Evaluation of Profiling Tools for the Acquisition of Time Independent Traces

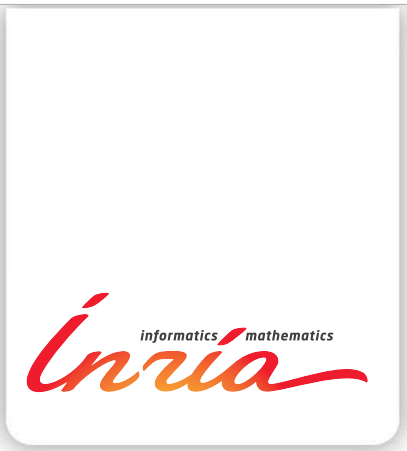
Frédéric Desprez, George S. Markomanolis, Frédéric Suter

Evaluation of Profiling Tools for the Acquisition of Time Independent Traces

Frédéric Desprez, George S. Markomanolis, Frédéric Suter

TECHNICAL REPORT
N° XXXX
July 2013
Project-Team AVALON

ISSN 0249-0803 ISRN INRIA/RT--XXXX--FR+ENG



Evaluation of Profiling Tools for the Acquisition of Time Independent Traces

Frédéric Desprez, George S. Markomanolis, Frédéric Suter*

Project-Team AVALON

Technical Report n° XXXX — July 2013 — 43 pages

Abstract: In a previous work, we proposed a framework for the off-line simulation of MPI applications. Its main originality with regard to the literature is to rely on time-independent execution traces. Time-independent traces are an original way to estimate the performance of parallel applications. To acquire time-independent traces of the execution of MPI applications, we have to instrument them to log the necessary information. There exist many profiling tools which can instrument an application. In this report we propose a scoring system that corresponds to our framework specific requirements and evaluate the most well-known and open source profiling tools according to it. Furthermore we introduce an original tool called Minimal Instrumentation that was designed to fulfill the requirements of our framework.

Key-words: MPI, Profiling tools, Traces, Performance Analysis, off-line simulation

* IN2P3 Computing Center, CNRS, Lyon-Villeurbanne, France

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Évaluation des outils de profiling pour l'acquisition de traces d'exécution indépendantes du temps

Résumé : Dans nos précédents travaux, nous avons proposé un environnement pour la simulation hors-ligne d'applications MPI. Sa principale originalité vis-à-vis de la littérature est de s'appuyer sur des traces d'exécution indépendantes du temps. Cela constitue une manière originale d'estimer les performances d'applications parallèles. Pour acquérir de telles traces indépendantes du temps lors de l'exécution d'applications MPI, nous devns les instrumenter afin de recueillir toutes les informations nécessaires. Il existe de nombreux outils de profiling permettant d'instrumenter une application. Dans ce rapport, nous proposons une méthode de notation correspondant aux besoins spécifiques de notre environnement et évaluons les outils de profiling open-source les plus connus selon cette méthode. De plus, nous introduisons un outil original, appelé Minimal Instrumentation, spécialement conçu pour répondre aux besoins de notre environnement.

Mots-clés : MPI, outils de profiling, traces, analyse de performance, simulation hors-ligne

1 Introduction

Nowadays there are many performance measurement tools which can provide useful information about the behavior of parallel applications and record the events that happen during the execution. These tools can be categorized into tracing, profiling and statistical profiling or a combination of these modes. The purpose of our framework is to predict the performance of a parallel application by replaying their time-independent traces [1],[2] using the SIMGrid simulation toolkit [3]. Since version 3.3.3, SIMGrid provides the option to describe an applicative workload as time-independent trace. So it is necessary to instrument an application and convert the measurement data into the appropriate format.

However, all this procedure raised an issue about which tool should be used for our framework. Thus we needed to evaluate them in order to conclude which ones are appropriate according to our requirements. In [1],[2], there is further information about the structure of the specific time-independent traces.

The remaining of this document is organized as follows. Section 2 describes the requirements of our framework that the profiling tools should comply with. Section 3 presents the evaluation scheme with the criteria and score method. The Section 4 provides the evaluation of all the participated tools. Finally we aggregate all the evaluation's scores in and we draw some conclusions in Section 5.

2 Requirements

In order to extract the time-independent traces from an application is necessary to acquire some specific data. For each event that occurs during the execution of an instrumented application, either is a computation or a communication operation, the amount of the executed flops or bytes is saved respectively. In this point, we should mention that although for the experiments on the current evaluations we measure the flops, we can change this metric to another one such as the total executed instructions which we used in [1],[2]. The main idea is to use a metric that corresponds to the computation part of the application.

Afterwards the tracing of an application and the extraction of the time-independent traces, we have a list of *actions* which correspond to the computations and the communications that took part during the execution of the instrumented application. An action is described by the *id* of the process that does the action, its *type*, e.g., a computation or a communication operation, a *volume* of flops or bytes and some parameters related to the action, such as the destination of the message in the case of an MPI_Send call or the sender process id for an MPI_Recv call.

For example, the left hand side of Figure 1 shows a simple computation executed on a ring of four processes. Each process computes one million flops and send one million bytes to its neighbor. The right hand side of this figure displays the corresponding time-independent trace.

As we can observe, the following requirements should be applied on the execution traces in order to be replayed:

- For compute parts, the trace should contain the id of the process that executes the instructions and the corresponding amount;
- For point-to-point communications, it is mandatory to know both the sender and the receiver of the message, the type (`MPI_Send`, `MPI_Recv`, etc.) and the message size in bytes;
- For collective communications, it is important to know at least the size of the message and the type of the MPI communication (`MPI_Broadcast`, etc.). However, there are some

```
for (i=0; i<4; i++){
 if (myId == 0){
  /* Compute 1M flops */
  MPI_Send(1MB,..., (myId+1));
  MPI_Recv(...);
 } else {
  MPI_Recv(...);
  /* Compute 1M flops */
  MPI_Send(1MB,...,
           (myId+1)% nproc);
 }
}
```

```
0 compute 1e6
0 send 1 1e6
0 recv 3 1e6

1 recv 0 1e6
1 compute 1e6
1 send 2 1e6

2 recv 1 1e6
2 compute 1e6
2 send 3 1e6

3 recv 2 1e6
3 compute 1e6
3 send 0 1e6
```

Figure 1: MPI code sample of some computation on a ring of processes (left) and its equivalent time-independent trace (right).

MPI calls such as `MPI_Allreduce`, where the participating processes execute also some computation, summing values or determining a maximum for example. In this case, it is mandatory to log the number of executed instructions in order to simulate the compute part;

- For synchronization calls such as `MPI_Barrier`, it is mandatory to know the MPI Communicator.

To select the tool that corresponds the most to the requirements expressed by our *time-independent* traces replay framework, we conducted an evaluation based on the following criteria and scoring method.

3 Evaluation Criteria and Scoring Method

Many profiling tools can provide useful information about the behavior of parallel applications and record the events that occur during their execution. PAPI[4] provides access to the hardware counters of a processor. In our prototype framework we measured the compute parts of an application in numbers of floating point operations (or flops). However, using another counter, such as the number of instructions, has no impact on the instrumentation overhead as long as the chosen measure does not combine the values of several counters. To select the tool that corresponds the most to the requirements expressed by our *time-independent* traces replay framework, we conducted an evaluation based on the following criteria and scoring method:

3.1 Profiling Features

Many tools exist to profile parallel applications, i.e., get an idea of the behavior of the application over its execution time. A classification of these tools can be made following two axes. The first distinction depends on the type of output produced by the considered tool. It can be either a *profile* or a *trace*. A profile gives a higher view of the behavior of an application as information is grouped according to a type of operation or to the call tree. A trace gives lower level information as every event is logged without any aggregation. The second way to classify tools depends on the kind of events they can be profiled or traced. Such events are mainly related to communication

or computation. A third important parameter for this evaluation is the type of value associated to each logged event, e.g., volumes or timings. Finally, the instrumentation of an application is mandatory to obtain a profile or a trace. Such a instrumentation can be applied to the entire application or to a specific block of code, e.g., , a particular function or a MPI call. In this evaluation we give a higher score to tools offering an automatic instrumentation feature. Such a feature lighten the burden of getting a profile from a user point of view.

With regard to the specific requirements of our time-independent trace replay framework, we adopt the following scoring scheme for the *Profiling feature* criterion:

- Tracing → 2 points
- Information on communication → 1 point
- Information on computation → 1 point
- Information on communication volumes (in bytes) → 1 point
- Information on computation volumes (in flops) → 1 point
- Automatic instrumentation of the complete application → 1 point
- Automatic instrumentation of a block of code → 1 point

Each tool can then obtain a score ranging from 0 to 8, the higher being the better.

3.2 Quality of Output

Profiling tools usually output the results in files. These files could be of various formats such as plain text, XML, or binary formats. As mentioned earlier our objective is to convert such output files into set of actions to be replayed by SimGrid. The readability of the output files is then a very important requirement. However, tools that produce binary files often come with an application or an API to extract the required information. Finally we aim at replaying executions that involve large numbers of processes. To ensure a good scalability, it is important that the trace of each process is stored in a separate file. Indeed a tool enforcing a unique trace file for the complete execution will have a poor scalability due to merging overhead. Moreover as we intend to read these traces and extract the *time-independent* ones, if there is at least one trace file per node, then we can use the same number of nodes with a parallel application to achieve our purpose. This means that even for large trace sizes we would be able to use many processors and hard disks. So the conversion will be much faster than using just one node. Then we propose the following scoring scheme (ranging from 0 to 3) for the *Output quality* criterion:

- Capacity to extract the required information → 1 point
- Direct extraction → 1 point
- One pass conversion → 1 point

3.3 Space and Time Overheads

As already mentioned, files that contain all the relevant information are created during the tracing or profiling of an application. Profiling provides aggregated data about an application. This means for example that it records the number of times a function is called and its average duration. Conversely tracing records each call independently with the timestamp and some other

information which depend on what the user wants to measure. The source of the time overhead for an instrumented application are the measurement of a PAPI counter on the processor, the instrumentation of the MPI communications, and writing traces on disks. In terms of space overhead, profiling obviously leads to smaller traces, as logged events are aggregated.

