INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Time-Independent Trace Acquisition Framework A Grid'5000 How-to

George S. Markomanolis (INRIA, LIP, ENS Lyon, Lyon, France)
Frédéric Suter (IN2P3 Computing Center, CNRS, IN2P3, Lyon-Villeurbanne, France)

N° 0407

2011

Domaine 3

Rapport technique

ISSN 0249-0803 ISRN INRIA/RT--0407--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Time-Independent Trace Acquisition Framework A Grid'5000 How-to

George S. Markomanolis (INRIA, LIP, ENS Lyon, Lyon, France)
Frédéric Suter (IN2P3 Computing Center, CNRS, IN2P3,
Lyon-Villeurbanne, France)

Domaine : Réseaux, systèmes et services, calcul distribué
Équipe-Projet GRAAL

Rapport technique n° 0407 — 2011 — 26 pages

Abstract: This manual describes step-by-step how to create a Grid'5000 appliance that comprises all the tools needed to acquire time-independent traces of the execution of an MPI application. Time-independent traces are an original way to estimate the performance of parallel applications. It allows to totally decouple the acquisition of a trace from its replay in a simulation framework. This manual also details the different acquisition scenarios allowed by this approach. Traces can be acquired in a very classical way, by folding the execution on less resources, or by scattering the execution across multiple clusters.

Key-words: Grid'5000, Trace acquisition, appliance, off-line simulation.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Environnement d'acquisition de traces indépendantes du temps Manuel pour Grid'5000

Résumé : Ce manuel décrit pas à pas la création d'une image système pour Grid'5000 comprenant tous les outils nécessaires à l'acquisition de traces de l'exécution d'une application MPI qui sont indépendantes du temps. L'utilisation de telles traces est une approche originale pour estimer les performances d'applications parallèles. Cela permet de découpler entièrement l'acquisition d'une trace de son rejeu dans un environnement de simulation. Ce manuel décrit également les différents scénarios d'acquisition rendus possibles par cette approche. Les traces peuvent être obtenues de façon classique, en repliant l'exécution sur moins de ressources, ou encore en répartissant l'exécution sur plusieurs grappes de machines.

Mots-clés : Grid'5000, acquisition de traces, image système, simulation hors ligne

1 Introduction

Simulation is a popular approach to obtain objective performance indicators on platforms that are not at one's disposal. For instance, it may help the dimensioning of compute clusters in large computing centers. A framework for the off-line simulation of MPI applications was proposed in [4]. Its main originality with regard to the literature is to rely on time-independent execution traces.

In time-independent traces, each event occurring during the execution of an application, e.g., a CPU burst or a communication operation, is associated to the volume of the operation (in number of instructions or bytes) instead of the time spent to execute it. A time-independent trace can then been seen as a list of *actions*, e.g., computations and communications, performed by each process of an MPI application. An action is described by the *id* of the process that does this action, a *type*, e.g., a computation or a communication operation, a *volume*, i.e., a number of instructions or bytes, and some action specific parameters, e.g., the *id* of the receiving process for a one-way communication.

The NAS Parallel Benchmarks (NPB) suite, version 3.3, was used to illustrate our methodology. This suite comprises several benchmarks which can be either computation intensive, communication intensive, or both. Almost all the benchmarks can be executed for 7 different *classes*, denoting different problem sizes: S (the smallest), W, A, B, C, D, and E (the largest). For instance, a class D instance corresponds to approximately 20 times as much work and a data set almost 16 as large as a class C problem.

This manual describes how to build an appliance that comprises all the necessary software to acquire such time-independent traces. For the sake of simplicity we detail the building process of our appliance in the context of the Grid'5000 platform. The objective of the Grid'5000 project is to ensure the availability of a scientific instrument for experiment-driven research in the fields of large-scale parallel and distributed systems. This instrument was built as from 2003 under the initiative of the French ministry of Research under the ACI GRID program, and equipped with hardware comparable to that of a national scale grid (9 sites and over 5000 cores). It has since demonstrated that its fundamental concepts and tools to support experiment-driven research in parallel and distributed systems are solid enough attract a large number of users and to stay pertinent even though the focus of research in these areas has evolved in the past 7 years.

The remaining of this document is organized as follows. Section 2 describes the different steps of the building of an appliance, i.e., a full fledged system image. Section 3 presents the few configuration steps that have to be done in user space. Then Section 4 details the different time-independent trace acquisition processes.

2 Building the Appliance

2.1 Selecting a Base Image

Several base system images are made available by the Grid'5000 technical staff. All these images are based on the Debian Linux distribution and can be deployed on all Grid'5000 sites. There exist images based on three different versions of the Debian distribution, but one of them is aging (`etch`) while another one is unstable (`sid`).

Then we just detail the specifics of the images based on the Lenny version, which is the stable Debian release at the time of writing.

- **`lenny-x64-base`** is a minimal environment with no support of NFS and LDAP services. Only the drivers necessary to the support of the Grid'5000 infrastructure and high-speed network interconnect are installed.
- **`lenny-x64-nfs`** is based on `lenny-x64-base` and enables the LDAP and NFS services.
- **`lenny-x64-big`** is based on `lenny-x64-nfs` and includes various pre-installed packages such as `cmake`, `gfortran`, `OpenMPI`, `taktuk`, as well as several packages for development, system tools, and editors. All these tools provide a stable platform for configuring the operating system and execute various experiments.

In what follows we customize a `lenny-x64-big` base image to create an appliance adapted to our trace acquisition purposes.

2.2 Downloading Utility Software Packages

Downloading the sources of software packages that are available on the Internet from a node of Grid'5000 is prohibited for security reasons. Then some tools has to be downloaded on the user's workstation and then copied on Grid'5000 before starting to customize the appliance. We do not detail the purpose of each of the following software packages yet as it will be done later. The user has to download the PAPI, PDT, TAU, and NPB3.3 source packages as well as our own programs on his/her workstation. Note that the download of the NAS Parallel Benchmarks suite forces users to register first. This registration is free, and just aimed for download tracking. On `https://www.nas.nasa.gov/cgi-bin/software/start`, users can select "`NPB- NAS Parallel Benchmarks`" and "`Source Code NPB -3.3 602kb`" on the next page to download the archive. For the other tools, using `wget` is enough.

```
$ wget http://icl.cs.utk.edu/projects/papi/downloads/papi-4.1.1.tar.gz
$ wget http://tau.uoregon.edu/pdt.tgz
$ wget http://tau.uoregon.edu/tau.tgz
# wget http://graal.ens-lyon.fr/~gmarkoma/files/programs.tgz
```

The user can then copy the tools on one of the Grid'5000 frontend. Here we use that of the Sophia site.

```
$ scp papi-4.1.1.tar.gz username@access.sophia.grid5000.fr:
$ scp pdt.tgz username@access.sophia.grid5000.fr:
$ scp tau.tgz username@access.sophia.grid5000.fr:
$ scp NPB3.3.tar.gz username@acces.sophia.grid5000.fr:
$ scp programs.tgz username@acces.sophia.grid5000.fr:
```

