2 to 256 is shown in Fig. 28. We observed overheads ranging from 0.1–2.3 %.

5.4 SMG2000 (SPMD)

SMG2000 is a parallel semi-coarsening multigrid solver for the linear systems arising from finite difference, finite volume, or finite element discretizations of the diffusion equation on logically rectangular grids [7,23]. The code solves both 2D and 3D problems and is derived directly from the hypre library [24], a large linear solver library that is being developed at LLNL.

SMG2000 is an SPMD code written in C that uses MPI. Parallelism is achieved by data decomposition based on subdividing the grid into logical P × Q × R (in 3D) chunks of equal size. SMG2000 is a highly synchronous code. The communication and computation patterns exhibit the surface-to-volume relationship common to many parallel scientific codes. Hence, parallel efficiency is largely determined by the size of the data chunks mentioned above, and the speed of communications and computations on the machine. SMG2000 is also memory-access bound, doing only about 1–2 computations per memory access, so memory-access speeds also have a large impact on performance.

We have also abstracted the SMG2000 program executed on 8 processors (UAB cluster) for small problem sizes (35 × 35 × 35). PTAG was exported into the XML format and its size was 736 KB. Unfortunately, the visualization of the resulting abstraction is too big and not readable, as we can observe in Fig. 29.

With this application, we could see the limitations of our prototype tool implementation. SMG2000 makes intensive use of recursion (in fact, the hypre library is a highly recursive code). Data-driven recursion generates a huge number of distinct call-paths which introduce noise to structural patterns captured by the TAG. The fundamental assumption in the TAG is that each node is identified by a unique stack trace that ends in an MPI call. Typically, the number of possible distinct stack traces is limited by the application static structure. This assumption changes in the presence of recursion when it becomes data dependent. In the case of a multigrid solver, we have observed an explosion in the number of nodes. In particular, the same activities were abstracted multiple times when executed at different levels of recursion.

There are different ways of representing recursion more compactly [37]. The most common approach used by the majority of stack walking tools is to limit the considered stack trace to some maximum depth (e.g., the mpiP tool uses this approach [17]). Our intuitive approach is that any recursive implementation could be transformed to its iterative version. The solution to this problem is to dynamically cluster similar stack traces using an equivalence metric that combines similar stacks into the same representation. In this case, during the application execution, we would obtain a code structure correctly, and the abstraction would be understandable. We believe that this would reduce the possible call-path space and better reflect the structural code patterns. However, finding a cost-effective stack trace clustering method remains as future work.

5.5 XFire (master/worker)

We have selected XFire as an example application based on Master/Worker paradigm. XFire is a forest fire propagation simulator [19] that calculates the expansion of the fireline considering the initial fireline position and different aspects, such as weather (wind, temperature, moisture), vegetation

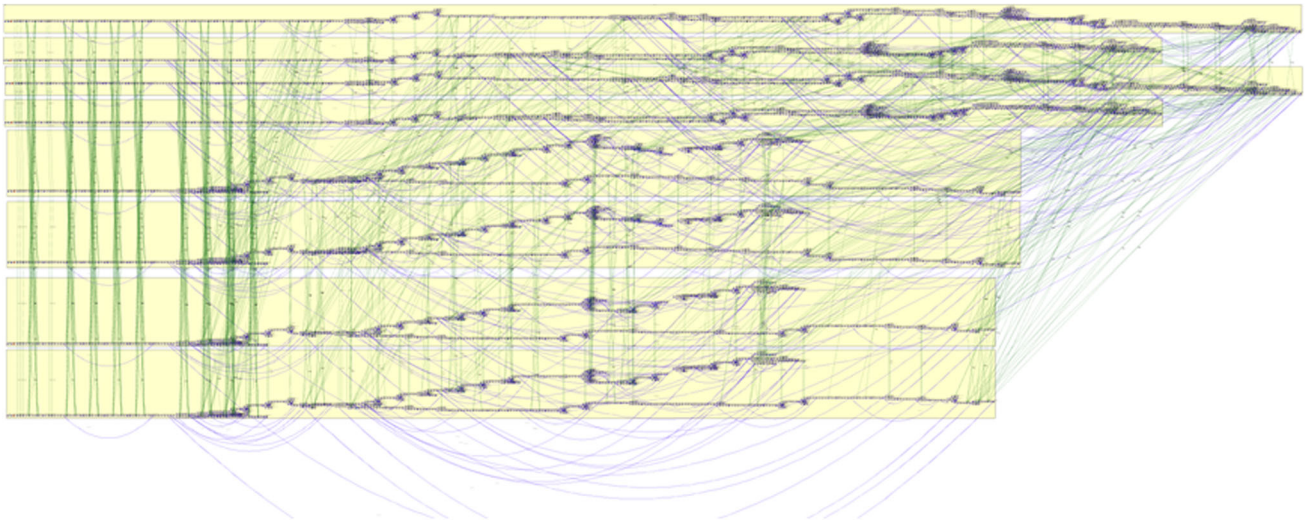

Fig. 29 PTAG for SMG2000 application. The structural patterns of recursive codes are not well captured due to call-path explosion