Then, for *Space and Time overhead*, we adopt the following scoring scheme:

- Overhead $< 50\%$ of the initial execution time → 2 points
- $50\% < Overhead < 100\%$ of the initial execution time → 1 point
  The executed time of the instrumented application includes the time needed to write the produced traces on disk.
- Linear increase of the overhead → 1 point
- Constant overhead → 2 points
- Linear decrease of the overhead → 3 points
- Linear increase of trace size with regard to problem size → 1 point
- Linear increase of trace size with regard to number of processes → 1 point
- Any special technique integrated in the tool to reduce the time overhead → 1 point

Each tool can then obtain a score ranging from 0 to 8, the higher being the better.

3.4 Software Quality

Sometimes installing a profiling tool is not just a simple procedure such as executing the commands `./configure`, `make`, and `make install` in a Linux environment. In some cases many flags should be added to install a tool, or dependencies on other tools should be taken under consideration. Moreover the tools that are released under open source license are preferred compared to the ones with any non open source license. Every tool should support a variety of hardware e.g., processors and software developed under various programming languages e.g., C/C++/Fortran or implementations of the Message Passing Interface e.g., Mpich, OpenMPI. A well written manual is also always helpful for both installing and using a tool. It also has to be up to date and inform the user about new features. Finally it is desirable that the tool is maintained in order to support all the new versions of MPI and PAPI. As result for the *software quality* criterion we adopt the following scoring scheme:

- Ease of installation → 1 point
- Software dependencies → 1 point
- License → 1 point
- Hardware compatibility → 1 point
- Support of C/C++/Fortran programming language, 1 point for each → 3 points
- Compatibility with the major MPI implementations, i.e., Mpich or OpenMPI → 1 point
- Documentation → 1 point
- The project is active → 1 point
- There is a support team for this tool → 1 point

Each tool can then obtain a score ranging from 0 to 11, the higher being the better.

4 Evaluation of Tracing and Profiling Tools

Based on the aforementioned evaluation criteria, the following tools: PAPI [4] which gives access to the performance counters, PerfBench [5] which can instrument only compute parts, PerfSuite [6] that provides statistical profiling for the compute parts, MpiP [7] which can apply statistical profiling for the MPI communication, MPE [8, 9] that instruments MPI communication operations, IPM [10] which supports profiling for MPI communication and compute parts. The other tools provide both profiling for communication and compute parts: Extrae [11], Scalasca [12], TAU [13], VampirTrace [14, 15], Score-P[16], and our contribution, called Minimal Instrumentation. In this section, we only include the evaluations of a selected subset of tools that are representative.

4.1 PAPI

The Performance Application Programming Interface (PAPI) is a tool developed at the University of Tennessee, Knoxville [4]. We based our evaluation on the version 3.7.0 while the last one is version 5.1.0. The differences with the last version are the support of multiple components, or counting domains, the possibility of developing new components and the support of more platforms. These recent additions do not impact our evaluation. PAPI relies on the *hardware performance counters* that are available on most processors. It provides the connection between software and hardware performance counters.

In computers, hardware performance counters, are a set of special-purpose registers built into modern microprocessors to store the counts of hardware-related events within computer systems. The counters are used for online monitoring of hardware events. Compare to software profilers, hardware counters provide low-overhead access performance information related to cpu's functional units, caches etc. PAPI can extract information from many hardware counters. Depending on the brand of the processor, the number of available hardware counters does vary. For instance on a AMD Opteron processor, PAPI can access 4 counters at the same moment without activating multiplexing feature. Multiplexing is an option where the user can measure more counters with a small loss of accuracy (usually less than 1%).

Properties of the events are presented in Listing 1. First the name of each hardware counter is given along with its internal code. The `Avail` column declares if the event is available and the `Deriv` column shows if this PAPI counter uses two or more counters to produce a value or there is a hardware counter that provides this information directly. Finally, the last line presents how many counters are available for the used processor.

So by using this metric, it is possible to create traces with the exact amount of executed flops.

Profiling features With regard to the *profiling features* evaluation criterion, PAPI obtains a score of 4 points. Indeed, PAPI offers tracing facilities (2 points), that allows us to obtain information about the computation (1 point) and especially about volume for each computation event (1 point). However, PAPI does not provide any mechanism to trace communication events and there is no way to trace an application automatically using only PAPI. Nevertheless it is possible to instrument an application manually.

Listing 2 presents an example of how to measure the number of flops computed by a block of code. In order to trace a block of code, some commands should be inserted at the beginning and the end of it. If we want to measure the amount of executed flops for the function `ssor(itmax)` in the LU benchmark, the commands `PAPIF_start` and `PAPIF_stop` should be inserted as shown in Listing 2.

Listing 1: Events measured by PAPI on a AMD processor

```
The following correspond to fields in the PAPI_event_info_t structure.

    Name         Code     Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No    Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No    Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   No    Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No    Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No    Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No    Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes   Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No    Level 2 cache misses
PAPI_L3_TCM  0x80000008  No    No    Level 3 cache misses
...
PAPI_TOT_IIS 0x80000031  No    No    Instructions issued
PAPI_TOT_INS 0x80000032  Yes   No    Instructions completed
PAPI_INT_INS 0x80000033  No    No    Integer flops
PAPI_FP_INS  0x80000034  Yes   No    Floating point flops
PAPI_LD_INS  0x80000035  No    No    Load flops
PAPI_SR_INS  0x80000036  No    No    Store flops
PAPI_BR_INS  0x80000037  Yes   No    Branch flops
PAPI_VEC_INS 0x80000038  Yes   No    Vector/SIMD flops
PAPI_RES_STL 0x80000039  Yes   No    Cycles stalled on any resource
PAPI_FP_STAL 0x8000003a  No    No    Cycles the FP unit(s) are stalled
PAPI_TOT_CYC 0x8000003b  Yes   No    Total cycles
PAPI_LST_INS 0x8000003c  No    No    Load/store flops completed
PAPI_SYC_INS 0x8000003d  No    No    Synchronization flops completed
PAPI_L1_DCH  0x8000003e  Yes   Yes   Level 1 data cache hits
PAPI_L2_DCH  0x8000003f  Yes   Yes   Level 2 data cache hits
PAPI_L1_DCA  0x80000040  Yes   No    Level 1 data cache accesses
PAPI_L2_DCA  0x80000041  Yes   No    Level 2 data cache accesses
PAPI_L1_DCR  0x80000043  No    No    Level 1 data cache reads
PAPI_L2_DCR  0x80000044  No    No    Level 2 data cache reads
PAPI_L1_DCW  0x80000046  No    No    Level 1 data cache writes
PAPI_L2_DCW  0x80000047  No    No    Level 2 data cache writes
PAPI_L1_ICH  0x80000049  Yes   Yes   Level 1 instruction cache hits
PAPI_L2_ICH  0x8000004a  Yes   No    Level 2 instruction cache hits
PAPI_L3_ICH  0x8000004b  No    No    Level 3 instruction cache hits
PAPI_L1_ICA  0x8000004c  Yes   No    Level 1 instruction cache accesses
PAPI_L2_ICA  0x8000004d  Yes   No    Level 2 instruction cache accesses
PAPI_L3_ICA  0x8000004e  No    No    Level 3 instruction cache accesses
...
PAPI_FML_INS 0x80000061  Yes   No    Floating point multiply flops
PAPI_FAD_INS 0x80000062  Yes   No    Floating point add flops
PAPI_FDV_INS 0x80000063  No    No    Floating point divide flops
PAPI_FSQ_INS 0x80000064  No    No    Floating point square root flops
PAPI_FNV_INS 0x80000065  No    No    Floating point inverse flops
PAPI_FP_OPS  0x80000066  Yes   No    Floating point operations (Counts
speculative adds and multiplies. Variable and higher than theoretical.)

-------------------------------------------------------------------------
Of 103 possible events, 36 are available, of which 8 are derived.
```

Listing 2: Example of measuring the flops for the ssor iteration.

```
0       integer events (1), numevents, check, EventSet
1       integer Values (1)
2       character(15) ch,ch2
3       numevents=1
4       EventSet = PAPI_NULL
5       events(1) = PAPI_FP_OPS
6       check = PAPI_VER_CURRENT
7       call PAPIf_library_init(check)
8       call PAPIF_create_eventset(EventSet, check)
9       call PAPIF_add_event(EventSet, events(1),check)
10      call PAPIF_start(EventSet,check)
11      call ssor(itmax)
12      call PAPIF_stop(EventSet, Values,check)
13      write(ch,'(i10)') Values(1)
14      write(ch2,'(i6)') id
15      write(*,100) trim(ch2),' compute ',trim(ch)
16  100 format (A,A9,A)
```

Lines 0-1 declare the `events` array in which are saved the codes of the events to be measured. The variable `numevents` declares the number of events that PAPI should monitor and the variable `check` is used for checking the status of the commands. Furthermore the `EventSet` is the set with all events that are going to be used. The values from the counters are saved in the array `Values`. Line 2 declares two strings for printing the number of flops and the rank of the process. Line 3 declares the number of events that are going to be monitored. Line 4 sets `EventSet` to `PAPI_NULL` on the next line, the event `PAPI_FP_OPS` is saved to the events array. The variable `check` on the line 6 is always equal to the code of the current PAPI's version. Between the lines 7 and 12 the following commands are executed: the initialization of PAPI, the creation of the null set, the event is added to the null set, the measurement starts, the ssor function is called, and the measurement stops respectively. Finally from lines 13 to 16 the value of the counter is saved to the string `ch`, the id which is the rank of the process is saved to the string `ch2` and with the last two lines the number of the flops and the rank are printed with the format "rank compute value" which is the format of time-independent traces.

Moreover each process saves its own trace to a separate file so if the application is executed on N processes, then there are N trace files.

Quality of output About the *quality of output* evaluation criterion, PAPI obtains a score of 3 points. PAPI allows for the extraction of information on computations (1 point). By using appropriate commands such as those in Listing 2 it is possible to print the number of flops according to the time-independent trace format (1 point). As result there is no need for any conversion (1 point).