2.3 Deploying The Base Image

The first mandatory step to customize a base image on Grid'5000 is to deploy it on one compute node in one of the Grid'5000 clusters. Here we illustrate the procedure using a node located in the Sophia site. The user first has to log his/herself on the access node of this site

```
$ ssh username@access.sophia.grid5000.fr
```

and then on the frontend of the site onto which it is possible to reserve compute nodes

```
$ ssh frontend
```

On Grid'5000, resource reservations are made through the OAR resource management system [2]. OAR distinguishes classical compute job from jobs that require to install their own system images. Such jobs are called *deployment* jobs and are submitted with the `-t deploy`. Moreover in the objective of customizing a base image, one requires to have a direct access to the deployed environment. The resource reservation is thus submitted in the *interactive* mode thanks to the `-I` flag. In the following example, one node is requested for three hours with the following OAR command

```
$ oarsub -I -t deploy -l nodes=1,walltime=3
```

Once the reversation request has been serviced by OAR, the user can get the list of his/her reserved nodes by doing

```
$ cat $OAR_FILE_NODES
```

In our example, the reserved node is `helios-7.sophia.grid5000.fr`. The next step is to actually deploy the base image on the reserved node. On Grid'5000 this is done thanks to the `kadeploy3` tool [5] with the following command

```
$ kadeploy3 -m helios-7.sophia.grid5000.fr -e lenny-x64-big \
  -k ~/.ssh/id_rsa.pub
```

where

- `-m` is either the list of the reserved nodes given by `$OAR_FILE_NODES` or directly the name of the reserved node;
- `-e` is the name of the image to deploy. This image has to be registered by `kadeploy3` before any deployment;
- `-k` allows the user to copy his/her ssh key on reserved node(s). This ssh key has be generated beforehand.

Deploying an image on a node of Grid'5000 gives to the user the possibility to connect to this node as a priviledged user. From now, we will assume that the operations are done as `root`, as denoted by the # prompt. To log on the reserved node as `root` the user has to execute the following command

```
$ ssh root@helios-7.sophia.grid5000.fr
```

The user will be asked for the same password that has to be known by Grid'5000 users for all the deployed base images.

2.4 Enabling the Access to Hardware Counters

The targeted time-independent traces use numbers of instructions as unit for computation actions. The appliance we are currently customizing thus has to enable the access to hardware performance counters. Hardware counters are a set of special-purpose registers built into modern microprocessors to store the counts of hardware-related activities within computer systems.

The Performance Application Programming Interface (PAPI) [1] is a consistent interface and methodology for use of the hardware counters. With Linux kernels prior to 2.6.31, using PAPI requires to patch the kernel with either `Perfctr` [7] or `Perfmon2` [6]. These are performance counters drivers that provide patches for the kernels to enable the access to the counters. Each patch applies for a specific kernel.

More recent kernels (above 2.6.31) include a built-in support for the `perf_counter` interface, also called PCL (Performance Counters for Linux). This interface was renamed to `perf_event` in kernel 2.6.32 and above.

While PAPI supports all of these interfaces, here we describe the procedure to patch and recompile the kernel of our appliance. Indeed the selected image is based on a 2.6.26 kernel.

Once the node is deployed, we copy the different tools from the previous section in the `/tmp` of this node.

We copy the sources from the frontend and extract them from the archive in the `/usr/local/src` directory

```
# scp username@frontend.sophia.grid5000.fr:papi-4.1.1.tar.gz /tmp
# tar xvfz /tmp/papi-4.1.1.tar.gz -C /usr/local/src
```

To apply the `perfctr` patch to the base kernel, it is also necessary to download the source files of the kernel and some Debian packages. For instance, the `build-essential` and `kernel-package` are respectively needed to build Debian and kernel image packages. The Debian source repositories are updated before download.

```
# apt-get update
# apt-get install linux-source-2.6.26 build-essential kernel-package
```

Then the kernel source files need to be installed

```
# cd /usr/src
# tar jxvf linux-source-2.6.26.tar.bz2
# ln -s linux-source-2.6.26 linux
# cd linux
# cp /boot/config-2.6.26-2-amd64 .config
```

On non-Grid'5000 configurations, or if the current image already comprises source files, the kernel source tree will need to be reset to a condition where the process of applying a patch can be done reliably.

```
# cp .config /tmp/config-bak
# make-kpkg clean
# make mrproper
# cp /tmp/config-bak .config
```

PAPI includes a set of `perfctr` patches in its distribution that corresponds to different kernel versions. It is recommended to test the patch first

```
# /usr/local/src/papi-4.1.1/src/perfctr-2.6.x/update-kernel --test \
  --patch=2.6.26
```

If the test did succeed, the patch can be safely installed.

```
# /usr/local/src/papi-4.1.1/src/perfctr-2.6.x/update-kernel \
  --patch=2.6.26
```

Once the kernel source has been patched, the kernel has to be configured to use the new `perfctr` code

```
# make oldconfig
```

The execution of `make oldconfig` will stop to ask for `perfctr` configuration.

```
[...]
Performance monitoring counters support (PERFCTR) [N/m/y/?] (NEW) m
```

Enter `"m"` to enable `perfctr` as a module in the new kernel. Then you will be prompted for additional `perfctr` configuration options. The most reasonable choices are

```
 Additional internal consistency checks (PERFCTR_DEBUG) [N/y/?] (NEW) n
 Init-time hardware tests (PERFCTR_INIT_TESTS) [N/y/?] (NEW) y
 Virtual performance counters support (PERFCTR_VIRTUAL) [N/y/?] (NEW) y
 Global performance counters support (PERFCTR_GLOBAL) [N/y/?] (NEW) y
```

The new kernel can now be rebuilt.

```
# CONCURRENCY_LEVEL=8 make-kpkg kernel-image --initrd \
  --append-to-version=-perfctr0
```

This command creates a Debian kernel image in `/usr/src`. The following commands install this new kernel

```
# cd /usr/src
# dpkg -i \
  linux-image-2.6.26-perfctr0_2.6.26-perfctr0-10.00.Custom_amd64.deb
```

At this point it is useful to note the names of the `vmlinuz` and `initrd.img` files which should be selected during the boot of the operating system (they are in the folder /boot). In our example, we have:

```
initrd.img-2.6.26-perfctr0
vmlinuz-2.6.26-perfctr0
```

To complete the installation of the `perfctr` driver, it is necessary to reboot on the new kernel. As we are building an appliance, it is possible to reboot the node within the current reservation. However, editing the grub configuration (`/boot/grub/grub.cfg`) is mandatory. The names of the kernel/initrd files has to be those produced by the kernel image installation as follows

```
set timeout=0
menuentry Linux {
        set root=hd0,3
        linux /boot/vmlinuz-2.6.26-perfctr0 root=/dev/sda3 ro
        initrd /boot/initrd.img-2.6.26-perfctr0
}
```

It is then possible to reboot the node.

```
# reboot
```

This reboot operation closes the connection with the reserved node and let the user on the frontend. He/She should wait for some time for the node to reboot (a couple of minutes). This time depends on the size of the image. Note that a mistake made in the grub configuration file, will prevent the node to boot correctly. In such a case, the user would have to restart from the base image again.