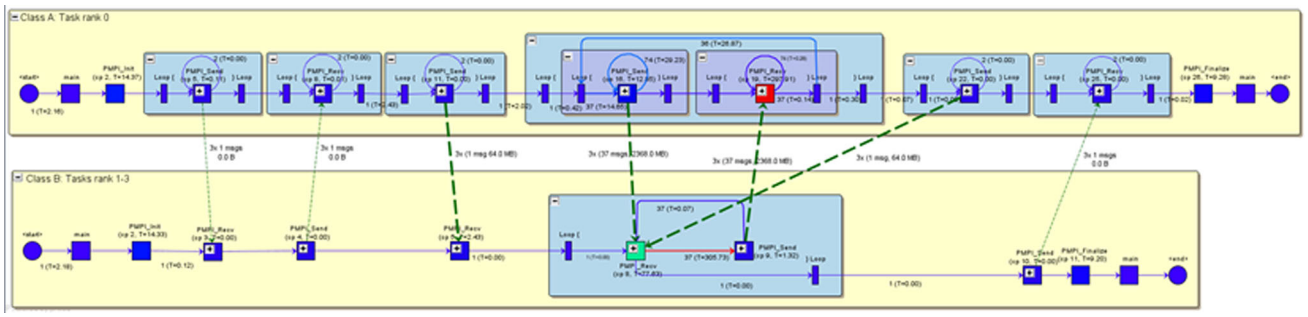

Fig. 30 Scalable PTAG visualization of XFire application after behavioral clustering

and topography (terrain). This application represents a typical computationally intensive Master/Worker code and is written in C++.

Figure 30 presents the visualization of the PTAG of the XFire application executed on 4 processors after applying the behavioral clustering. As expected, we observe two behavioral classes of processes: Class A, which corresponds to the process *Task rank 0* (master), and Class B, which corresponds to processes *Task rank 1–3* (workers). The visualization is scalable to an arbitrary number of workers. The abstraction indicates three phases of execution. We can interpret the first phase as initialization, which includes the synchronization and distribution of 64 MB of data from master to workers. The next phase is the main loop. In each iteration, the master sends requests to the workers. Statistics suggest one request per worker (with 64 MB of data). Next, it waits for the results from all of the workers (64 MB of data received from each worker), and, finally, performs some calculations. Each worker waits for a request, processes it and then responds. Finally, we observe the coordinated termination phase. The performance metrics reveal the existence of severe bottlenecks in the second phase. The MPI_Recv node in the master

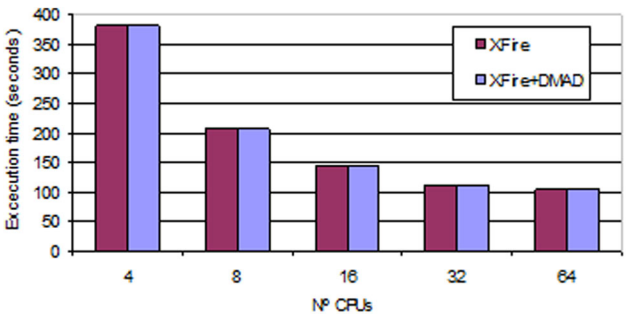

Fig. 31 XFire execution times (MareNostrum)

with 77.4 % of the total time indicates the bored master waiting for calculating workers. The MPI_Send node in the workers reveals an inefficiency (20.8 %) that we manually infer to be caused by master calculations at the end of each iteration. This indicates the opportunity for potential optimizations that could bring significant benefits.

We present overheads in Fig. 31. The experiments were conducted varying the number of CPUs from 4 to 64. We performed the tests with a higher number of CPUs, but it did not provide any additional speedup. This effect is due to the

Table 4 Comparison of XFire execution times with and without DMAD (MareNostrum)

No. CPUs	Exec time (s)	Exec time + DMAD (s)	Overhead (s)	Overhead (%)	PTAG size (KB)	No. snapshots
4	380.71	380.60	−0.11	−0.03	18.77	14
8	208.29	207.46	−0.83	−0.40	36.43	8
16	144.60	144.59	−0.01	−0.01	71.49	6
32	111.13	110.89	−0.24	−0.22	141.84	5
64	105.42	105.10	−0.32	−0.30	281.60	5

limited scalability of the Master/Worker approach as the master process quickly becomes a bottleneck when managing a higher number of workers. Table 4 lists the detailed measurements registered during the experiments. One interesting fact is that the total overhead is constant and negative in all cases (up to 0.4 %). This peculiar effect might be attributed to three facts. First, there is a small number of exchanged messages and, thus, low overhead. Second, the inherent inefficiency in all the program's processes that are mostly blocked waiting on messages hides the monitoring overhead. Third, the measurement error might be significant, given the small differences between compared times.