Example of traces for the LU benchmark

Table 1 presents the first four lines of the traces of the LU benchmark, class C, executed on 4 nodes. The benchmark was traced with PAPI, hence there is no info about the communication. After each MPI call we start measuring the executed flops and stop just before the next MPI call, where we save the corresponding action into our traces.

Table 1: The first lines of the time-independent traces of LU benchmark, class C, 4 nodes.

```
0 compute 10632621
0 compute 244390418
0 compute 11612863
0 compute 6492893
```

```
1 compute 10633358
1 compute 244372181
1 compute 9328093
1 compute 2284893
```

```
2 compute 10633400
2 compute 244372720
2 compute 9328049
2 compute 2284855
```

```
3 compute 10632459
3 compute 244394098
3 compute 9327954
3 compute 6492794
```

Space and time overhead Thanks to `papi_cost`, it is possible to compute the minimal, maximal, mean and the standard deviation of the execution time of basic PAPI operations, e.g., start/pairs and reads. The results are presented in Table 2.

	Min cycles	Max cycles	Mean cycles	std. deviation
Papi_start/stop	9	26381	12	26.73
Papi_read	111	28568	114	81.9
Addition	235	9537	254	10

Table 2: Overhead in cycles of basic PAPI operations as measured with `papi_cost`.

So the mean execution cost for the pairs `PAPI_start/stop` is 12 cycles and for `PAPI_read` it is 114 cycles.

We compare these values with the cycles that are needed for an addition and an assignment. A simple program was implemented for this comparison. The basic part of this program is presented in the Listing 3. With this program we count the total cycles of the cpu for computing the `c=c+1`. The hardware event `PAPI_TOT_CYC` is used and the addition is executed for 1,000,000 iterations such as the `papi_cost` program does.

Note: During the experiments we observed that the value of the cycles for the first loop is always big enough, such as in our example is 9537 cycles. All the next values are less than 300 cycles. So the cold start of the measurement causes big overhead. According to the values, the cost of `Papi_read` is not negligible compared to one addition but the benchmarks that are used compute a lot of flops compared to an addition and an assignment. For a cold start of measuring an addition the cost of calling these commands is big enough but not if they are called more times, i.e., if we start and stop PAPI measurement inside our code many times.

With regard to the space and time overheads, PAPI obtains the score of 6 points. In Table 3, we see that the time overhead is less than 50% for the LU benchmark class C (2 points). While slightly increasing, the time overhead remains in the 5-7% range, then we consider it as constant (2 points).

In Table 4 we can observe that when the number of the nodes is doubled then the size of the traces is almost two times bigger. We can thus claim that the increase is linear (1 point). Moreover with 32 nodes, the difference of the trace sizes from class A to C is of 25 MB per class, so the trace size increases linearly with the problem size, except for class D. Compared to the

Listing 3: Measuring the cycles of an addition

```
      count=0
      sd=0
      numevents=1
      EventSet = PAPI_NULL
      events = PAPI_TOT_CYC
      check = PAPI_VER_CURRENT

      call PAPIf_library_init(check)

      call PAPIF_create_eventset(EventSet, check)

      call PAPIF_add_event(EventSet, events, check)
      do 20 i = 1000000, 1, -1
        call PAPIF_start(EventSet, check)
        count=count+1
        call PAPIF_stop(EventSet, Values,check)
        val(i)=Values(1)
   20 continue
      max=MAXVAL(val)
      min=MINVAL(val)
      sm=SUM(val)
      print *, 'average', sm/count
      print *, 'max', max
      print *, 'min', min
      do 200 i = 1000000, 1, -1
        sd=sd+(val(i)-sm/count)*(val(i)-sm/count)
  200 continue
      avg=sd/1000000.0
      sd=SQRT(avg)
      print *,'sd', sd
      end
```

other classes, Class D works on larger matrices (14 times bigger than class C) and needs more iterations (300 vs. 250). The observed can then easily be justified and we give 1 point on this criterion.

Software quality Considering the *software quality*, PAPI obtains the score of 11 points. The installation of the PAPI is easy without any problem but there are some restrictions by the software dependencies (1 point). PAPI depends on linux performance counters driver (`perfctr`) [17] or performance monitoring interface (`perfmon2`) [18]. For the kernels older than 2.6.30 it is needed to patch the kernel, otherwise there is at least support for `perfmon2`. In the case that a patch is needed, the configuration of the kernel has to be changed. The monitoring counters should be enabled through the `make oldconfig` command and afterwards to build and install the modified kernel (1 point). Moreover PAPI is released under the New BSD license template which is a GPL-compatible software license (1 point). PAPI is the standard tool to access the hardware counters of the CPU and it is compatible with the major platforms as shown in Table 5 (1 point). Furthermore PAPI supports the C/C++ and Fortran programming languages (3 points). About the documentation, the installation process is well documented on the official

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)
4	685.18	696	1.58
8	345.2	362.5	5.01
16	184	195.4	6.2
32	100.1	106.6	6.5
64	53.08	56.8	7

Table 3: Time overhead of PAPI for the LU benchmark, Class C.

LU, class C	
Nodes	Size (in MB)
4	11
8	23
16	46
32	94
64	193

LU, 32 nodes	
Class	Size (in MB)
A	36
B	59
C	94
D	299

Table 4: Space overhead of PAPI for the LU benchmark.

web page (1 point), the usage of PAPI is well explained through the `PAPI references`, the `Presentations` and the `Doxygen docs` that are available under the `Documentation` menu of the official web page (1 point). Finally the project is active, patches are released often (1 point) and there is a support team answering any question (1 point).

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	Linux 2.2, 2.4,2.6	Mikael Pettersson's PerfCtr kernel patch for Linux (included)
AMD Opteron, Intel Pentium M, Core2 Series	Linux 2.6	Stephane Eranian's Perfmon2 kernel patch from sourceforge
Intel Pentium IV, D	Linux 2.2,2.4,2.6	Mikael Pettersson's PerfCtr 2.6.x kernel patch for Linux (included)
Intel Itanium II, Montecito, Montvale	Linux 2.4, 2.6	none

Table 5: Principal Hardware/OS combinations supported by PAPI.

Table 6 summarizes the results of the evaluation of PAPI.

Profiling features	Quality of output	Overheads	Quality of Software	**Total**
4	3	6	11	**24**

Table 6: Summary of the evaluation of PAPI.

4.2 PerfBench

PerfBench (PB) [5] is a set of programs that instruments a code, and then creates timing and performance files during run time. This tool is developed by the Instrumental company [19]. The evaluation was based on the version 5.2 which is the last one. Finally this tool can only profile an application.

Unfortunately the PerfBench is compatible only with PAPI version 2.X. It was possible to be installed but there were errors related to the instrumentation. PB can be used to get information for each subroutine about the time and the flops.

Profiling features With regard to the *profiling features* evaluation criterion, PB obtains the score of 2. This tool provides information on computation (1 point). Furthermore an application can be automatically instrumented (1 point) with the help of the scripts that PB provides.

Quality of Output PB could not profile an application without errors so we could not acquire any output files. As conclusion there is no ranking for the *quality of output* criterion (0 points).

Space and Time overheads For the same reasons as in the *quality of the output* criterion, there is no ranking (0 points).

Software quality Regarding the *software quality* criterion, PB obtains the score of 5. PB's installation is easy (1 point) but it depends on PAPI 2.X (0 points). After some modifications it is possible to install the PB tool but is not functioning as it should. PB is released under GPL license (1 point). The tool has the same hardware compatibility with PAPI 2.X as it can be seen in the Table 7. Only the programming languages Fortran and C are supported (2 points). Finally there are flops about the installation and the usage of the program (1 point).

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M	x86-Linux	PAPI 2.X
Intel Pentium M,	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II,	IA_64-Linux	

Table 7: Principal Hardware/OS combinations supported by PerfBench.

Table 8 presents the results of the evaluation of PerfBench.

Profiling features	Quality of output	Overheads	Quality of Software	**Total**
2	0	0	5	**7**

Table 8: Summary of the evaluation of PerfBench.

4.3 PerfSuite

PerfSuite [6, 20] is a collection of tools, utilities, and libraries for software performance analysis. This software is implemented by the University of Illinois/NCSA. The version 1.1.0 of PerfSuite was used in this evaluation while the last one is the 1.1.2 which fixes some bugs not related to our framework and it supports more processors. This version is the development version that supports the latest version of PAPI. Moreover PerfSuite is only a profiling software.

`Psrun` is a tool that can apply statistical profiling of hardware performance counters during the execution of an application. However, this tool provides the aggregation of the events so it is not possible to know their sequence. There is no need to compile again the application for profiling it, just execute `psrun` with the name of the application as argument. The `psrun` utility, can operate in two modes, the one is the counting mode where we can measure for example the flops of the whole application without having any information about the source code, so we do not know how many flops are computed per function. On the other side with profiling mode the tool provides the number of the flops per function.

The third utility, `psprocess`, is related to pre- and post-processing of performance measurements. Typically with `psprocess` it is possible to convert the XML results that psrun provides into HTML or even combine the results from many processors into one final file.

We present a simple example, profiling the LU, class C benchmark on four processes. Listings 4 and 6 respectively show the output of the counting and profiling modes.

The results of the counting mode are presented in the Listing 4, which shows that the application computes 703,804,547,988 floating point operations, this value is measured with the hardware counter `PAPI_FP_OPS`, the counting domain is the user (it measures only the events caused from the application). Finally it measures data for one counter, while the limit of the counter for multiplexing is four counters, thus the multiplexing is not activated , the value of Mflop/s is 1008,669 and the execution time is 697.756 seconds.

So from the counting mode is not possible to acquire any info relevant to the time-independent traces.

The Listing 5 presents the output of the profiling mode information. The version 3.7.0 of the PAPI tool is used with the hardware metric `PAPI_FP_OPS`. The period is the sampling rate of the profiling mode, afterwards is declared the number of the samples and it profiles only the user domain. In the continuation is presented the execution time is 716.84 seconds and the sampling mode is not activated.