Once the reserved node is avalaible again, we can connect to it and check the kernel version as follows

```
$ ssh root@helios-7.sophia.grid5000.fr
# uname -a
```

If the kernel is the appropriate one, we can continue the installation of the `perfctr` driver.

Once the machine is rebooted with the patched kernel, the `perfctr` module has to be activated to enable the performance counters.

```
# modprobe -a perfctr
```

By tailing `/var/log/messages` it is possible to verify that the counters are succesfully loaded.

```
# dmesg | tail -18
 Please email the following PERFCTR INIT lines to mikpe@it.uu.se
 To remove this message, rebuild the driver with
   CONFIG_PERFCTR_INIT_TESTS=n
 PERFCTR INIT: vendor 0, family 6, model 26, stepping 5,
   clock 2260996 kHz
 PERFCTR INIT: NITER == 64
 PERFCTR INIT: loop overhead is 292 cycles
 PERFCTR INIT: rdtsc cost is 23.8 cycles (1816 total)
 PERFCTR INIT: rdpmc cost is 33.6 cycles (2448 total)
 PERFCTR INIT: rdmsr (counter) cost is 112.0 cycles (7460 total)
 PERFCTR INIT: rdmsr (evntsel) cost is 92.5 cycles (6216 total)
 PERFCTR INIT: wrmsr (counter) cost is 169.3 cycles (11132 total)
 PERFCTR INIT: wrmsr (evntsel) cost is 158.4 cycles (10432 total)
 PERFCTR INIT: read cr4 cost is 10.0 cycles (936 total)
 PERFCTR INIT: write cr4 cost is 119.1 cycles (7920 total)
 PERFCTR INIT: write LVTPC cost is 35.9 cycles (2592 total)
 PERFCTR INIT: sync_core cost is 195.8 cycles (12828 total)
 PERFCTR INIT: read fixed_ctr0 cost is 33.2 cycles (2420 total)
 PERFCTR INIT: wrmsr fixed_ctr_ctrl cost is 166.8 cycles (10972 total)
 perfctr: driver 2.6.41, cpu type Intel Nehalem at 2260996 kHz
```

The permissions on `/dev/perfctr` should also be changed to give users access to the performance counters. This is done by adding the following line to `/etc/udev/rules.d/91-permissions.rules`

```
KERNEL=="perfctr",MODE="0666"
```

anywhere, before the following line (if it exists):

```
LABEL="permissions_end"
```

This modification will be taken into account by executing the following commands that remove the module, reset the permissions, and reload the module

```
# rmmod perfctr
# /etc/init.d/udev restart
# modprobe -a perfctr
# ls -l /dev/perfctr
crw-rw-rw- 1 root root 10, 182 Jan 28 00:53 /dev/perfctr
```

Finally, to automatically load the `perfctr` module at boot time, `perfctr` has to be appended to `/etc/modules`.

```
# echo perfctr >> /etc/modules
```

Then we can configure, compile, test, and finally install PAPI in `/usr/local/papi`

```
# cd /usr/local/src/papi-4.1.1/src
# ./configure --prefix=/usr/local/papi
# make
# make fulltest
# make install-all
```

Note that this installation is only possible if the node has a kernel with an enabled `perfctr` driver. Moreover if most of the tests fail when executing `make fulltest`, this indicated issues with the activation of the kernel module. The correct loading of the module, and the permissions settings would then have to be checked.

2.5 Installing the Tracing Tools

While PAPI gives a direct access to hardware counters through the `perfctr` driver, we need to rely on a higher level tracing tool to acquire execution traces. Indeed traces do not only have to include information about computations, but also on communications, i.e., MPI operations.

Many tracing and profiling tools exist. To build this appliance we decided to install some components of the Tuning and Analysis Utilities (TAU) [8] suite. TAU is a well established tracing and profiling tool that supports automatic performance instrumentation at source level thanks to the Program Database Toolkit (PDT). This component then has to be installed first.

As for PAPI, we copy the archive from the frontend and extract the source of PDT in the `/usr/local/src` directory and install it.

```
# scp username@frontend.sophia.grid5000.fr:pdt.tgz /tmp
# tar zxvf /tmp/pdt.tgz -C /usr/local/src
# cd /usr/local/src/pdtoolkit-3.16
# ./configure -prefix=/usr/local/pdt
# make
# make install
```

We conclude this installation by adding the path to the directory in which PDT binary files are stored to the `$PATH` environment variable.

```
# echo "export PATH=$PATH:/usr/local/pdt/x86_64/bin" >> ~/.bashrc
# source ~/.bashrc
```

Here we use TAU with the OpenMPI runtime. This requires to install the OpenMPI development files to have access to the headers of the library and to define it as the default runtime.

```
# apt-get install libopenmpi-dev
# echo "/usr/lib/openmpi/lib" >> /etc/ld.so.conf.d/openmpi.conf
# ldconfig
# ln -sf /usr/bin/mpif77.openmpi /etc/alternatives/mpif77
# ln -sf /usr/bin/mpicc.openmpi /etc/alternatives/mpicc
# ln -sf /usr/bin/mpirun.openmpi /etc/alternatives/mpirun
```

Then we repeat the same procedure with the source archive of TAU. We configure TAU by giving it the paths to PDT, PAPI and MPI includes and binaries.

```
# scp username@frontend.sophia.grid5000.fr:tau.tgz /tmp
# tar zxvf /tmp/tau.tgz -C /usr/local/src
# cd /usr/local/src/tau-2.20.1
# ./configure -prefix=/usr/local/tau -pdt=/usr/local/pdt \
 -papi=/usr/local/papi \
 -mpiinc=/usr/lib/openmpi/include \
 -mpilib=/usr/lib/openmpi/lib
```

The call to `configure` should end by displaying the following message

```
Configuration complete!
   Please add  /usr/local/tau/x86_64/bin  to your path
   Type "make install" to begin compilation
```

Thus we add this path to the `$PATH` environment variable and execute the installation command

```
# echo "export PATH=$PATH:/usr/local/tau/x86_64/bin" >> ~/.bashrc
# source ~/.bashrc
# make install
```

2.6 Instrumenting an Application

As mentioned in the introduction of this manual, we use the NAS Parallel Benchmark (NPB) suite as an illustrative example. More information about the NPB suite

is available on `http://www.nas.nasa.gov/Resources/Software/npb.html`.

As in the previous sections, the archive is copied on the deployed node and the sources are extracted in the `/usr/local/src` directory of the deployed node.

```
# scp username@frontend.sophia.grid5000.fr:NPB3.3.tar.gz /tmp
# tar zxvf /tmp/NPB3.3.tar.gz -C /usr/local/src
```

The source code of the NPB comes in three different versions:

`/usr/local/src/NPB3.3/NPB3.3-SER` contains the *serial* version of the benchmark suite. These implementations are to be executed on a single processor are there for comparison purposes, in terms of numerical accuracy, with the following parallel implementations;

`/usr/local/src/NPB3.3/NPB3.3-MPI` contains the *MPI* version of the benchmark suite. This version relies on message passing through the *Message Passing Interface* to make the processes communicate.