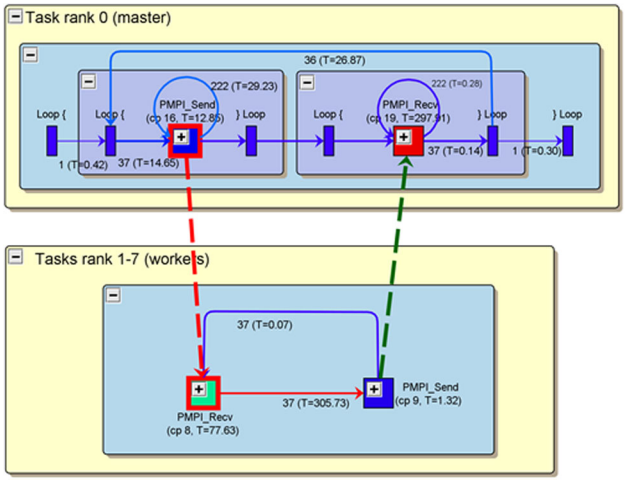

Fig. 32 Details of the problematic execution paths in master and workers

5.6 Use case of the abstraction-based analysis

As indicated in Sect. 3.4, the proposed application abstraction can be used for performance analysis in order to identify bottlenecks and their root causes. To show its applicability, we have chosen the abstraction of the XFire application presented in Fig. 30.

To identify performance problems, the first step is to determine activity rankings for each process. Analyzing the application abstraction, we prepare a list of the top activities that consume the most execution time. For example, we can observe the existence of inefficiency resulting from MPI_Recv (cp19), as, on average, almost 47 % of the time, the master is blocked waiting on incoming messages with results from slaves. The second and third top-activities are the computational edges 14 → 14 (some computation between consecutive sends that consumes 13 % of the time) and 18 → 13 (calculations of a global abstraction that consume 22 % of the time). The behavior of all of the workers is similar. The top activity is the computational edge 7 → 8 (representing the execution of a local fire propagation model) that consumes about 41 % of the time. This indicates some kind of performance problem, as workers are expected to maximize efficiency and avoid idle times. Effectively, in these processes, we observe an inefficiency in a blocked receive call (MPI_Recv, cp8) that is used to receive work requests from the master. We may interpret that the workers are blocked, waiting to receive work from the master. This inefficiency reaches 36 %, which indicates a severe performance problem. The most problematic execution paths are zoomed in on in Fig. 32.

The second step is the in-depth analysis of each individual problem. The blocked receive call in the master process revealed the existence of 7 communication links with workers where the distribution of waiting times per process is not uniform. This manifests the existence of a synchronization problem in the form of multiple late senders and one receiver. Moreover, the analysis of top-activities in the worker diagnosed the existence of multiple instances of a late sender problem, i.e., each worker is waiting to receive work from the master. The identified performance problems are presented in Table 5.

Finally, once the performance problems are determined, we search for causal relationships between the synchronization problems. To analyze the instances of a late sender problem in the workers, we compare corresponding causal execution paths. Example causal execution paths are presented in Fig. 33.

Our algorithm identified the following activities with the largest contribution to the length of the master's path as the causes of the late sender:

Table 5 List of performance problems found after the first analysis interval of the XFire application

No.	Problem	Location	Details	Severity at source process (%)
1	Multiple late senders one receiver	Task rank 0	MPI_Recv(Task rank 0, cp19), MPI_Send(Task rank 1–7, cp9)	47.32
2	Computation	Task rank 0	Edge 14 → 14	13.11
3	Computation	Task rank 0	Edge 18 → 13	8.93
4	Computation	Task rank 1–7	Edge 7 → 8	41.18
5	Late sender	Task rank 1–7	MPI_Recv(Task rank 1–7, cp8), MPI_Send(Task rank 0, cp16)	36.45

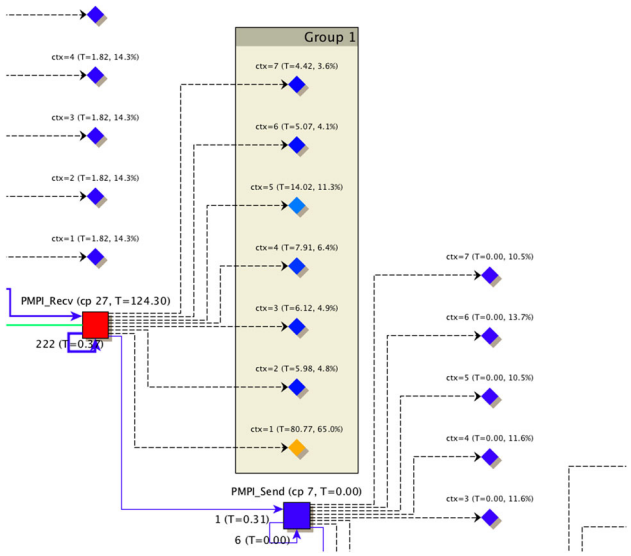

Fig. 33 Example execution paths for XFire application

– *MPI_Recv* (cp19) reception of work from all of the workers.
– *Back loop edge* 18 → 13 global model calculations in the master.
– *Edge between sends* 14 → 14 source code examination revealed that this edge corresponds to message packing with MPI_Pack calls.
– *Edge* 13 → 14 fireline data memory management.

The path comparison revealed the following findings:

– *Edge* 7 → 8 source code examination helped us determine that the activity represents the local model calculations performed by the worker (79 ± 1 % contribution to the inefficiency in the master). Our interpretation is that the master distributes the work and immediately waits for the results. The slaves receive the requests and require time to calculate the response. This time contributes to almost 80 % of the blocked receive problem.
– *MPI_Recv* call (cp8) reception of work by the worker.