From the output file in the Listing 6, we can see the percentage of the flops per function for the LU benchmark. These results provide aggregate information which is not compatible with the requirements of our framework. Unfortunately there are some question marks in the output because PerfSuite could not find the number of the lines where the functions are called. Afterwards, there are some sections in this listing with a column which declares the number of the samples, another one for the percentage of the samples that are caused due to the specific module, file or function, the total percentage of the samples till this point and the name of the module, file or function.

Listing 4: Example of profiling LU class C 4 nodes for counting mode part 3

```
Index Description                                              Counter
Value
===========================================================================
1 Floating point operations.............................    703804547988

Event Index
===========================================================================
1: PAPI_FP_OPS

Statistics
===========================================================================
Counting domain.............................................          user
Multiplexed.................................................            no
MFLOPS (wall clock).........................................      1008.669
Wall clock time (seconds)...................................       697.756
```

Listing 5: Example of Perfsuite output for a LU class C on 4 nodes part1

```
Profile Information
===========================================================================
Class                        : PAPI
Version                      : 3.7.0
Event                        : PAPI_FP_OPS (Floating point operations)
Period                       : 100000
Samples                      : 7039052
Domain                       : user
Run Time                     : 716.84 (seconds)
Min Self %                   : (all)
```

Listing 6: Example of Perfsuite output for a LU class C on 4 nodes part 2

```
Module Summary

  Samples    Self %   Total %   Module

   7039052   100.00%   100.00%   /tmp/lu.C.4

File Summary

  Samples    Self %   Total %   File

   7039052   100.00%   100.00%   ??

Function Summary

  Samples    Self %   Total %   Function

   1899668    26.99%    26.99%   rhs_
   1690194    24.01%    51.00%   jacld_
   1514948    21.52%    72.52%   jacu_
    917681    13.04%    85.56%   blts_
    917671    13.04%    98.60%   buts_
     45350     0.64%    99.24%   exact_
     38557     0.55%    99.79%   ssor_
     10962     0.16%    99.94%   erhs_
      3449     0.05%    99.99%   setiv_
       412     0.01%   100.00%   l2norm_
       152     0.00%   100.00%   error_
         6     0.00%   100.00%   pintgr_
         2     0.00%   100.00%   setbv_

Function:File:Line Summary

  Samples    Self %   Total %   Function:File:Line

   1899668    26.99%    26.99%   rhs_:??:0
   1690194    24.01%    51.00%   jacld_:??:0
   1514948    21.52%    72.52%   jacu_:??:0
    917681    13.04%    85.56%   blts_:??:0
    917671    13.04%    98.60%   buts_:??:0
     45350     0.64%    99.24%   exact_:??:0
     38557     0.55%    99.79%   ssor_:??:0
     10962     0.16%    99.94%   erhs_:??:0
      3449     0.05%    99.99%   setiv_:??:0
       412     0.01%   100.00%   l2norm_:??:0
       152     0.00%   100.00%   error_:??:0
         6     0.00%   100.00%   pintgr_:??:0
         2     0.00%   100.00%   setbv_:??:0
```

Profiling features Regarding the criterion of the *profiling features*, PerfSuite achieves the score of 2. This tool provides information on computation using the PAPI tool (1 point). It is possible to instrument an application automatically by executing the `psrun` utility (1 point).

Quality of output With PerfSuite is not possible to extract the required information because there is no tracing mode. As conclusion, the *quality of output* is marked with 0 points for all the sub-criteria.

Space and time overhead Moreover about the *space and time overhead* because Perfsuite has not the capability of tracing so the results are only for profiling, that's why we can not study the overhead (0 points). A small sample of the profiling overhead is presented in Table 9.

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)
4	685.18	717.24	4.68
8	345.2	363.197	5.21
16	184	191.349	3.99
32	100.1	106.606	6.5
64	53.08	55.3	4.18

Table 9: Time overhead of PerfSuite for the LU benchmark, Class C.

Software Quality Regarding the *software quality* criterion, PerfSuite obtains the score of 10 points. The installation of the program is very easy without any issue (1 point). Moreover it depends on the libraries `tDom` and `Tcl` but there is no installation issue (1 point). The tool is released under the University of Illinois/NCSA Open Source License. This license is based on MIT/X11 and BSD licenses and are GPL-compatible (1 point). Furthermore according to Table 10 this tool is compatible with the major platforms (1 point). It supports at least the profiling of the programming languages C/C++ and Fortran (3 points) and but not the profiling of the MPI communication (0 points). During the execution of an instrumented application there were some errors because of incompatibility issues with Mpich that's why we used OpenMpi. About the documentation, the installation and the usage processes are well documented (1 point), there are many information about how to link (if it is needed) or execute the program at their official web site. The project is still active (1 point) and there is a support team which answers to the emails of the users (1 point).

Table 11 shows the results of the evaluation of PerfSuite.

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux x86_64-Linux	PAPI
AMD Opteron, Intel Pentium M, Core2 Series		
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	PAPI

Table 10: Principal Hardware/OS combinations supported by PerfSuite.

Profiling features	Quality of output	Overheads	Quality of Software	**Total**
2	0	0	10	**12**

Table 11: Summary of the evaluation of PerfSuite.

4.4 MpiP

MpiP [21] is a lightweight profiling library for MPI applications. It is developed by The Regents of the University of California and the version 3.3 is evaluated, which is the last one.

Profiling features With regard to the *profiling features* evaluation criterion, MpiP obtains the score of 2. This tool provides information on communication (1 point). Moreover an application can be automatically instrumented (1 point) by just re-linking the application with the MpiP library.

Example of output, LU class C, 4 nodes: The output of the execution is presented in the Listings 7 and 8. In the first one, the callsite section identifies all the MPI callsites within the application. The first number, is the callsite ID. The next column contains the stack trace level of the execution, the third column is the name of the file or the address from which the MPI command was executed. Afterwards, there is a column with the line number of the file and in the continuation another one with the name of the function which calls the command. Finally the last one contains the name of the MPI command (w/o the MPI_ prefix).

In the Listing 8, are presented all the profiled MPI commands. In the first column, is the name of the command, in the second one, is the number of the site in which was executed. In the next column there is the rank which corresponds to this command, where in the case that there is the symbol $*$ instead of a number, is the aggregate line for this site and command. The column called count declares how many times this command is called and the rest columns contain what their header describes, the maximum, mean, minimum and the sum of the messages' sizes. Moreover it is a statistical profiling tool, thus it is possible for the output files not to include all the MPI calls. Finally in the case that an MPI call is executed from the same site many times, if the size of the message varies then the MpiP tool provides only the maximum, minimum, mean and sum values of the sizes without knowing exactly the size per message. So it is not possible to acquire the needed data for creating the time-independent traces.

Listing 7: Example of MpiP output for the LU benchmark class C on 4 nodes part 1

```
@--- Callsites: 83 ----------------------------------------------------------

 ID  Lev  File/Address        Line  Parent_Funct             MPI_Call
  1    0  bcast_inputs.f       28  bcast_inputs             Bcast
  1    1  read_input.f        122  read_input
  1    2  lu.f                 72  applu
  2    0  exchange_3.f        288  exchange_3               Wait
  2    1  erhs.f              243  erhs
  2    2  lu.f                112  applu
  .
  .
  .
 81    2  ssor.f               61  ssor
 82    0  exchange_5.f         76  exchange_5               Send
 82    1  pintgr.f            171  pintgr
 82    2  lu.f                138  applu
 83    0  exchange_6.f         76  exchange_6               Send
 83    1  pintgr.f            250  pintgr
 83    2  lu.f                138  applu
```

Listing 8: Example of MpiP output for the LU benchmark class C on 4 nodes part 2

```
@--- Callsite Message Sent statistics (all, sent bytes) ---------------------

Name                 Site  Rank  Count      Max     Mean      Min      Sum
Allreduce              17     0      1        8        8        8        8
Allreduce              17     1      1        8        8        8        8
Allreduce              17     2      1        8        8        8        8
Allreduce              17     3      1        8        8        8        8
Allreduce              17     *      4        8        8        8       32
.
.
.
Send                   82     3      1     1296     1296     1296     1296
Send                   82     *      1     1296     1296     1296     1296
Send                   83     3      1     1296     1296     1296     1296
Send                   83     *      1     1296     1296     1296     1296

@--- End of Report ---------------------------------------------------------
```

Quality of Output Mpip does not support the tracing mode, so it is not possible to acquire related info for the the time-independent traces. As conclusion the *quality of output* criterion is ranked with 0 points for all of its sub-criteria.

Space and Time Overhead As the MpiP tool provides only the profiling mode, it is not proper to study the time overhead of this tool (0 points). However, some results about the overhead of the LU benchmark, class C, are presented in the Table 12.

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)
4	685.18	693.12	1.16
8	345.2	356.54	3.28
16	184	190.21	3.37
32	100.1	104.20	4.09
64	53.08	55.53	4.61

Table 12: Time overhead of Mpip for the LU benchmark, Class C.

Table 12 shows that the overhead of MpiP is small enough. However, this is logic as the tool collects statistical information.

Software Quality Regarding the `software quality` criterion, MpiP obtains the score of 11. This tool can be installed without any issue (1 point) and it depends on the library libunwind which was installed successfully (1 point). This software is released under the New BSD License which is GPL-compatible (1 point). According to Table 13, MpiP is compatible with many hardware platforms (1 point) and the tool supports the programming languages C/C++ and Fortran (3 points). About the compatibility with MPI implementations, MpiP support both Mpich and OpenMpi (1 point). All the instructions about the installation and the usage of this tool are available in the website [21] (1 point). The project is active (1 point) and there is available a mailing list for the users of the tool (1 point).

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	binutils
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	libunwind, binutils
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	libunwind, binutils

Table 13: Principal Hardware/OS combinations supported by MpiP.

Table 14 summarizess the results of the evaluation of MpiP.

Profiling features	Quality of output	Overheads	Software Quality	**Total**
2	0	0	11	**13**

Table 14: Summary of the evaluation of MpiP.