`/usr/local/src/NPB3.3/NPB3.3-OMP` contains the *OpenMP* version of the benchmark suite. This version implements inter-process communication through shared memory and thus target SMP nodes.

We continue our illustration by focusing on the instrumentation of the MPI version of the NPB. Then we go in the `/usr/local/src/NPB3.3/NPB3.3-MPI` directory and list its contents

```
# cd /usr/local/src/NPB3.3/NPB3.3-MPI
# ls
bin  BT  CG  common  config  DT  EP  FT  IS  LU  Makefile  MG
MPI_dummy  README  README.install  SP  sys
```

We see that there is one directory for each benchmark (`BT`, `CG`, `DT`, `EP`, `FT`, `IS`, `LU`, `MG`, and `SP`). The `common` directory contains random number generators, timers, and printing functions used by all the benchmarks.

The `config` and `sys` directories contain files relative to the compilation of the benchmark suite that we will detail later. It is also possible to test the behavior of this MPI implementation without any installed MPI runtime thanks to the contents of the `MPI_dummy` directory.

Now, we detail how to instrument one given benchmark, the LU factorization, with the TAU profiling tool. TAU provides various options to instrument an application. The most basic way is to instrument the whole benchmark. In this case, it is just required to compile and link the source code with the appropriate scripts and libraries provided by TAU. For instance, to generate a instrumented binary of the entire LU benchmark, the user has to execute the following steps. First it is necessary to create a proper `make.def` file in the `config` directory. This file is loaded by the `Makefile` of each benchmark and defines the location of the FORTRAN and C compilers as well as the compiler flags and libraries to use. The NPB distribution comes with a template of the `make.def` that we copy and edit

```
# cp config/make.def.template config/make.def
```

In this file, we define `tau_f77.sh` as the program to use to compile MPI/FORTRAN codes

```
MPIF77 = tau_f77.sh
```

This script is a wrapper on the FORTRAN compiler provided by TAU which instruments and compiles a source code. This script also loads a TAU `Makefile` that contains more options and library links. Several files are provided by TAU and are located in `/usr/local/tau/x86_64/lib`. In our particular case, we need to load the file that links TAU with PAPI, MPI, and PDT, named `Makefile.tau-papi-mpi-pdt`. To enable the load of this file by `tau_f77.sh` and the use of PDT, we have to set some environment variables

```
# echo "export TAU_OPTIONS=-optPDTInst" >> ~/.bashrc
# echo "export TAU_MAKEFILE=\
/usr/local/tau/x86_64/lib/Makefile.tau-papi-mpi-pdt">> ~/.bashrc
# source ~/.bashrc
```

At this point, it is possible to compile whatever instance of the LU benchmark as it will be described later in this section.

One of the characteristics of the NPB codes, is to include a first phase in which the data are loaded and touched to warm up the system. The performance of this first phase may be different of the main computing part of the benchmark. Moreover, each parallel benchmark ends by gathering the produced results and checking their consistency. Then a user may want to instrument not the entire source code but only the main computing part of it. For instance, the main computing part of the LU benchmark is the `SSOR(itmax)` function call that occurs in the `lu.f` file.

In this case, which is the one chosen for our time-independent trace acquisition framework, it is required to resort to the *selective instrumentation* feature of TAU. This feature allows the user to enable or disable the tracing of the execution thanks to specific functions of the instrumentation API of TAU. As these functions go by pair, the basic scheme is to:

- Disable tracing at the beginning of the application;
- Enable tracing just before the interesting part of the source code;
- Disable tracing just after the traced source code;
- Enable tracing again just before the call to `MPI_Finalize()`.

To allow the user to compile either the instrumented or regular versions of a benchmark, these calls to the TAU instrumentation API are inserted in a copy of the main file. In the case of the LU benchmark, we thus copy `lu.f` to the new file `instr_lu.f`. This latter file is then edited to insert the following calls.

```
# cp LU/lu.f LU/instr_lu.f
```

The tracing is first disabled once all the variables have been declared. It corresponds to line 63 in the `lu.f` file

```
      integer ierr
      call TAU_DISABLE_INSTRUMENTATION()
```

Then the instrumentation is enabled just before the call to `ssor(itmax)`, which is the main computing part of the benchmark. This call occurs line 128. The instrumentation is disabled again right after the call to `ssor`

```
      call TAU_ENABLE_INSTRUMENTATION()
      call ssor(itmax)
      call TAU_DISABLE_INSTRUMENTATION()
```

Finally the tracing is enabled again before the call to `MPI_Finalize` on line 161.

```
      call TAU_ENABLE_INSTRUMENTATION()
      call mpi_finalize(ierr)
```

A similar procedure can be applied to the other FORTRAN benchmarks, while the C version of the instrumentation API has to be used for the C benchmarks.

The next step consists in defining a new compilation chain for the instrumented versions of the benchmarks. We aim at giving access to the compiled binaries from a standard location. Then we first create a new directory in `/usr/local/bin`.

```
# mkdir /usr/local/bin/NPB3.3
```

The main `Makefile` in `/usr/local/src/NPB3.3/NPB3.3-MPI` includes the `config/make.def` file that declares compiler locations, flags, and linked libraries. This `Makefile` also includes the `sys/make.common` file that defines the compilation targets. These two files have to be modified. We start with the `config/make.def` file that we create from the available template.

```
# cp config/make.def.template config/make.def
```

Then we edit this file to set the location of the MPI compilers for the regular version, add new entries for the instrumented version, and modify the destination directory for the binaries to `/usr/local/bin/NPB3.3`.

```
MPIF77 = mpif77                              # modification
INSTR_MPIF77 = tau_f77.sh                    # new
INSTR_FLINK   = $(INSTR_MPIF77)              # new
[...]
FMPI_LIB  = -L/usr/lib/openmpi/lib -lmpi
[...]
FMPI_INC = -I/usr/lib/openmpi/include
[...]
MPICC = mpicc                                # modification
INSTR_MPICC = tau_cc.sh                      # new
INSTR_CLINK = ${INSTR_MPICC}                 # new
[...]
MPI_LIB  = -L/usr/lib/openmpi/lib -lmpi          # modification
[...]
CMPI_INC = -I/usr/lib/openmpi/include        # modification
[...]
BINDIR  = /usr/local/bin/NPB3.3              # modification
```

To prevent memory issues that may occur for large instances such as class D, some flags also have to be added. The `-mcmodel=medium` flag allows code and data to be bigger than 2GiB on x64 operating systems. We thus add this flag for C and FORTRAN in the `make.def` file.

```
FFLAGS = -mcmodel=medium -0       # modification
[...]
CFLAGS = -mcmodel=medium -0       # modification
```

Now we edit the existing `sys/make.common` file to add the new compilation targets related to the instrumented versions. the following lines have to be added at the beginning of the file.

```
INSTR_PROGRAM = $(BINDIR)/instr_$(BENCHMARK).$(CLASS).$(NPROCS)
INSTR_FCOMPILE = $(INSTR_MPIF77) -c $(FMPI_INC) $(FFLAGS)
INSTR_CCOMPILE = $(INSTR_MPICC)  -c $(CMPI_INC) $(CFLAGS)
```