We may conclude that the master is implemented to wait for the workers to finish calculations. On the other hand, the workers lose time waiting for work assignments, because the master processing time for assigning work is significant. This effect is attributed to three main causes: global model calculations in the master, message packing during work distribution, and the fireline data management code.

To relax the scalability limitations of XFire, we recommend:

– Overlapping the reception of results (communication) with global model calculations in edge 18 → 13 of the master. This optimization could help reuse waiting times in the receive call to prepare fragments of input for the next iteration.
– Avoid using explicit message packing (edge 14 → 14 in the master) and instead use derived data types.
– Pre-allocate and initialize the memory necessary to manage the fireline data before the main loop instead of managing it dynamically for each iteration.

In order to verify the correctness and the influence of the suggested recommendations, we have manually changed the source code of the XFire application, applying them to the corresponding places. Finally, we have compared the original version of the application with the tuned code where the suggested recommendations were taken into account. As we can see in Fig. 34, the execution time is reduced by up to 27 % when applying the recommendations. A detailed explanation of this evaluation is presented in [43].

5.7 Evaluation of overheads

We classified and evaluated the overheads caused by our prototype tool as follows.

5.7.1 Offline startup overhead

At startup, each tool daemon performs four actions. First, it starts up and connects to the parent process. Second, the daemon parses the program executable using Dyninst. The cost

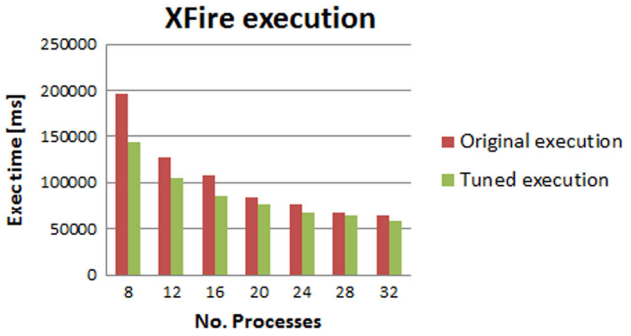

Fig. 34 Comparison of the XFire execution time for the original and tuned versions

of this action depends on the executable size. Third, it starts the application process, loads the RTLib library (158 KB), parses it and performs initialization. Fourth, the daemon instruments the executable. This includes wrapping MPI calls that are used by the application, and inserting instrumentation into target functions. This cost depends on the number of MPI calls used and the number of predefined target functions and loops selected during static code analysis. We have observed that this cost is higher for Fortran programs. To summarize, the total startup cost is constant for a given executable. Table 6 lists a detailed breakdown of startup times for evaluated applications.

5.7.2 Online TAG construction overhead

This includes the runtime cost of executing event tracing instrumentation, walking the stack, finding a call-path identifier in the hash table, updating the graph nodes, edges and metrics. The penalty is nearly constant. Overall, application overhead depends mainly on the number of communication calls invoked and instrumented loop entries/exits. Additionally, this overhead increases when there are more performance metrics to be evaluated.

5.7.3 Online TAG sampling overhead

Each daemon periodically samples the TAG. To take a snapshot, the daemon copies a contiguous block of memory shared with the application process to its local memory. The cost of this action depends on the graph size, which in turn reflects the program structure.

5.7.4 Online MPI piggyback overhead

This includes the cost of the wrapped MPI calls and datatype wrapping overhead for small payloads and the cost of sending an extra message for large payloads. We compared the results of the SKaMPI Benchmark [38] for original MPI and MPI with an active piggyback mechanism. As illustrated in Fig. 35, for point-to-point operations the absolute overhead is nearly constant when varying message size. Its impact decreases from 10 % for messages smaller than 1 KB to 2 % for 1 MB messages. We observed similar effects for collective operations where overhead varied from 15 to 0.5 %. This is illustrated in Fig. 36. Although these overheads might be considered relevant for some scenarios, we did not observe important impacts when evaluating application-level overheads.

The included experiments match well with those presented in [41]:

– Depending on the message size, packing data and piggyback together leads to a worse performance, and, hence, it is better to send two separate messages.

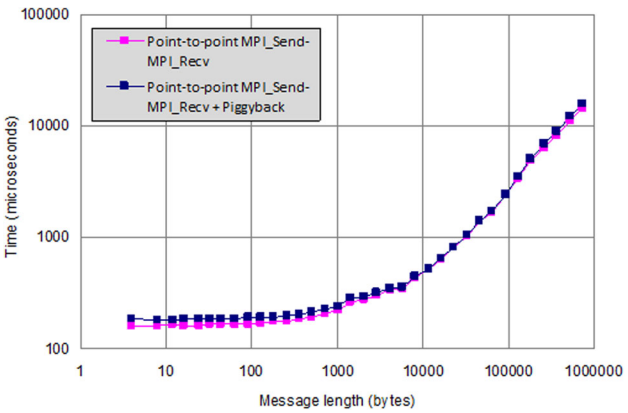

Fig. 35 MPI piggyback overhead for point-to-point, send-receive pattern

Table 6 Breakdown of startup times for evaluated applications

Program	Language	Size (KB)	No. modules	No. funcs	Parsing	Load lib. (%)	Preinstr. (%)	Other (%)
WaveSend	C	15.4	15	6952	44	36	17	3
NAS IS	C	25.6	17	6971	22	34	15	29
NAS LU	Fortran	222.4	48	8126	16	29	39	16
Sweep3D	Fortran	95.5	30	8178	18	32	44	6
XFire	C	824.6	48	10,428	25	51	17	7
SMG2000	C	504.9	60	7289	21	32	23	24