4.5 MPE

MPE [8, 9, 22] (Multi-Processing Environment) is a software package for performance analysis of MPI programs. It is developed by the Argonne National Laboratory. We based our evaluation on the version 1.0.6p1, while the latest one is 1.3.0. The recent additions of the latest version, do not impact our evaluation. MPE can only trace MPI calls.

Currently, MPE offers the following three profiling libraries. The tracing library traces all the MPI calls. Each MPI call is preceded by a line that contains the rank in the global communnicator of the calling process and followed by another line indicating that the call is completed. The animation library provides a simple form of real-time program animation that requires X window routines. Finally with the logging library, it is possible to generate log files from user MPI programs. There are three different log file formats allowed in MPE. The `CLOG2` is the default one. It is basically a collection of events with timestamps. `ALOG` is an older format and it exists mainly for compatibility reasons, is not developed anymore. The Scalable log file format (`SLOG-2`) allows for much improved scalability for visualization purpose.

We use only the logging and tracing libraries as the animation libraries are out of the scope of our framework. Unfortunately, the default output of the tracing library, is the standard output. This increases the execution time but for all the experiments we redirect the output into a single file. The logging libraries contain enough data about the visualization of the MPI calls. With the tools `clog2TOslog2` it is possible to convert the `CLOG2` file to `SLOG-2` in order to provide scalable traces as input to a visualization tool. The tools `clog2print` and `slog2print` convert the respectively files into text format. For both formats it was not possible to provide the message size for the call `MPI_Bcast` by default.

Unfortunately, the default output of the tracing library is the standard output. This increases the execution time but for all the experiments we redirect the output into a single file.

The manual indicates the procedure for applying a selective instrumentation for the logging libraries but with many modifications into the code. In the case of the tracing libraries it is not mentioned a procedure and all the efforts did not provide the desired results. So it is not possible to acquire the traces for a specific block of code automatically.

Profiling features With regard to the *profiling features* evaluation criterion, MPE obtains the score of 4. This tool can trace an application (2 points) and can provide information about the communication (1 point). Moreover, an application can be automatically instrumented (1 point) by just re-linking the application with the MPE library.

Example of output, LU benchmark, class C, 4 nodes A part of the output of the execution by linking with the tracing library is presented in the Listing 9.

During the MPI_Send call, the variable count represents the number of elements in send buffer, the variable destination is the receiver of the message and the variable tag is the message tag.

Quality of output Regarding the *quality of output*, the MPE tool obtains the score of 1. This tool provides one pass conversion (1 point) but we can not extract all the required information

Listing 9: Output from the MPE tool for the LU benchmark class C on 4 nodes

```
[2] Starting MPI_Bcast
[2] Ending MPI_Bcast
[0] Starting MPI_Wait
[1] Ending MPI_Wait
[1] Starting MPI_Send with count = 131220, dest = 0, tag = 1
[0] Ending MPI_Wait
[1] Ending MPI_Send
[0] Starting MPI_Recv with count = 400, source = 2, tag = 3
[0] Ending MPI_Recv from 2 with tag 3
[3] Starting MPI_Send with count = 400, dest = 2, tag = 1
[3] Ending MPI_Send
[3] Starting MPI_Send with count = 400, dest = 1, tag = 3
[3] Ending MPI_Send
```

for our framework.

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)
4	685.18	755.35	10.24
8	345.2	393.8	14.08
16	184	225.5	22.55
32	100.1	144.3	44.15
64	53.08	251.1	373.06

Table 15: Time overhead of MPE for the LU benchmark, Class C.

Space and time overhead About the *space and time overhead* evaluation criterion, MPE obtains the score of 2. The traces' sizes are included in Table 16. In the case that the number of processes is doubled the traces' sizes are increasing linearly (1 point). With regard to the problem size, by changing only the class of the problem, the size of the traces is increased linearly (1 point).

LU, class C	
Nodes	Size (in mb)
4	58.9
8	147.3
16	358.32
32	786.67
64	1702

LU, 32 nodes	
Class	Size (in mb)
A	291.24
B	469
C	786.67
D	2300

Table 16: Space overhead of MPE for the LU benchmark.

Software quality MPE achieves a score of 10 points regarding the *software quality* criterion. The installation of the tool is very simple and no problems occured (1 point). There is no

dependency on another library (1 point). MPE is released under the same license with Mpich. According to this license, permission is hereby granted to use, reproduce, prepare derivative works and redistribute to others (1 point). Table 17 shows that MPE has no hardware compatibility issues (1 point). It supports C/C++ and Fortran programming languages (3 points). The MPE tool is included in the Mpich and all the experiments were executed with OpenMPI, so it supports both of them (1 point). The documentation of the tool contains all the necessary information (1 point). Moreover this project is active (1 point) there is a support team for answering any question (1 point).

Hardware	Operating System
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux
Intel Pentium IV, D	
Intel Itanium II, Montecito, Montvale	IA_64-Linux

Table 17: Principal Hardware/OS combinations supported by MPE.

Table 18 shows the overall results of the evaluation of MPE.

Profiling features	Quality of output	Overheads	Quality of Software	**Total**
4	1	2	10	**17**

Table 18: Summary of the evaluation of MPE.

4.6 IPM

Integrated Performance Monitoring (IPM) [10, 23] is a portable profiling infrastructure for parallel codes. It provides a low-overhead performance profile of the performance aspects and resource utilization in a parallel program. Communication, computation, and IO are the primary focus. The tool is is a collaborative project between National Energy Research Scientific Computing Center and San Diego Supercomputer Center. The evaluation was based on the IPM v0.982 while the last one is v.0.983 and provides only a profiling mode.

Profiling features With regard to the *profiling features* evaluation criterion, IPM obtains the score of 3. This tool provides information on communication (1 point) and computation by using PAPI tool (1 point). Finally it can automatically instrument an application (1 point) but for selective instrumentation is needed to add some commands in the code.

Example of output, LU class C, 4 nodes: In the Listing 10 is presented the output of a profiled application.

Listing 10: Example of IPM output for the LU benchmark class C on 4 nodes

```
##IPMv0.982 ############################################################################
#
# command : unknown (completed)
# host      : pastel-5/x86_64_Linux          mpi_tasks : 4 on 4 nodes
# start     : 07/01/10/15:48:54              wallclock : 694.390258 sec
# stop      : 07/01/10/16:00:29              %comm     : 3.56
# gbytes    : 0.00000e+00 total              gflop/sec : 4.05424e+00 total
#
############################################################################################
# region  : *       [ntasks] =       4
#
#                           [total]         <avg>           min           max
# entries                         4             1             1             1
# wallclock                2777.56        694.389       694.388       694.39
# user                     2676.78        669.194       668.294       670.826
# system                    101.03         25.259        23.6215       26.157
# mpi                        98.99         24.748        21.856        28.853
# %comm                                     3.564         3.14752       4.155
# gflop/sec                 4.05424       1.01356       1.01355       1.013
# gbytes                          0             0             0             0
#
# PAPI_FP_OPS          2.81523e+12   7.03806e+11   7.03798e+11   7.03811e+11
#
#                           [time]        [calls]       <%mpi>      <%wall>
# MPI_Recv                 39.4388         321280         39.84          1.42
# MPI_Wait                 32.5669           2040         32.90          1.17
# MPI_Send                 26.855          323320         27.13          0.97
# MPI_Allreduce             0.123987           40          0.13          0.00
# MPI_Irecv                 0.00638825       2040          0.01          0.00
# MPI_Bcast                 0.000819312        36          0.00          0.00
# MPI_Barrier               0.000624671         8          0.00          0.00
# MPI_Comm_rank             1.66488e-06         4          0.00          0.00
# MPI_Comm_size             1.47433e-06         5          0.00          0.00
############################################################################################
```

In the beginning there are some information about the root node, the number of the nodes in which the instrumented application was executed, the size of the files that were saved on the disk and the total Gflops. Afterwards there is a region for analyzing the times. For each category there are four columns; total, average, minimum and maximum. The *entries* describes how many times the tracing starts in the case that it stops during the execution. The *wallclock* is the total time, the *user* is the duration caused by the application, the *system* is the duration of tasks not related to the application, the *mpi* is the communication time. The *%comm* is the percentage of the communication time regarding the total time, the *gflop/sec* is the flop rate and the *gbytes* is the size of the files that were saved. In the continuation are presented for every MPI call that is included in the executed application, the total duration time, the number of the calls, the percentage compared to the total MPI time and the percentage compared to the total time.

Quality of Output IPM does not support the tracing mode, so it is not possible to acquire the time-independent traces. As conclusion the *quality of output* criterion can not be evaluated (0 points).

Space and Time overheads For the same reasons, the *space and time overheads* can not be evaluated (0 points). Although some results about the overhead of the LU benchmark, class C, are presented in Table 19.

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)
4	685.18	693.1	1.15
8	345.2	361.8	4.8
16	184	187.4	1.84
32	100.1	106.9	6.79
64	53.08	53.8	1.35

Table 19: Time overhead of IPM for the LU benchmark, Class C.

As it can be seen from Table 19, the overhead of IPM is at most 6.79%.

Software Quality Regarding the *software quality* criterion, IPM obtains the score of 11. The installation of the IPM tool is easy (1 point) and there is no dependency (1 point). This software is released under the LGPL license (1 point). Its only restriction about hardware compatibility is the PAPI tool, so according to Table 20 there is no issue (1 point) and the tool supports the programming languages C/C++ and Fortran (3 points). Furthermore it is compatible with both Mpich and OpenMpi (1 point). The documentation is available at the web site [23] with instructions about the installation and the usage of the tool (1 point). Finally the project is active (1 point) and there is a mailing list for asking questions about this tool (1 point).

Table 21 summarizes the results of the evaluation of IPM.

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 20: Principal Hardware/OS combinations supported by IPM.

Profiling features	Quality of output	Overheads	Quality of Software	**Total**
3	0	0	11	**14**

Table 21: Summary of the evaluation of IPM.

4.7 Extrae

Extrae [11] (formerly named MPItrace) is a dynamic instrumentation package to trace programs. This tool is developed by Barcelona Supercomputing Center. The evaluation was based on Extrae version 2.2.0 while the last one is version 2.3.2.