The next step requires to modify the `Makefile` of each benchmark. For example, `LU/Makefile` has to be modified as follows. First we declare the objects related to the new instrumented version target. With regard to the original `OBJS` rule, the only modification is to replace `lu.o` by `inst_lu.o`.

```
INSTR_OBJS = instr_lu.o init_comm.o read_input.o bcast_inputs.o \
       proc_grid.o neighbors.o nodedim.o subdomain.o setcoeff.o \
       sethyper.o setbv.o exact.o setiv.o erhs.o ssor.o exchange_1.o \
       exchange_3.o exchange_4.o  exchange_5.o exchange_6.o rhs.o \
       l2norm.o jacld.o blts$(VEC).o jacu.o buts$(VEC).o error.o \
       pintgr.o verify.o ${COMMON}/print_results.o ${COMMON}/timers.o

OBJS = lu.o init_comm.o read_input.o bcast_inputs.o proc_grid.o \
       neighbors.o nodedim.o subdomain.o setcoeff.o sethyper.o \
       setbv.o exact.o setiv.o erhs.o ssor.o exchange_1.o \
       exchange_3.o exchange_4.o exchange_5.o exchange_6.o rhs.o
       l2norm.o jacld.o blts$(VEC).o jacu.o buts$(VEC).o error.o \
       pintgr.o verify.o ${COMMON}/print_results.o ${COMMON}/timers.o
```

Then we declare a new rule called `inst_exec` that uses these instrumented objects.

```
exec: $(OBJS)
        ${FLINK} ${FLINKFLAGS} -o ${PROGRAM} ${OBJS} ${FMPI_LIB}
instr_exec: ${INSTR_OBJS}
        ${INSTR_FLINK} ${FLINKFLAGS} -o ${INSTR_PROGRAM} \
        ${INSTR_OBJS} ${FMPI_LIB}
```

To enable this new rule, we resort to a new the compilation parameter called `INSTR` whose default value is 0. When set to 1, this parameter will activate the compilation of the instrumented version. The `${PROGRAM}` rule also has to be modified as follows.

```
${PROGRAM}: config
        @if [ x$(VERSION) = xvec ] ; then \
                ${MAKE} VEC=_vec exec; \
        elif [ x$(VERSION) = xVEC ] ; then \
                ${MAKE} VEC=_vec exec; \
        elif [ $(INSTR) -eq 1 ] ; then \        # modified
                ${MAKE} instr_exec; \           # new
        else \                                  # new
                ${MAKE} exec; \
        fi
```

The `instr_exec` rule is applied if and only if the `INSTR` parameter is set to 1. Finally the creation of the objects should be modified as follows.

```
.f.o :
        @if [ ${INSTR} -eq 1 ] ; then \
```

```
                ${INSTR_FCOMPILE} $< ; \
        else \
                ${FCOMPILE} $< ; \
        fi

lu.o:           lu.f applu.incl npbparams.h
instr_lu.o:     instr_lu.f applu.incl npbparams.h
```

The final step is to modify the main `NPB3.3-MPI/Makefile` to initialize the default value of the `INSTR` parameter and take it into account in the LU compilation rule.

```
NPROCS=1
INSTR=0  #new
SUBTYPE=
[...]
LU: lu
lu: header
        cd LU; $(MAKE) INSTR=$(INSTR) NPROCS=$(NPROCS) CLASS=$(CLASS) \
        VERSION=$(VERSION)  # modification
```

A similar procedure has to be applied to each benchmark to have an instrumented version of the entire NPB benchmark suite.

2.7 Compiling the Instrumented Version

Once `Makefile` and configuration files have been modified as detailed in the previous section, it is possible to compile both regular and instrumented versions. The `Makefile` located in `/usr/local/src/NPB3.3/NPB3.3-MPI` allows for the compilation of each version thanks to different command line arguments through the following command.

```
# make <benchmark-name> NPROCS=<number> CLASS=<class> [INSTR=<value>]
```

```
<benchmark-name>  "bt" "cg" "dt" "ep" "ft" "is" "lu" "mg" "sp"
<value>           1 for the instrumented compilation
<number>          the number of processes
<class>           "S", "W", "A", "B", "C", "D", or "E"
```

Note that class E is not available for the DT and IS benchmarks.

For instance to compile the non-instrumented version of the class C of the LU benchmark to be executed by 64 processes we have to execute the following command in `/usr/local/src/NPB3.3/NPB3.3-MPI/`

```
# make LU NPROCS=64 CLASS=C
```

This will generate a binary, named `lu.C.64`, in `/usr/local/bin/NPB3.3`. The command to produce the instrumented version is very similar as it just requires to set the `INSTR` parameter to 1 on the command line.

```
# make LU NPROCS=64 CLASS=C INSTR=1
```

the generated binary is named `instr_lu.C.64`.

Compiling each possible combination of benchmark, class, number of processes, and type in both regular and instrumented version one by one is a tedious procedure. The NPB suite thus allows the user to compile all the desired instances in a simpler way. First the `config/suite.def` file has to be created from the available template and then edited.

```
# cd /usr/local/src/NPB3.3/NPB3.3-MPI/config
# cp suite.def.template suite.def
```

In this file, each line contains a benchmark name, a class, and number of processes. The possible value for the benchmark names and classes are those listed earlier. The different fields are separated by tabulations. For instance, the following lines in `suite.def` declare that the LU benchmark has to be compiled for classes A and B for 8 and 16 processes.

```
lu      A     8
lu      A     16
lu      B     8
lu      B     16
```

To support the generation of the instrumented version of the benchmarks with this method, the `sys/suite.awk` file has to be edited as follows to handle the `INSTR` parameter.

```
BEGIN { SMAKE = "make" } {
  if ($1 !~ /^#/ &&  NF > 2) {
    printf "cd `echo %s|tr '[a-z]' '[A-Z]'`; %s clean;", $1, SMAKE;
    printf "%s  CLASS=%s NPROCS=%s", SMAKE, $2, $3;
    printf "%s INSTR=1 CLASS=%s NPROCS=%s", SMAKE, $2, $3;     #new
```

Now we can compile all the instances listed in the `suite.def` file in one command.

```
# cd /usr/local/src/NPB3.3/NPB3.3-MPI/
# make suite
```

All the generated binaries are now in `/usr/local/bin/NPB3.3`.

2.8 Installing Tools for Trace Post-Processing

When the execution of an MPI application instrumented with TAU completes, many files are produced. They fall in two categories: *trace* files and *event* files. A *trace file* is a binary file that includes all the events that occur during the execution of the application for a given process. For each event (e.g., a function call or an instrumented block), this file indicates when this event starts and finishes. The time spent and the number of computed instructions between these begin/end tags are also stored. For MPI events all the parameters of the MPI call, including source, destination, and message size, are stored. To reduce the size of the trace files, TAU stores a unique id for each traced event instead of its complete signature. The matching between the ids and the descriptions of the functions can be found in the *event files*.