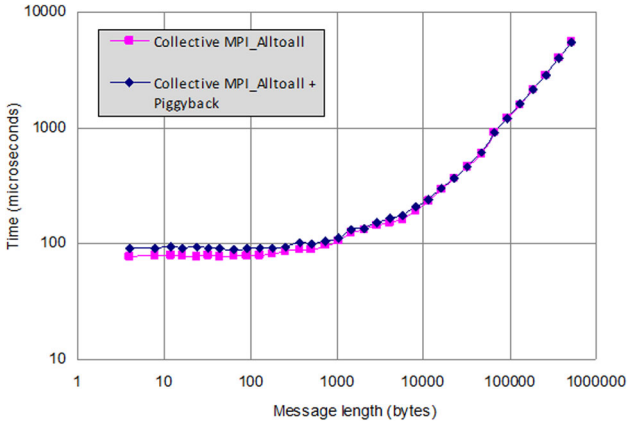

Fig. 36 MPI piggyback overhead for an all-to-all collective communication pattern

- A constant bandwidth varying piggyback message sizes is observed in some benchmarks, but latency depends directly on the piggyback size. It is not always necessary to translate into application overhead.
- For certain communication intensive benchmarks, an average overhead is around 10 %, similar to our tested case of the SkaMPI benchmark.
- Performance in piggyback depends on the MPI implementation, especially on the optimization of advanced mechanisms such as custom datatypes or message coalescing. Therefore, extending the MPI standard to include piggyback mechanisms would be beneficial.

5.7.5 PTAG construction overhead

After sampling the TAG, each daemon sends its snapshot to the parent process. The intermediate TBON processes receive snapshots from multiple processes, merge them and send them up the TBON tree. Finally, the front-end process merges the complete PTAG snapshot. However, all TBON processes (including front-end) are physically located on separate CPUs to avoid computational interference with the application processes. Therefore, we do not attribute the PTAG construction costs to the overhead induced in the application.

6 Related work

The concept of the detection of causal execution flows for cause-effect inference has been studied by Mirgorodskiy in his automated problem diagnosis thesis [29]. This work introduces self-propelled instrumentation, an approach for tracking the control-flow of a process and across process and kernel boundaries, which overcomes the limitation of application-specific tracing and introduces a low overhead. It introduces a flow separation algorithm, a statistics-rule-based approach for separating concurrent activities in a system. This approach allows for the identification of particular requests within the application with little user intervention. Their approach collects function-level control-flow traces, while our technique aggregates repetitive patterns while preserving probabilistic causality.

Our work is also related to communication pattern extraction and runtime MPI trace compression techniques. Noeth et al. in [34] proposed a framework for scalable trace compression and replay of communication traces. Their approach allows for a scalable, lossless, near-constant size trace compression scheme, regardless of the number of CPUs, while preserving structural information and a temporal ordering of events. Process-level MPI trace compression takes advantage of the regular nature of communication patterns, represented in the source code mostly by loops. This method drastically reduces the size of the individual process' trace. The compression algorithm maintains a queue of MPI events and uses a greedy pattern-matching scheme to compress the queue. To preserve the locations of MPI calls, the tracing framework identifies and records the calling sequence by logging the call sites of the calling stack in the stack walk. This feature is similar to our approach presented in this work. Both approaches use pattern matching to dynamically discover repeating communication structures in an MPI event graph. Our work is different in that it uses statically determined loops to find pattern boundaries. Our on-the-fly trace analysis is lossy, as it uses aggregation and builds statistical profiles.

There are also many similarities to the critical path profiling presented by Schulz in [40]. Their approach dynamically builds an execution graph at runtime by tracking communication activities and piggybacking data over MPI messages. While they focus on complete critical path extraction composed of individual events, our approach is different, as it represents the whole application with all of the executed paths together with their statistical execution profiles and aggregating repetitions.

Concerning application behavior, our approach is also related to Palm, described by Tallent et al. in [45]. Palm uses an annotated modeling language on an application source code to describe application execution. It allows for the creation of an analytical application model post-mortem that can then be used for performance prediction and application diagnostics reports. Palm combines top-down (human-provided) semantic insight with bottom-up static and dynamic analysis. Given annotated source code, Palm generates a model based on the static and dynamic mapping of annotations to program behavior.

As the generated abstraction may be used for performance problem diagnosis, our approach is also closely related to analysis tools. These are a set of tools that are based on

trace file analysis, such as Vampir [20], a commercial tool currently known as Intel Trace Analyzer. It uses trace files and provides a GUI to visualize a timeline of MPI events after application execution. This tool has its own mechanism to trace the application called VAMPIRtrace. It is a library with MPI tracing wrappers and records point-to-point communications, collective operations, MPI I/O operations and user-defined procedures. When the user-defined procedure is to be traced, the VampirTrace API must be used to modify the application and, hence, the recompilation phase is required.