Profiling features With regard to the *profiling features* evaluation criterion, Extrae obtains the score of 7. It can trace an application (2 points) and provide information on communication (1 point) and computation (1 point). Moreover the communication and computation volumes are recorded in the trace files in bytes and flops respectively (2 points). Extrae can trace automatically an application (1 point).

Quality of output About the criterion *quality of output* the Extrae achieves the score of 2. The trace files can not be handled without the use of a tool which is provided by the Supercomputing Center of Barcelona. The trace files should be converted into a Paraver file by using either the serial Paraver merger called `mpi2prv` or the parallel one `mpimpi2prv`. Then with the help of a script it is possible to convert the Paraver file into time-independent traces (1 point) and it can be done by reading the file just one time (1 point).

Space and time overhead After the execution of an instrumented application, the tool creates intermediate trace files (*.mpit). These files are readable only from the Paraver or Dinemas mergers. The parallel Paraver merger `mpimpi2prv` was used for all the experiments. All the intermediate trace files are converted into one Paraver file. This procedure can take a lot of time because it demands to save the traces in a global file system. Afterwards a script should read this file in order to convert into time-independent traces. In the cases that this file is big, then a lot of time would be needed for the conversion.

With regard to the *space and time overhead* criterion, Extrae obtains the score of 5 points. The overhead for tracing is less than the 50% of the execution time (2 points), however a lot

of time is needed in order to convert the traces into Paraver files. According to Table 22, the overhead increases linearly as we increase the number of the processes (1 point). As it can be seen in Table 23 the traces' sizes increase linearly while the problem size is increased (1 point). Similar in the case which the number of the processes increases (1 point).

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)	Mpimpi2prv (in sec.)
4	685.18	696.3	1.62	28
8	345.2	358.34	3.8	43
16	184	194	5.43	61.7
32	100.1	106.89	6.78	130.1
64	53.08	60.15	13.1	290.7

Table 22: Time overhead of Extrae for the LU benchmark, Class C.

LU, class C	
Nodes	Size (in mb)
4	153
8	348
16	834
32	1800
64	3800

LU, 32 nodes	
Class	Sizes (in mb)
A	784
B	1300
C	1800
D	5400

Table 23: Space overhead of Extrae for the LU benchmark.

Software quality The score for the criterion *software quality* is 11 points. The installation of the tool is easy without any issue (1 point) and it depends on the libxml library (1 point). It is released under the LGPL license which was designed as a compromise between the strong-copyleft GNU General Public License or GPL and permissive licenses such as the BSD licenses and the MIT License (1 point). According to Table 24 there is not mentioned any issue about hardware compatibility (1 point). Moreover the programming languages C/C++ and Fortran are supported from the tool (3 points) and it is compatible with both Mpich and OpenMpi (1 point). A manual is provided with a lot of details about the installation and the usage of the tool (1 point). The project is active (1 point) and there is a support team for answering any question (1 point).

Table 25 presents the results of the evaluation of Extrae.

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI libxml
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 24: Principal Hardware/OS combinations supported by Extrae.

Profiling features	Quality of output	Overheads	Quality of Software	**Total**
7	2	5	11	**25**

Table 25: Summary of the evaluation of Extrae.

4.7.1 Scalasca

Scalasca [12, 24] is an open-source toolset that can profile and trace an application. It is developed by the Forschungszentrum Jülich, Jülich Supercomputing Centre and the German Research School for Simulation Sciences, Laboratory for Parallel Programming. We based our evaluation on version 1.3.3 while the latest is version 1.4.3. When the execution of an instrumented application ends, a parallel automatic event trace analysis tool called `SCOUT` is executed. This tool identifies and reports any logical clock violations. Since the version 1.3.3, a new environment variable is introduced that disables the execution of this tool as it does not influence the creation of the traces. The Program Database Toolkit (PDToolkit) ([25], [26]) is a framework for analyzing source code written in several programming languages and for making rich program knowledge accessible to developers of static and dynamic analysis tools. This framework is partially supported by Scalasca.

Profiling features Regarding the *profiling features*, Scalasca obtains a score of 6 points. This tool can trace an application (2 points), but it is not possible to trace the size of the message for a `MPI_Recv` call. Thus it can not extract from the trace files all the required data related to communication (0 point). All the required data about computation (1 point), the volumes in bytes (1 point) and the volumes in flops (1 point) are respectively recorded into the trace files. Finally, Scalasca can automatically trace an application (1 point).

Quality of output Scalasca provides the PEARL API [27] to create a C++/MPI tool to extract the required data from the traces (1 point). Thanks to the format of the trace files, it is possible to convert them in one pass (1 point).

Space and time overhead Considering the *space and time overhead*, Scalasca obtains a score of 8 points. According to Table 26, the time overhead stays under 50% (2 points). Moreover, we

see that it linearly decreases as the number of participating processes increases (3 points). As we execute the benchmark with one process per node, there is always one disk per process onto which flush the traces. The size of the trace produced by a single process decreases as the total number of processes grows, hence the reduction of the overhead.

problem than between

Nodes	LU benchmark, class C		
	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)
4	685.18	893	30.33
8	345.2	449	30.07
16	184	216	17.39
32	100.1	110	9.89
64	53.08	58.6	10.39

Table 26: Time overhead of Scalasca for the LU benchmark, Class C.

In terms of space overhead, the evolution of the trace size is correlated to that of the size of the instance (1 point). Class D is much bigger than the other instances (larger matrices and more iterations). This bigger gap is also observed in the size of the traces. From class A to class C, size grows by a factor of 2.3 and 2.6, while from class C to class D, the trace grows by a factor of 4.7. When the number of processes is increased for a given class, we observe a linear increase of the trace size (1 point). However, a large part of the trace size comes from the tracing of a function that does not depend on the number of processes and is not relevant for our studies. Scalasca allows the user to declare that some functions do not have to be traced, which allows for a reduction the space overhead (1 point).

LU, class C	
Nodes	Size (in MB)
4	2,100
8	2,200
16	2,500
32	2,900
64	3,900

LU, 32 nodes	
Class	Size (in MB)
A	476
B	1,100
C	2,900
D	36,000

Table 27: Space overhead of Scalasca for the LU benchmark.

Software quality Although some flags have to be set to install the tool, the installation was easy (1 point). It depends on `libbfd` and `libiberty` libraries but both libraries are part of standard distributions (1 point). Furthermore Scalasca is released under the New BSD license which is a GPL-compatible free software license and has been vetted as open source by the Open Source Initiative (1 point). According to Table 28 Scalasca is compatibile with a lot of hardware (1 point). Scalasca can trace applications which are implemented in C/C++ and Fortran (3 points) and is compatible with MPICH and OpenMPI (1 point). The documentation is well written and covers a lot of topics about the installation and usage of the tool (1 point). Finally the project is active (1 point) and there is a team providing support to the users (1 point).

Table 29 shows the overall results of the evaluation of Scalasca.

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI, PDToolkit, libbfd,libiberty,Qt4
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 28: Principal Hardware/OS combinations supported by Scalasca.

Profiling features	Quality of output	Overheads	Software Quality	**Total**
6	2	8	11	**27**

Table 29: Summary of the evaluation of Scalasca.

4.7.2 TAU

TAU [13, 28] is an advanced profiling and tracing toolkit for the performance analysis of parallel programs. It is developed by the Department of Computer and Information Science, University of Oregon, the Forschungszentrum Jülich, Jülich Supercomputing Centre, ZAM and the Advanced Computing Laboratory, Los Alamos National Laboratory. The evaluation is based on TAU version 2.21 while the last one is version 2.22.2 which suppors new technologies and processors and a new implementation of the MPI wrapper. According to our experiments these changes do not influence the current results.

Profiling features Regarding the *profiling features*, TAU obtains a score of 8 points. TAU can trace an application (2 points), and logs data related to the computation (1 point) and the communication (1 point) into trace files. Moreover the volumes of the communication and computation are in bytes (1 point) and flops (1 point) respectively. TAU supports the PDToolkit framework so it provides the user with a method to trace all the application (1 point) or only a block of code (1 point).

Quality of output The binary trace files produced by TAU contain all the required data for the simulation. It is possible to extract them with the help of the TAU trace format reader library [29] (1 point). We used the API to implement a tool that converts TAU traces into *time-independent* traces in one pass (1 point).

Space and time overhead The time overhead of TAU is given in Table 30. It is under 50% of the execution time of the application (1 point) but does not decrease as the number of processes grows. In terms of space overhead, the size of the traces increases linearly along with both the number of processes (1 point) and the problem size (1 point), as shown by Table 31. Finally, TAU provides a way to declare that some functions are not to be traced in order to decrease the time and space overheads (1 point).

Nodes	LU benchmark, class C		
	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)
4	685.18	983	43.46
8	345.2	493.5	42.96
16	184	239.3	30.05
32	100.1	128.8	28.67
64	53.08	71.1	33.94

Table 30: Time overhead of TAU for the LU benchmark, Class C.

LU, class C	
Nodes	Size (in MB)
4	5,100
8	5,400
16	6,000
32	7,300
64	9,900

LU, 32 nodes	
Class	Size (in MB)
A	1,300
B	2,700
C	7,300
D	87,200

Table 31: Space overhead of TAU for the LU benchmark.

Software quality The installation of TAU is easy. It can be configured thanks to compilation flags (1 point). It has a software dependency on the PDToolkit which can be respected (1 point). TAU is free under BSD style license which is GPL-compatible (1 point). According to Table 32 it supports various hardware configurations (1 point). Moreover all the programming languages C/C++ and Fortran (3 points) and the major MPI implementations such as MPICH and OpenMPI (1 point) are supported. The official web site gives access to manuals about usage and installation of this tool (1 point). Finally the project is active (1 point) and there is a team for answering any question (1 point).

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI, PDToolkit (optional)
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 32: Principal Hardware/OS combinations supported by TAU.

The overall results of the evaluation are presented in Table 33.

Profiling features	Quality of output	Overheads	Software Quality	**Total**
8	2	5	11	**26**

Table 33: Summary of the evaluation of TAU.