To perform an off-line simulation from these traces produced by TAU, two steps are mandatory. First we have to *extract* a time-independent trace from the trace and event

files produced by TAU. Second we have to *gather*, and sometimes *merge*, the extracted traces on a single node where the replay will take place.

As the trace files generated by TAU are binary files, there is a need for an interface to extract information. Such an API is provided by the TAU Trace Format Reader library (TFR). This tool provides the necessary functions to handle a trace file, including a function to read events. It also defines a set of eleven callback methods, that correspond to the different kinds of events that appear in a TAU trace file. For instance there are callbacks for entering or exiting a function and triggering a counter. The implementation of these callback methods is let to the developer.

We thus developed a C/MPI parallel application, called `trace_extract`, that implements the different callback methods of the TFR library. This program basically opens, in parallel, all the TAU trace files and read them line by line. For each event, the corresponding callback function is called. To gather the traces produced by `trace_extract`, developed a simple program that relies on a K-nomial tree reduction allowing for $\log_{(K+1)} N$ steps, where N is the total number of files, and K is the arity of the tree. This program can be configured to adapt the arity to the total number of traces and the number of compute nodes involved in the trace acquisition.

We get back this archive from the cluster frontend. We extract this archive directly in `/usr/local/src` and use a classical compilation chain to install the binaries in `/usr/local/bin`

```
# scp username@frontend:programs.tgz /tmp
# tar zxvf /tmp/programs.tgz -C /usr/local/src
# cd /usr/local/src/programs
# make
# make install
```

Now the image should be saved and declared again at the kadeploy software suite.

2.9 Saving and Recording the Appliance

Once all the necessary tools have been installed, the appliance can be saved and recorded by `kadeploy` for further usage. Tools are provided by the Grid'5000 technical staff to perform these operation. The `tgz-g5k` tool, which is installed in each base image, creates a tarball of the appliance on the frontend of the cluster. The following call, executed on the deployed node in `root` mode, will create a tarball named `newimage.tgz` in the home directory of the specified Grid'5000 user.

```
# tgz-g5k username@frontend:newimage.tgz
```

The next step consists in recording the appliance for future deployments. This has to be done by the user on the frontend of the cluster. First the user has to get the description of an existing environment from `kaenv3`, for instance that of the base image `lenny-x64-big`. Note that to list the names of all the registered appliances, a user can call `kaenv3 -l`.

```
$ kaenv3 -p lenny-x64-big -u deploy > ~/newimage.dsc
```

This file gives the name of the appliance, its version, its creator, the location of the tarball, the names of boot kernel and initial ramdisk, plus some other general characteristics. For instance the `newimage.dsc` includes the following information:

```
name : lenny-x64-big
version : 5
description : https://www.grid5000.fr/index.php/Lenny-x64-variants-2.3
author : support-staff@lists.grid5000.fr
tarball : /grid5000/images/lenny-x64-big-2.3.tgz|tgz
postinstall : /grid5000/postinstalls/debian-x64-big-2.3-post.tgz|tgz|
traitement.ash /rambin
kernel : /boot/vmlinuz-2.6.26-2-amd64
initrd : /boot/initrd.img-2.6.26-2-amd64
fdisktype : 83
filesystem : ext3
environment_kind : linux
visibility : public
demolishing_env : 0
```

Some fields have to be updated, name, author, tarball, kernel, initrd and visibility, according to the specifies of the customized appliance. Note that the visibility field should be declared as `shared` to allow other users to use this image. The modified description file is

```
name : ti_trace_acquisition-x64-2.6.26
version : 1
description : A Debian Lenny image with enabled access to hardware \
performance counters. Available tools: PAPI, TAU, NAS benchmarks, \
  trace_extract for extracting time-independent traces, \
  trace_gather for gathering traces into a single node.
author : user_email
tarball : /home/username/path_to_image/newimage.tgz|tgz
postinstall : /grid5000/postinstalls/debian-x64-big-2.3-post.tgz|tgz| \
  traitement.ash /rambin
kernel : /boot/vmlinuz-2.6.26-perfctr0
initrd : /boot/initrd.img-2.6.26-perfctr0
fdisktype : 83
filesystem : ext3
environment_kind : linux
visibility : shared
demolishing_env : 0
```

It is then possible to record the environment

```
$ kaenv3 -a newimage.dsc
```

If an already registered appliance has to be modified, the procedure is to:

1. Save the new tarball with `tgz-g5k`;
2. Update the version number in the description file;
3. Record again the environment with
   `kaenv3 -a newimage.dsc`

The old and new images can have the same name.

3 User Space Configuration

While the appliance built in the previous section can be deployed by any user of Grid'5000, most of the aforementioned configuration steps have been done with super user privileges. Some configuration has to be made in user space once the `ti_trace_acquisition-x64-2.6.26` appliance has been deployed.

Some environment variables related to TAU configuration have to be declared in the `~/.bashrc` file:

```
export TAU_MAKEFILE=/usr/local/tau/x86_64/lib/Makefile.tau-papi-mpi-pdt
export TAU_TRACE=1
export TAU_COMM_MATRIX=1
export TAU_OPTIONS="-optTauSelectFile=<path>/prof.tau \
-optPDTInst -optKeepFiles"
export TAU_METRICS=TIME:PAPI_TOT_INS
```

`TAU_MAKEFILE` points to the file which contain all the proper data for the compilation of the application. `TAU_TRACE` activates the tracing mode, while `TAU_COMM_MATRIX` generates detailed information about senders and receivers, and `TAU_CALLPATH` activates the tracing of each event callpath. The `optTauSelectFile` option points to a file which contains information relative to selective instrumentation. The `<path>` has to be set according to the correct location of the `prof.tau` file. The `optPDTInst` option is used to declare that the instrumentation is done by PDT, while the `optKeepFiles` option indicated to keep the intermediate `.pdb` and `.inst` files generated by PDT. Finally `TAU_METRICS` lists the counters that have to be traced. The first one is always the time and in our case the second one is `PAPI_TOT_INS` which counts the total number of instructions executed by the application.

A user also have to declare the path to the directory in which the benchmark binaries are located by editing the `~/.bashrc` file.

```
export PATH=/usr/local/bin/NPB3.3:$PATH
```

Once modified, the `/.bashrc` file has to be sourced again to take the modification into account.

```
$ source ~/.bashrc
```

In case a user would like to acquire a trace for an instance that is not already available in `/usr/local/bin/NPB3.3`, it is possible to compile and install extra instance as follows. For instance, to compile the instrumented version of the LU benchmark, class D for 128 processes the following commands should be executed

```
# cd /usr/local/src/NPB3.3/NPB3.3-MPI
# make LU NPROCS=128 CLASS=D INSTR=1
```

Note that this new binary will only be available on the node where the compilation was made. Then it has to be copied to every deployed node for immediate utilization. This can be done by using TakTuk [3].

```
$ taktuk -l root -f ~/node_file broadcast put \
{ /usr/local/bin/NPB3.3/instr_lu.D.128 } { /usr/local/bin/NPB3.3/ }
```

where `node_file` is a file listing all the machines the binary has to be copied to. Its life time will also be limited to that of the current deployment. To include this instance as part of the appliance, the procedure described in Section 2.9 has to be executed once again.