The Scalasca toolset, described by Geimer et al. in [12], mainly focuses on trace analysis to produce post-mortem reports in a 3D dimension space view. It is oriented to search a list of problems and locate them in the program. Using the CUBE tool, it performs an interactive exploration of the produced data for providing performance metrics and diagnosis to the analyst, who can then take specific actions to eliminate or minimize performance bottlenecks. Scalasca includes different tools to combine or manipulate various generated reports, allowing comparisons and aggregations, focus on specific parts of the reports, and generating additional performance metrics.

Another tool is TAU (Tuning and Analysis Utilities), described by Malony et al. in [27]. This is a portable profiling and tracing toolkit that uses the static instrumentation approach for the performance analysis of parallel programs written in C, C++ and several other languages. To perform tracing, TAU provides an instrumentation API and tools to facilitate source code instrumentation. It also provides call-path specific profiling. The TAU traces function entries and exits via instrumentation code inserted into the program and uses this information to generate a stack of currently executing functions. TAU uses the stack of active functions as a representation of the current call-path, so their approach is based on whole-program instrumentation.

Furthermore, to address more types of parallel applications, we could complete our technique by adding support for I/O activities. The problems of I/O operations on an extreme scale and how to characterize these systems are presented by Wiedemann et al. in [49]. We believe it would be straightforward to intercept and abstract both sequential I/O calls (e.g., open, create, write, read, flush) and parallel MPI I/O (e.g., MPI_File_read, MPI_File_write). By abstracting individual I/O calls and providing I/O specific knowledge (e.g., specification of I/O inefficiencies), our approach could be used to understand the behavior of I/O-based codes.

7 Conclusions

We have presented an online performance abstraction technique and its prototype implementation, which automates the discovery of causal execution paths, made up of communication and computational activities, for arbitrary message-passing parallel programs. Multiple executions of a defined activity are reflected in the TAG just by one instance, and its behavior is statistically summarized in an execution profile. These factors contribute to the compactness of the application performance abstraction while reflecting high-level communication patterns. This approach enables autonomous and low-overhead execution monitoring that generates performance knowledge about application behavior for the purpose of online performance diagnosis. By following the flow of control and intercepting communication between processes at runtime, the cornerstone of this technique is the ability to discover causal execution paths through high-level application structures, such as loops and communication operations, and characterize them with statistical execution profiles. We have developed a set of techniques based on dynamic instrumentation that enables the online construction of such abstractions with acceptable intrusion on application execution. Our technique maintains a trade-off between large volumes of collected data and a preserved level of details. We take advantage of tracing, but avoid the high data volume problem by consuming the data on-the-fly, using lossy trace compression. When the analysis needs more detailed data, our technique enables the collection of causal execution paths that capture selective traces that preserve event ordering and low-level information.

Furthermore, we have proposed a scalable PTAG construction technique that can be used on HPC systems to analyze large-scale scientific applications. Our scalable TAG merging algorithm works best for applications that are composed of groups of identical processes that perform similar repetitive activities. Our approach enables the evaluation of the performance of individual processes and provides a high-level view of the entire application. It can be used to shorten the performance understanding process and serve as a base for developing a variety of online analysis techniques. We have demonstrated this ability in several real-world MPI applications. The abstraction of these applications reveals details about their behavior and structure, without requiring explicit knowledge or a source code. In all scenarios, our online performance abstraction technique proved effective for the low-overhead capturing of a program's behavior and facilitated performance understanding.

As future work, we are planning to improve the scalability of the DMAD tool to perform tests on thousands of processors. For this purpose, we are planning to improve the functionality of the global front-end for scalable TAG collection by distributing PTAG clustering to a tree-based overlay network infrastructure. We have used this technique as the basis of our investigation of automated root-cause performance analysis, and, hence, we plan to follow this research line. Moreover, we are planning to develop additional functionality for the DMAD, as it has certain limitations: poor

management of recursive functions, no support for I/O MPI or one-sided operations of MPI-2 and MPI-3 .

Acknowledgments The authors thankfully acknowledge the computer resources, technical expertise and assistance provided by the BSC-Marenostrum. This work has been supported by MINECO-Spain under contract TIN2014-53234-C2-1-R and GenCat-DIUiE(GRR) 2014-SGR-576.