4.7.3 Score-P

Score-P [16, 30] has been developed by the German BMBF project SILC and the US DOE project PRIMA. It is a highly scalable tool suite for profiling, tracing, and the online analysis of HPC applications. The evaluated version is the 1.0-beta while the latest one is 1.1.1 that corrects some bugs not related with our usage. This tool is the result of the collaboration between the researchers involved in the development of many well known tools such as TAU, Scalasca and VampirTrace.

Profiling features With regard to *profiling features* criterion, Score-P achieves the score of 7 points. It can trace an application (2 points) and record all the computation (1 point) and communication events (1 point) with their volumes in flops by using PAPI (1 point) and bytes (1 point) respectively. It can automatically instrument an application (1 point).

Quality of output The Score-P tool achieves the score of 2 points for the criterion of *quality of output*. The format of the generated traces is the Open Trace Format Version 2 (OTF2) [31] which is a highly scalable, memory efficient event trace data format and will become the new standard trace format for TAU, Vampir and Scalasca. It is the common successor format for the Open Trace Format (OTF) and the Epilog trace format [32]. When the execution ends, the traces comprise the computation and the communication events with all the necessary data (1 point). From the OTF2 files it is possible to acquire the time-independent traces by using the OTF2 API and only one pass (1 point).

Space and time overhead Table 34 shows that the time overhead is less than 50% (2 points). Moreover the overhead decreases linearly as the number of the processes increases (3 points).

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %) (in %)
4	685.16	915.919	33.67
8	345.2	443.59	28.5
16	184	220.78	19.99
32	100.1	109.95	9.84
64	53.08	58.16	9.57

Table 34: Time overhead of Score-P for the LU benchmark, Class C.

According to Table 35 the increase of the number of the processes causes a linear increase of the size of the traces (1 point). Similarly, when the size of the problem increases, then the size of the traces increases also linearly (1 point). With the combination of PDT and the Score-P user API, code regions can be excluded from the instrumentation (1 point).

LU, class C	
Nodes	Size (in mb)
4	2800
8	2900
16	3200
32	3700
64	4900

LU, 32 nodes	
Class	Size (in mb)
A	583
B	1400
C	3700
D	48000

Table 35: Space overhead of Score-P for the LU benchmark.

Software quality Score-P achieves the score of 11 points regarding the *software quality* criterion. There is no issue about the installation (1 point) and no dependency on other libraries (1 point). It is available in Open Source under a BSD license (1 point). The tool is compatible with various hardware and is supported on many platforms (1 point) as it can be seen in Table 36.

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI, PDToolkit
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 36: Principal Hardware/OS combinations supported by Score-P.

It is compatible with the C/C++ and Fortran programming languages (3 points) and with the major MPI implementations (1 point). There is a manual which describes many different aspects of what this tool can do, how to install it and how to use it (1 point). This is a new project which aims at providing the best characteristics of some well known tools (1 point) and there is a support team for solving any problem (1 point). We aggregate the results of the evaluation in Table 37.

Profiling features	Quality of output	Overheads	Software Quality	**Total**
7	2	8	11	**28**

Table 37: Summary of the evaluation of Score-P.

4.7.4 Minimal Instrumentation (MinI)

There are two undesirable side effects caused by complex profiling tools. First, they may induce a large time overhead. Second, the measure of hardware counters may be skewed, as the

Listing 11: Handling MPI_Send function with MinI.

```
int  MPI_Send( buf, count, datatype, dest, tag, comm )
{
 ...
  if(PAPI_accum_counters(values, 1)!=PAPI_OK) {
        printf("PAPI does not support this metric\n");
  }
  counter2=values[0];

  MPI_Type_get_name(datatype,t_data,&np);
  this_type=encode_datatype(&t_data);
  PMPI_Comm_rank( MPI_COMM_WORLD, &llrank );


  sprintf(msg,"%d compute %lld\n",llrank,counters2-counter1);
  strcat(longmsg,msg);

  returnVal=PMPI_Send( buf, count, datatype, dest, tag, comm );

  if(this_type==0) sprintf(msg, "%d send %d %d\n",
                           llrank,dest,count);
  else sprintf(msg, "%d send %d %d %d\n",
                       llrank,dest,count,this_type);
  strcat(longmsg,msg);

  if(PAPI_accum_counters(values, 1)!=PAPI_OK) {
        printf("PAPI does not support this metric\n");
  }
  counter1=values[0];
  ...
}
```

flops caused by the tool itself are also measured. Unnecessary instrumentation, in our case instrumentation that generates trace data beyond the information strictly needed for our replay framework, would then unnecessarily increase overhead and skew. In order to control the instrumentation at a lower level, we implemented our own instrumentation method, called MinI. The MPI standard exposes two interfaces for each MPI function, one prefixed with `MPI_` and the other prefixed with `PMPI_`, the former calling the latter directly. This provides developers with the opportunity to insert their own code, e.g. for profiling purposes, in the implementation of all `MPI_` functions. This mechanism is used by several of the aforementioned tools, and we ourselves use it to insert code that is executed upon entry and exit for all MPI calls. This code retrieves hardware counters through PAPI, and generates event traces such as those in Figure 1. This approach is guaranteed to perform the minimal amount of instrumentation needed for our purpose.

According to both MPICH and OpenMPI implementations, the Fortran PMPI layer calls C PMPI either directly or through C MPI layer. Thus we just have to provide profiling wrappers in C to trace an application. Furthermore, to decrease the I/O overhead, we implement a simple buffering mechanism. We can declare how many events are saved in the memory before flushing them to the hard disk.

Listing 11 presents the instrumentation of the `MPI_Send` call, with this minimal instrumentation. The general concept is that when there is an `MPI_Send` call, we create a `compute` action with the amount of flops computed since the last MPI call. Then, `PMPI_Send` is called, and the corresponding `send` action is created, saved in the buffer and it starts measuring again the flops. We use the `PAPI_accum_counters` command which measures the corresponding PAPI metric since the beginning of the execution and increases monotonically.

Profiling features With regard to the *profiling features* evaluation criterion, the minimal instrumentation tool, obtains a score of 7 points. This tool can trace an application (2 points) and record the communication (1 point) and computation information (1 point) with the corresponding data in bytes (1 point) and flops (1 point). Moreover it can instrument automatically an application (1 point), but all the application has to be traced.

Quality of output Regarding the *quality of output*, it achieves a score of 3 points. This tool can record all the required information (1 point), it saves directly the measured data into time-independent traces (1 point), thus there is no need to read the traces at all (1 point).

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %) (in %)
4	685.18	695.8	1.55
8	345.2	360	4.29
16	184	191.63	4.15
32	100.1	105.2	5.09
64	53.08	55.8	5.12

Table 38: Time overhead of MinI for the LU benchmark, Class C.

Space and time overhead About the *space and time overheads* evaluation criterion, our minimal instrumentation obtains a score of 6 points. The time overhead in Table 38 is around 4-5% in all the cases, then smaller than 50% (2 points) and constant (2 points). The trace sizes are given in Table 39. When the number of processes is doubled, the size of the traces is increased linearly (1 point). With regard to the problem size, by changing only the class of the problem, the size of the traces is increased linearly (1 point).

LU, class C	
Nodes	Size (in MB)
4	20
8	49
16	120
32	257
64	549

LU, 32 nodes	
Class	Size (in MB)
A	99
B	161
C	257
D	804

Table 39: Space overhead of MinI for the LU benchmark.

Software quality MinI achieves a score of 10 points regarding the *software quality* criterion. There is no need to install the tool, just to compile it (1 point). There is no dependency except on PAPI (1 point). MinI is released under the LGPL license (1 point). Table 40 shows that MinI has no hardware compatibility issues (1 point). It supports C/C++ and Fortran programming languages (3 points). The MinI tool is developed according to the MPI standards, thus it supports both MPICH and OpenMPI (1 point). There are instructions on how to compile and use it (1 point). Moreover this project is active (1 point) there is a support team for answering any question (1 point).

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 40: Principal Hardware/OS combinations supported by MinI.

Finally we aggregate the results of the evaluation in the Table 41.

Profiling features	Quality of output	Overheads	Quality of Software	**Total**
7	3	6	10	**26**

Table 41: Summary of the evaluation of MinI.

4.8 VampirTrace

VampirTrace [14, 33] has been developed at ZIH, TU Dresden in collaboration with the KOJAK project from JSC/FZ Juelich. We use the version 5.13 while the last one is version 5.14.3 but the new features do not influence the current experiments. Although OpenMpi 1.3.3 includes VampirTrace 5.4.9, it is needed to use at least the version 5.8.0 and the reason is explained in this section. Moreover VampirTrace can trace an application and is possible to measure both volumes of computation and volumes of communication.

Profiling features With regards to the criterion of *profiling features*, VampirTrace achieves the score of 7. This tool can trace an application (2 points) and record all the events. Moreover it provides information for communication (1 point) and computation (1 point). For both communication and communication there is information about the volumes in bytes (1 point) and the volumes in flops (1 point) respectively. Finally an application can be traced automatically (1 point).

Quality of output About the criterion of *quality of output* the VampirTrace achieves the score of 2. This tool records all the events into Open Trace Format (OTF) [34] files which contain all the needed information for the time-independent traces (1 point). When the application terminates, the traces are unified with the use of the tool `vtunify` automatically. In the case that the traces are not saved in a global file system, then they are not unified and there is available only one `uctl` file per process instead of OTF files. This means that the files should be gathered into one node, be converted to OTF files and then to *time-independent* traces which is time consuming. The version 5.8, introduced the `vtunify-mpi` which is the parallel version of `vtunify`, that's why it is better to use at least this version which also requires that the files are saved into a global file system. In this case, the unification is implemented with an parallel algorithm. From the OTF files it is possible to acquire the time-independent traces by reading the traces once time (1 point).

Space and time overhead Furthermore the *space and time overheads* criterion is studied. At the Table 42, are presented the timings for the overhead which are less than 50% (2 points).

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %) (in %)
4	685.18	922.3	34.5
8	345.2	458.17	32.72
16	184	220.6	19.89
32	100.1	111.46	11.34
64	53.08	70.85	33.47

Table 42: Time overhead of VampirTrace for the LU benchmark, Class C.