4 Acquiring a Time-Independent Trace

In this section we detail how to actually acquire an execution trace of an MPI application. We focus on the LU factorization benchmark for which we get traces with different acquisition modes. First we need to reserve some nodes of the Grid'5000 platform by submitting an allocation request to the OAR resource management system. Such a request is submitted from the frontend of a given Grid'5000 cluster in user mode. In the following example, we request 8 nodes of the `helios` cluster (located in Sophia) for 4 hours to deploy our custom image in an interactive mode.

```
username@fsophia:~$ oarsub -I -t deploy -l nodes=8,walltime=4 \
  -p "cluster='helios'"
```

Once the requested nodes have been allocated, the `$OAR_FILE_NODES` environment variable lists the names of all the obtained machines. As the `helios` cluster comprises 4 cores per nodes, each machine name appears 4 times. Now we can deploy our custom image on the nodes.

```
username@fsophia:~$ kadeploy3 -e ti_trace_acquisition-x64-2.6.26 \
  -f $OAR_FILE_NODES -k ~/.ssh/id_rsa.pub  -o ~/deployed_nodes
```

The output of this command is the list of the nodes for which the deployment did succeed. It is stored in the `deployed_nodes` file. It may happen that the deployment fails for some node. Then we have to check `deployed_nodes` against `$OAR_FILE_NODES` as follows.

```
username@fsophia:~$ for i in `cat ~/deployed_nodes`; \
do grep -v $i $OAR_FILE_NODES; done > ~/failed_nodes
```

If the `failed_nodes` file is not empty then we should deploy again the appliance on these nodes until all nodes are successfully deployed.

```
username@fsophia:~$ kadeploy3 -e ti_trace_acquisition-x64-2.6.26 \
-f ~/failed_nodes -k ~/.ssh/id_rsa.pub
```

As `$OAR_FILE_NODES` is available only on the frontend and not on the deployed nodes, we save its information in a separate file that will be used to run MPI programs. We also create a file in which each machine appears only once. This file will be used to broadcast commands to every node.

```
username@fsophia:~$ cat $OAR_FILE_NODES > node_file
username@fsophia:~$ uniq node_file > node_file_uniq
```

To execute an MPI application on an homogeneous network, each machine has to execute only one MPI process. In other words, only one core is used on each multi-core nodes. This is mandatory for timed traces analysis, and thus to compare timed and time-independent traces. To ensure that, it is possible to disable the other cores by setting `/sys/devices/system/cpu/cpu[1-9]*/online` to 0 on each node. We rely on TakTuk [3] to broadcast the appropriate command.

```
username@fsophia:~$ taktuk -l root -f node_file_uniq broadcast \
  exec [ 'for i in /sys/devices/system/cpu/cpu[1-9]*/online; do \
          echo 0 > $i ; done' ]
```

Now we can use one of the deployed nodes, e.g., the first one, to run the instrumented MPI application. As large files may be produced by the tracing tool, it is better to start the execution within a local directory than in the `/home` directory shared through NFS. Moreover this allows each node to write on its own disk and thus prevents concurrency issues. Every cluster of Grid'5000 does not dispose of an identified `/scratch` directory. Then we use the `/tmp` to temporarily store the acquired traces.

```
username@fsophia:~$ ssh `head -n 1 node_file_uniq`
helios-7.sophia.grid5000.fr:~$ cd /tmp
helios-7.sophia.grid5000.fr:~$ mpirun -machinefile ~/node_file_uniq \
  -np 8 instr_lu.C.8
```

Once the execution of the instrumented version has completed, we call our `trace_extract` program to transform the TAU traces into equivalent time-independent traces.

```
helios-7.sophia.grid5000.fr:~$ mpirun -machinefile ~/node_file_uniq \
  -np 8 trace_extract 8
```

The parameter given to `trace_extract` correspond to the number of trace files produced by TAU. Here, each node runs one process and thus produces one trace file, hence this parameter is equal to the number of nodes .

Then we can gather the time-independent trace on a single node thanks to the `trace_gather` program

```
helios-7.sophia.grid5000.fr:~$ mpirun -machinefile ~/node_file_uniq \
  -np 8 trace_gather -a 4 -m ~/node_file_uniq
```

- The flag *-a* declares the arity of the reduction tree (4);
- The flag *-m* declares the name of the machinefile which is used during the execution (`node_file_uniq`);

The last step consists in retrieving the traces from the `/tmp` directory to store them in a more permanent location. Depending on the size, it can be either in the `/home` directory of the user, for instance

```
helios-7.sophia.grid5000.fr:~$ mv ti_traces ~/ti_traces_lu_c_8
```

or on a machine outside Grid'5000. In this case, it is better to create a tarball of the traces to reduce the transmission time.

```
helios-7.sophia.grid5000.fr:~$ tar czf ti_traces_lu_c_8.tgz \
ti_traces
```

The security policy of Grid'5000 prevents user to go out of the platform from deployed nodes. To access to such a node from his/her workstation then has to configure `ssh` to go through the access node of the cluster. He/She also has to copy his/her public key in the `~/.ssh` directory on this cluster. The `~/.ssh/config` file then has to be edited to add the following entries.

```
Host access.*.grid5000.fr
        User username
        ProxyCommand nc -q 0 %h %p

Host *.grid5000.fr
        User username
        ProxyCommand ssh username@access.sophia.grid5000.fr \
        "nc -q 0 `basename %h .grid5000.fr` %p"
        ForwardAgent no
        StrictHostKeyChecking no
```

These declare two shortcuts called `access.*.grid5000.fr` and `*.grid5000.fr`. The first one is for the access nodes of the different sites while the second one is for the other nodes. The `ProxyCommand` line tells the system to connect first through `ssh` to the access server of the Sophia's site, and then open a `netcat` connection. Finally the `ForwardAgent` line declares that there will be no connection authentication agent forwarded to the remote machine. It is also necessary to modify the user configuration of `ssh` on each site to add the following entry.

```
StrictHostKeyChecking no
```

Once `ssh` is configured, the user can retrieve the tarball of traces with a classical `scp` command.

```
workstation:~$ scp helios-7.sophia.grid5000.fr:\
/tmp/ti_traces_lu_c_8.tgz .
```

As described in [4], there exist several ways to acquire a time-independent trace apart from running one process per node. Note that all the timed traces acquired by the methods described hereafter are skewed and cannot be used for comparison purposes. The first method is to use all the available cores in each node. This can be done either by skipping the step that disables all the extra cores or by enabling them again as follows.

```
username@fsophia:~$ taktuk -l root -f node_file_uniq broadcast \
  exec [ 'for i in /sys/devices/system/cpu/cpu[1-9]*/online; do \
          echo 1 > $i ; done' ]
```

With the same number of nodes, it is now possible to acquire a trace with 4 times as much processes. The three parameters has to be changed in the call to `mpirun` with regard to the previously described acquisition method. Indeed we can now use the original node file (in which each node appears four times), the number of processes grows up to 32, and we change the instrumented instance to run accordingly.

```
helios-7.sophia.grid5000.fr:~$ mpirun -machinefile ~/node_file \
  -np 32 instr_lu.C.32
```

Again, once the execution has completed we can call `trace_extract` to obtain the time-independent traces.

```
helios-7.sophia.grid5000.fr:~$ mpirun -machinefile ~/node_file \
  -np 32 trace_extract 32
```

The call to `trace_gather` is identical to the previous one as the same number of physical nodes is used.