References

1. Unwind Library Project. http://www.nongnu.org/libunwind/ (2012). Accessed Apr 2016
2. Bailey, D.H., Harris, T., Saphir, W., der Wijngaart, R.V., Woo, A., Yarrow, M.: The NAS parallel benchmarks 2.0. Tech. Rep., NASA Ames Research Center, Report NAS-95-020 (1995)
3. Barcelona Supercomputing Center: MareNostrum system architecture. http://www.bsc.es/marenostrum-support-services/marenostrum-system-architecture (2014). Accessed Apr 2016
4. Barszcz, E., Fatoohi, R., Venkatakrishnan, V., Weeratunga, S.: Solution of regular, sparse triangular linear systems on vector and distributed-memory multiprocessors. Tech. Rep., NASA Ames Research Center, Technical Report NAS RNR-93-007 (1993)
5. Bernat, A.R., Miller, B.P.: Incremental call-path profiling. Concurrency **19**(11), 1533–1547 (2007)
6. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S., Leipert, S., Mutzel, P., Junger, M.: GraphML progress report, structural layer proposal. In: Graph Drawing—8th International Symposium, GD 2000, Colonial Williamsburg, VA, USA, pp. 501–512 (2001)
7. Brown, P.N., Falgout, R.D., Jones, J.E.: Semicoarsening multigrid on distributed memory machines. SIAM J. Sci. Comput. **21**(5), 1823–1834 (1999)
8. Preiss, B.R.: Bucket sort. http://www.brpreiss.com/books/opus5/html/page512.html (1998). Accessed Apr 2016
9. Buck, B., Hollingsworth, J.K.: An API for runtime code patching. Int. J. High Perform. Comput. Appl. **14**(4), 317–329 (2000)
10. Choi, J.D., Miller, B.P., Netzer, R.H.B.: Techniques for debugging parallel programs with flowback analysis. ACM Trans. Program. Lang. Syst. **13**(4), 491–530 (1991)
11. Fox, G.C., Johnson, M.A., Lyzenga, G.A., Otto, S.W., Salmon, J.K., Walker, D.W.: Solving problems on concurrent processors. In: General Techniques and Regular Problems, vol. 1. Prentice-Hall, Inc., Upper Saddle River (1988)
12. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurr. Comput. **22**(6), 702–719 (2010)
13. Gerndt, M., Ott, M.: Automatic performance analysis with periscope. Concurr. Comput. **22**(6), 736–748 (2010)
14. Giménez, J., Labarta, J., Pegenaute, F., Wen, H., Klepacki, D., Chung, I.H., Cong, G., Voigtländer, F., Mohr, B.: Guided performance analysis combining profile and trace tools. In: Euro-Par 2010 Parallel Processing Workshops, vol. 6586, pp. 513–521 (2011)
15. Hafeez, M., Asghar, S., Malik, U.A., ur Rehman, A., Riaz, N.: Survey of MPI implementations. In: DICTAP (2), Communications in Computer and Information Science, vol. 167, pp. 206–220. Springer (2011)
16. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '05, pp. 48–60. ACM, New York (2005)
17. Vetter, J., Chambreau, C.: mpiP: lightweight, scalable MPI profiling. http://mpip.sourceforge.net/ (2013). Accessed Apr 2016
18. Jorba, J., Margalef, T., Luque, E.: Performance analysis of parallel applications with KappaPI 2. In: PARCO, pp. 155–162 (2005)
19. Jorba, J., Margalef, T., Luque, E., Andre, J.C.S., Viegas, D.X.: Application of parallel computing to the simulation of forest fire propagation. In: Proceedings of the 3rd International Conference in Forest Fire Propagation, vol. 1, pp. 891–900 (1998)
20. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The vampir performance analysis tool-set. In: Tools for High Performance Computing—Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart, pp. 139–155. Springer (2008)
21. Kranzlmüller, D., Schaubschläger, C., Volkert, J.: An integrated record & replay mechanism for nondeterministic message passing programs. In: PVM/MPI, pp. 192–200 (2001)
22. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
23. Lawrence Livermore National Laboratory: SMG 2000 benchmark. http://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/smg2000_readme.html (2001). Accessed Apr 2016
24. Lawrence Livermore National Laboratory: Scalable linear solvers, hypre library. http://computation.llnl.gov/casc/linear_solvers/sls_hypre.html (2013). Accessed Apr 2016
25. Lee, B., Resnick, K., Bond, M.D., McKinley, K.S.: Correcting the dynamic call graph using control-flow constraints. In: International Conference on Compiler Construction, pp. 80–95 (2007)
26. Lorenz, D., Böhme, D., Mohr, B., Strube, A., Szebenyi, Z.: Extending scalascas analysis features. In: Cheptsov, A., Brinkmann, S., Gracia, J., Resch, M.M., Nagel, W.E. (eds.) Tools for High Performance Computing 2012, pp. 115–126. Springer, Berlin (2013)
27. Malony, A.D., Shende, S., Spear, W., Lee, C.W., Biersdorff, S.: Advances in the TAU performance system. In: Tools for High Performance Computing 2011—Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, ZIH, Dresden, September 2011, pp. 119–130. Springer, Berlin (2011)
28. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The paradyn parallel performance measurement tool. Computer **28**(11), 37–46 (1995)
29. Mirgorodskiy, A.V.: Automated problem diagnosis in distributed systems. Ph.D. Thesis, Madison, WI, USA (2006). AAI3245668
30. Mirgorodskiy, A.V., Maruyama, N., Miller, B.P.: Scalable systems software—problem diagnosis in large-scale computing environments. In: SC, p. 88 (2006)
31. Morajko, A., Caymes-Scutari, P., Margalef, T., Luque, E.: MATE: monitoring, analysis and tuning environment for parallel/distributed applications. Concurr. Comput. **19**(11), 1517–1531 (2007)
32. Morajko, O.: Online performance modeling and analysis of message-passing parallel applications. Ph.D. Thesis, Barcelona, Spain (2008)
33. Morajko, O., Morajko, A., Margalef, T., Luque, E.: On-line performance modeling for MPI applications. In: Euro-Par, pp. 68–77 (2008)