In the continuation, at the left side of Table 43, we can see that increasing the number of processes, causes linear increase of the size of the traces (1 point). Furthermore at the right side of the Table 43, the size of the traces is increasing linearly while the size of the problem is increasing (1 point). VampirTrace offers the option to declare which functions should not be recorded in the traces in the case that the user is not interested about them or to declare how often a function should be traced (1 point).

LU, class C	
Nodes	Size (in mb)
4	3500
8	3700
16	4000
32	4600
64	6000

LU, 32 nodes	
Class	Size (in mb)
A	706
B	1700
C	4600
D	59700

Table 43: Space overhead of VampirTrace for the LU benchmark.

Software quality About the criterion *software quality*, VampirTrace achieves the score of 11. The tool is very easy to be installed either by installing OpenMpi or by installing it separately (1 point). This software does not depend on other tools or libraries (1 point) and it is released under the New BSD license which it is GPL-compatible. There is no issue about the hardware

compatibility, it supports the major platforms (1 point) as it can be seen in Table 44 and it is compatibile with the programming languages C/C++ and Fortran (3 points). Moreover the most well known MPI platforms are supported (1 point). The manual that is available at the formal website is well written and all the procedures about the installation and the usage of VampirTrace are included (1 point). Furthermore the project is active (1 point) and there is a support team for solving any problem (1 point).

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI, PDToolkit (optional)
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 44: Principal Hardware/OS combinations supported by VampirTrace.

Finally we aggregate the results of the evaluation in the Table 45.

Profiling features	Quality of output	Overheads	Quality of Software	**Total**
7	2	5	11	**25**

Table 45: Summary of the evaluation of VampirTrace.

5 Final Scores and Discussion

The final score for all the tools is presented in the Table 46.

Tool	Profiling features	Quality of output	Space and Time Overheads	Quality of Software	**Total**
PerfBench	2	0	0	5	**7**
PerfSuite	2	0	0	10	**12**
MpiP	2	0	0	11	**13**
IPM	3	0	0	11	**14**
MPE	4	1	2	10	**17**
PAPI	4	3	6	11	**24**
Extrae	7	2	5	11	**25**
VampirTrace	7	2	5	11	**25**
MinI	7	3	6	10	**26**
TAU	8	2	5	11	**26**
Scalasca	6	2	8	11	**27**
Score-P	7	2	8	11	**28**

Table 46: Summary of the evaluation of the studied instrumentation tool.

First, we recall the evaluation criteria we used are driven by the requirements of our framework. Then, it is not an exhaustive evaluation of all the features of each tool. About the score, PAPI achieves a high score. However, it only provides information about computation with manual instrumentation. Then, Extrae needs a lot of time to convert the trace files into Paraver files and depending on the application, traces can be large. As the conversion to time-independent traces should be done by a serial application, the demanded time could be significant with regard to the execution of an application. Reading the provided OTF files of VampirTrace instrumentation, requires to merge them, so a global file system is demanded, or at least to gather all the traces into a single node. This is a procedure that we want to avoid because the global file system is not supported by default by all our clusters and the gathering of the traces would consume a lot of time. Although Scalasca is an efficient tool, it does not comply with all the necessary requirements expressed by our framework. TAU, is one of the tools that provide all the features and it is efficient enough according to our requirements. However, there are some cases which are going to be described later, where it demands a lot of memory in comparison with other tools. One of the newest tools, named Score-P, seems to be promising. It provides all the required information and with the OTF2 API it is possible to extract *time-independent* traces without the need for a global file system. The MinI tool succeeded to most of our tests. It has been developed to fulfill our framework requirements. It provides the time-independent traces directly after the application's execution, just by compiling MPI applications against our tool with less overhead than any other tool. The TAU, Score-P, and MinI can be used for our framework but with some restrictions. For example if we want to create an action corresponding to communication such as `MPI_Alltoallv`, then we need to know some specific information about the MPI call that only MinI provides directly.

While MinI does not achieve the highest score, it is the most efficient with regard to our framework requirements. It leads to smaller time and space overhead than other tools. Furthermore it creates the *time-independent* traces directly during the execution of the application, thus it needs less space to save trace files on hard disks. The instrumentation skew is also smaller

than other tools, so the traces represent accurately the performance of an application. Another advantage is its lack of dependencies and it is easy to use by linking MinI to the desired application. For our prototype we used the TAU tool, as Score-P was not mature at that time. However, the need for an even more efficient profiling tool, led us to develop MinI.

References

[1] Frédéric Desprez, George S. Markomanolis, Martin Quinson, and Frédéric Suter. Assessing the Performance of MPI Applications Through Time-Independent Trace Replay. In *Proc. of the 2nd International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*, pages 467–476, Taipei, Taiwan, September 2011.

[2] Frederic Desprez, George S. Markomanolis, and Frédéric Suter. Improving the accuracy and efficiency of time-independent trace replay. In *SC Workshops*, 2012.

[3] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *Proceedings of the 10th IEEE International Conference on Computer Modeling and Simulation*, March 2008.

[4] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[5] PerfBench. `http://icl.cs.utk.edu/papi/custom/index.html?lid=51&slid=71`.

[6] Rick Kufrin. Perfsuite: An Accessible, Open Source Performance Analysis Environment for Linux. In *Proceedings of the 6th International Conference on Linux Clusters: The HPC Revolution 2005 (LCI-05)*, Chapel Hill, NC, April 2005.

[7] Jeffrey Vetter and Michael Mccracken. Statistical Scalability Analysis of Communication Operations in Distributed Applications. In *Proccedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, pages 123–132, Snowbird, UT, Jun 2001.

[8] Anthony Chan, William Gropp, and Ewing Lusk. *User's Guide for MPE: Extensions for MPI Programs.* Argonne National Laboratory, Mathematics and Computer Science Division, 1998.

[9] Anthony Chan, William Gropp, and Ewing Lusk. An Efficient Format for Nearly Constant-Time Access to Arbitrary Time Intervals in Large Trace Files. *Scientific Programming*, 16(2-3):155–165, 2008.

[10] Nicholas J. Wright, Shava Smallen, Catherine Mills Olschanowsky, Jim Hayes, and Allan Snavely. Measuring and understanding variation in benchmark performance. *HPCMP Users Group Conference*, 0:438–443, 2009.

[11] Extrae. `http://www.bsc.es/ssl/apps/performanceTools/`.

[12] Markus Geimer, Felix Wolf, Brian Wylie, and Bernd Mohr. A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications. *Parallel Computing*, 35(7):375–388, 2009.

[13] Sameer Shende and Allen Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[14] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias Müller, and Wolfgang Nagel. *The Vampir Performance Analysis Tool-Set.* Stuttgart, Germany, July 2008.

[15] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, Wolfgang E. Nagel, C. Bischof, M. Bücker, P. Gibbon, G. R. Joubert, B. Mohr, F. Peters (eds, Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing scalable applications with vampir,.

[16] Dieter an Mey, Scott Biersdorff, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P–A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany*, pages 1–12. Gauß-Allianz, Springer, June 2010. (to appear).

[17] Mikael Pettersson. Perfctr: the Linux Performance Monitoring Counters Driver. `http://user.it.uu.se/~mikpe/linux/perfctr/2.6/`.

[18] Perfmon2. `http://perfmon2.sourceforge.net/`.

[19] PerfBench Instrumental. `http://www.instrumental.com`.

[20] PerfSuite. `http://perfsuite.ncsa.uiuc.edu`.

[21] Scalable MPI Profiling MpiP: Lightweight. `http://mpip.sourceforge.net/`.

[22] MPE. `http://www.mcs.anl.gov/research/projects/perfvis/software/MPE/index.htm`.

[23] IPM: Integrated Performance Monitoring. `http://ipm-hpc.sourceforge.net`.

[24] Scalasca. `http://www.scalasca.org`.

[25] Performance Research Lab. University of oregon: Tau reference guide, chapter tau instrumentaton options, `http://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch02.html`.

[26] Performance Research Lab. University of oregon: Tau user guide, chapter selectively profiling an application, `http://www.cs.uoregon.edu/research/tau/docs/newguide/ch03s03.html`.

[27] Markus Geimer, Felix Wolf, Andreas Knüpfer, Bernd Mohr, and Brian J. N. Wylie. A parallel trace-data interface for scalable performance analysis. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 398–408, Berlin, Heidelberg, 2007. Springer-Verlag.

[28] Tuning and Analysis Utilities. `http://www.cs.uoregon.edu/research/tau/home.php`.

[29] Performance Research Lab. *TAU User Guide, chapter Tracing, TAU Trace Format Reader Library*. University of Oregon. `http://www.cs.uoregon.edu/research/tau/docs/newguide/ch06s02.html`.

[30] Score-P. `http://www.score-p.org`.

[31] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open trace format 2 - the next generation of scalable trace formats and support libraries. In *Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium*, 2011. (to appear).

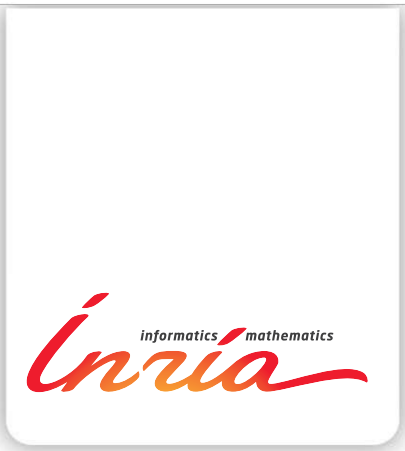
[32] F. Wolf and B. Mohr. *EPILOG binary trace-data format.* Technical report // Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich. FZJ-ZAM, 2004.

[33] VampirTrace. `http://www.tu-dresden.de/zih/vampirtrace/`.

[34] Allen Malony and Wolfgang Nagel. Open Trace - The Open Trace Format (OTF) and Open Tracing for HPC. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC'06)*, Tampa, FL, November 2006.

[35] The Top 500 List. `http://www.top500.org`.

[36] OTF. `http://www.tu-dresden.de/zih/otf`.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803