Another way to acquire traces for larger instances with the same number of physical nodes is to *fold* the execution, i.e., execute several processes on each core. To enable such a folding, we have to replicate the nodes in the machine file according to the desired folding factor. For instance, if we want to execute 4 MPI processes on each core of the reserved nodes, and thus run the benchmark with 128 processes, we can execute the following command.

```
username@fsophia:~$ while read line ; do yes $line | head -n 4; done \
  < node_file > node_file_fold4
```

Then we execute the benchmark as follows

```
helios-7.sophia.grid5000.fr:~$ mpirun -machinefile ~/node_file_fold4 \
  -np 32 instr_lu.C.128
```

To extract the time-independent traces in such a configuration, we have to run `trace_extract` on the eight nodes listed in `node_file`, and tell the program that 128 processes were actually executed on these 32 cores.

```
helios-7.sophia.grid5000.fr:~$ mpirun -machinefile node_file \
  -np 32 trace_extract 128
```

To gather all the produced traces on a single node, `trace_gather` has again to be called exactly as in the regular mode with one process per node.

As our framework produces traces that are independent of time, it is possible to use more than one cluster to acquire them. In this case, called *scattering mode* in [4], the procedure is more complex as resources have to reserved in more than one site.

To reserve 16 nodes evenly scattered across the `helios` cluster (located in Sophia) and the `bordereau` cluster (located in Bordeaux) for 4 hours, we use the `oargridsub` submission interface.

```
username@fsophia:~$ oargridsub -t deploy bordereau:rdef="/nodes=8", \
helios:rdef="/nodes=8" -w "4:00:00"
```

If the reservation is successful, the last lines of the output of this command should be:

```
[OAR_GRIDSUB] Grid reservation id = 27585
[OAR_GRIDSUB] SSH KEY : /tmp/oargrid//oargrid_ssh_key_\
username_27585
You can use this key to connect directly to your OAR nodes with the \
oar user.
```

It is then possible to get extra information about the submitted job thanks to the `oargristat` command with the job number as parameter. In particular, this allows the user to obtain the local job ID for each cluster.

```
username@fsophia:~$ oargristat 27585

output:
clusters with job id:
 bordereau --> 404390 (name = "", resources = "/nodes=8", \
 properties = "", queue = default, environment = "", partition = "")
 helios --> 457948 (name = "", resources = "/nodes=8", \
 properties = "", queue = default, environment = "", partition = "")
```

With a resource reservation on multiple sites, it is not possible to retrieve the list of obtained nodes thanks to the `$OAR_FILE_NODES` environment variable. This list is local to each cluster. However, it possible to call `oargristat` with the `-l` and `-w` options to respectively list the nodes and ensures they are all up and running. As in the single cluster case, we make a second version of this file, with only one occurence of each node in it.

```
username@fsophia:~$ oargridstat -w -l 27585  > node_file
username@fsophia:~$ uniq node_file > ~/node_file_uniq
```

Then the procedure is very similar to the case of a single cluster. We have to deploy our custom image on the nodes (with the extra option `--multi-server` passed to `kadeploy3`), disable the extra cores, connect to a deployed node, run the benchmark, and extract the time-independent traces.

```
username@fsophia:~$ kadeploy3 -e ti_trace_acquisition-x64-2.6.26 \
-f ~/macq -k ~/.ssh/id_rsa.pub --multi-server
username@fsophia:~$ taktuk -l root -f node_file_uniq broadcast \
  exec [ 'for i in /sys/devices/system/cpu/cpu[1-9]*/online; do \
          echo 0 > $i ; done' ]
username@fsophia:~$ ssh `head -n 1 node_file_uniq`
helios-7:~$ cd /tmp
helios-7:~$ mpirun -machinefile ~/node_file -np 16 instr_lu.C.16
helios-7:~$ mpirun -machinefile ~/node_file_uniq -np 16 \
  trace_extract 16
```

To be able to use `trace_gather` across multiple sites, we have to copy the list of reserved nodes on each frontend first as it is a parameter of the program.

```
username@fsophia:~$ scp node_file_uniq  bordeaux:
username@fsophia:~$ mpirun -machinefile ~/node_file_uniq -np 16 \
  trace_gather -a 4 -m ~/node_file_uniq
```

The aforementioned procedure to copy the obtained traces on the user's workstation can then be applied.

The final acquisition mode combines the possibility to scatter nodes across multiple sites and to run several processes on a given core. Using the same multi-site reservation as before, we simply have to create a new machine file, run a larger instance, and extract the corresponding time-independent traces.

```
username@helios-7:~$ while read line ; do yes $line | head -n 4; done \
  < ~/node_file_uniq > ~/node_file_fold_4
username@helios-7:~$ mpirun -machinefile ~/node_file_fold_4 \
  -np 64 instr_lu.C.64
username@helios-7:~$ mpirun -machinefile ~/node_file_fold_4 \
  -np 16 trace_extract 64
```

Again the machine file has to be replicated on each site before gather the traces on a single node.

```
fsophia:~$ scp node_file_uniq bordeaux:
fsophia:~$ mpirun -machinefile ~/node_file_uniq -np 16 \
  trace_gather -a 4 -m ~/node_file_uniq
```

The user can finally copy the traces on his/her workstation as described earlier.

Acknowledgments

This work is partially supported by the ANR project USS SimGrid (08-ANR-SEGI-022) and the CNRS PICS N° 5473. Experiments presented in this report were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`).

References

[1] 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.

[2] Nicolas Capit, Georges Da Costa, Yannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A Batch Scheduler with High Level Components. In *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 776–783, Cardiff, UK, May 2005.

[3] Benoit Claudel, Guillaume Huard, and Olivier Richard. TakTuk, Adaptive Deployment of Remote Executions. In Dieter Kranzlmüller, Arndt Bode, Heinz-Gerd Hegering, Henri Casanova, and Michael Gerndt, editors, *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, pages 91–100, Garching, Germany, June 2009.

[4] Frédéric Desprez, George S. Markomanolis, Martin Quinson, and Frédéric Suter. Assessing the Performance of MPI Applications Through Time-Independent Trace Replay. Research Report RR-7489, Institut National de Recherche en Informatique et en Automatique (INRIA), December 2010.

[5] Emmanuel Jeanvoine, David Margery, Nicolas Niclausse, and Rémi Palancher. Kadeploy3. `https://gforge.inria.fr/projects/kadeploy3`.

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

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

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

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803