34. Noeth, M., Mueller, F., Schulz, M., de Supinski, B.R.: Scalable compression and replay of communication traces in massively parallel environments. In: IPDPS, pp. 1–11 (2007)
35. Noeth, M., Ratn, P., Mueller, F., Schulz, M., de Supinski, B.R.: ScalaTrace: scalable compression and replay of communication traces for high-performance computing. J. Parallel Distrib. Comput. **69**(8), 696–710 (2009)
36. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: a tool to visualize and analyze parallel code. Tech. Rep., In WoTUG-18, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya (1995)
37. Ratn, P., Mueller, F., de Supinski, B.R., Schulz, M.: Preserving time in large-scale communication traces. In: Proceedings of the 22Nd Annual International Conference on Supercomputing. ICS '08, pp. 46–55. ACM, New York (2008)
38. Reussner, R., Sanders, P., Prechelt, L., M++ller, M.: SKaMPI: a detailed, accurate MPI benchmark. In: In Vassuk Alexandrov and Jack Dongarra, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 52–59. Springer, Berlin (1998)
39. Roth, P.C., Arnold, D.C., Miller, B.P.: MRNet: a software-based multicast/reduction network for scalable tools. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, SC '03, p. 21. ACM, New York (2003)
40. Schulz, M.: Extracting critical path graphs from MPI applications. In: 2013 IEEE International Conference on Cluster Computing (CLUSTER), pp. 1–10 (2005)
41. Schulz, M., Bronevetsky, G., Supinski, B.R.: On the performance of transparent MPI piggyback messages. In: Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 194–201. Springer, Berlin (2008)
42. Shende, S., Malony, A., Morris, A., Wolf, F.: Performance profiling overhead compensation for MPI programs. In: 12th European PVM/MPI User's Group Meeting, LNCS, vol. 3666, pp. 359–367 (2005). Record converted from VDB: 12.11.2012
43. Sikora, A., Margalef, T., Jorba, J.: Online root-cause performance analysis of parallel applications. In: Parallel Computing, pp. 81–107 (2015). doi:10.1016/j.parco.2015.05.003
44. Spear, W., Malony, A.D., Lee, C.W., Biersdorff, S., Shende, S.: An approach to creating performance visualizations in a parallel profile analysis tool. In: Proceedings of the 2011 International Conference on Parallel Processing. Euro-Par'11, vol. 2, pp. 156–165. Springer, Berlin (2012)
45. Tallent, N.R., Hoisie, A.: Palm: easing the burden of analytical performance modeling. In: 2014 International Conference on Supercomputing, ICS'14, Muenchen, Germany, June 10–13, 2014, pp. 221–230 (2014)
46. University of Wisconsin: MRNet: a multicast/reduction network. http://www.paradyn.org/mrnet/ (2012). Accessed Apr 2016
47. University of Wisconsin, University of Maryland: DyninstAPI programmer's guide, paradyn parallel performance tools. http://www.dyninst.org/manuals/dyninstAPI (2013). Accessed Apr 2016
48. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984)
49. Wiedemann, M.C., Kunkel, J.M., Zimmer, M., Ludwig, T., Resch, M.M., Bönisch, T., Wang, X., Chut, A., Aguilera, A., Nagel, W.E., Kluge, M., Mickler, H.: Towards I/O analysis of HPC systems and a generic architecture to collect access patterns. Comput. Sci. Res. Dev. **28**(2–3), 241–251 (2013)
50. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. J. Syst. Archit. **49**(10–11), 421–439 (2003)
51. yWorks Company: yWorks, yEd—java graph editor. http://www.yworks.com/products/yed/ (2014). Accessed Apr 2016

Anna Sikora got the BS degree in computer science in 1999 from Technical University of Wroclaw (Poland). She got the MSc in computer science in 2001 and in 2004 the PhD in computer science, both from Universitat Autònoma de Barcelona (Spain). Since 1999 her investigation is related to parallel and distributed computing. She is currently involved in Spanish national projects and she participated in the European project Autotune. Her main interests are focused on high performance parallel applications, automatic performance analysis and dynamic tuning. She has been involved in programming tools for automatic and dynamic performance tuning on cluster and Grid environments.

Tomàs Margalef got a BS degree in physics in 1988 from Universitat Autònoma de Barcelona (UAB). In 1990 he obtained the MSc in Computer Science and in 1993 the PhD in Computer Science from UAB. Since 1988 he has been working in several aspects related to parallel and distributed computing. Currently, his research interests focuses on development of high performance applications, automatic performance analysis and dynamic performance tuning. Since 2007 he is Full Professor at the Computer Architecture and Operating systems department. He is an ACM member.

Josep Jorba is a professor of Computer Architecture and Operating Systems at the Universitat Oberta de Catalunya. His research interest include design of middleware for distributed Computing environments, and designing tools for automatic performance analysis of parallel applications in message passing, multicore or manycore environments. He has a PhD in computer engineering from Universitat Autònoma de Barcelona.