logo

JUBE

Benchmarking Environment

August 4, 2008 | Wolfgang Frings, Marc-André Hermanns, Stefanie Meier

Contents

1 JUBE Page Documentation

1.1 Preface

1.1.1 About this document

This document is still incomplete in the coverage of all features of JUBE. It will be updated in the future to complete each chapters and to provide more information on how to use the benchmarking environment for your purposes.

To contact the developers of JUBE please mail to: `jube-jsc@fz-juelich.de`.

1.1.2 Disclaimer

1.2 Introduction

Benchmarking a computer system usually involves numerous tasks, involving several runs of different applications. Configuring, compiling, and running a benchmark suite on several platforms with the accompanied tasks of result verification and analysis needs a lot of administrative work and produces a lot of data, which has to be analysed and collected in a central database. Without a benchmarking environment all these steps have to be performed by hand.

For each benchmark application the benchmark data is written out in a certain format that enables the benchmarker to deduct the desired information. This data can be parsed by automatic pre- and post-processing scripts that draw information, and store it more densely for manual interpretation.

The JUBE benchmarking environment provides a script based framework to easily create benchmark sets, run those sets on different computer systems and evaluate the results. It is actively developed by the Jülich Supercomputing Centre of Forschungszentrum J\"ulich, Germany.

This document addresses users of benchmarks suites created using the JUBE environment as well as developers of benchmark suites. Chapter Getting Started gives a brief introduction into the modifications needed to configure the benchmark environment on a specific platform as well as a short paragraph on execution and evaluation a benchmark run. The architecture of the JUBE environment is discussed thoroughly in chapter Configuration. Short tutorials on different topics concerning the work with the JUBE environment conclude this document.

1.3 Getting Started

1.3.1 Installation

The DEISA Benchmark Suite can be downloaded from the DEISA website:

`www.deisa.eu/science/benchmarking/`

By downloading any of the benchmarks from this site, you acknowledge that you have read, understood and agree to abide by the licence agreement of the DEISA Benchmark Suite as well as the licence agreements of individual packages, if applicable.

Unpack the DEISA Benchmark Suite into a directory of your choice. For licensing reasons, the source code of some applications of the benchmark suite cannot be provided directly with the DEISA Benchmark Suite itself. In these cases you have to obtain the application source from the relevant download site, as described in the README files of the individual packages, and copy the source code into the directory `src/` directory inside the application's directory. These packages are: DL_POLY, Fenfloss and CPMD.

After unpacking the Benchmark Suite the following directory structure is available:

- DEISA_BENCH/
  - applications/
  - bench/
  - doc/
  - platform/
  - skel/
  - LICENCE
  - README

The applications/ subdirectory contains the individual benchmark applications. The bench/ subdirectory contains the benchmark environment scripts. The doc/ subdirectory contains the overall documentation of the benchmark suite. The platform/ subdirectory holds the platform definitions as well as job submission script templates for each defined platform. The skel/ subdirectory contains templates for analysis patterns for text output of different measurement tools.

1.3.2 Configuration

1.3.2.1 The platform

Once you have obtained all sources from the applications you want to use in the benchmark, you can start configuring the Benchmark Suite for use on your platform.

A platform is defined through a set of variables in the platform.xml file, which can be found in the platform/ directory. To create a new platform entry, copy an existing platform description and modify it to fit your local setup. The variables defined here will be used by the individual applications in the later process. Best practice for the platform nomenclature would be: <vendor>-<system_type>-<system_name|site>. Additionally, you have to create a template batch submission script, which should be placed in a subdirectory of the platform/ directory of the same name as the platform itself. Although this nomenclature is not required by the benchmarking environment, it helps keeping track of you templates, and minimises the amount of adaptation necessary for the individual application configurations.

1.3.2.2 The applications

Once a platform is defined, each individual application that should be used in the benchmark needs to be configured for this platform. In order to configure an individual application, copy an existing configuration file (e.g. bench-IBM-SP5-Jump.xml) to the file bench-<your_-platform>.xml. Then open an editor of your choice, to adapt the platform file to your needs. Change the settings of the platform parameter to the name of your defined platform. The platform name can then be referenced throughout the benchmarking environment by the $platform variable.

1.3.3 Execution

Assuming the Benchmark Suite is installed in a directory that can be used during execution, a typical run of a benchmark application will contain two steps.

1. Compiling and submitting the benchmark to the system scheduler.
2. Verifying, analysing and reporting the performance data.

1.3.3.1 Compiling and submitting

If configured correctly, the application benchmark, e.g. SU3, can be compiled and submitted on the system (e.g. the Cray XT4 system Louhi at CSC) with the commands:

```
bash-2.05a$ cd DEISA_BENCH/applications/SU3
bash-2.05a$ perl ../../bench/jube bench-Cray-XT4-Louhi.xml
```

The benchmarking environment will then compile the binary for all node/task/thread combinations defined, if those parameters need to be compiled into the binary. It creates a so-called sandbox subdirectory for each job, ensuring conflict free operation of the individual applications at runtime. Iif any input files are needed, those are prepared automatically as defined.

Each active benchmark in the application's top-level configuration file will receive an ID, which is used as a reference by JUBE later on.

1.3.3.2 Verifying, analysing and reporting

After the benchmark jobs have run, an additional call to the benchmarking environment will gather the performance data. For this, the special parameters `-update` and `-result` are used.

```
bash-2.05a$ cd DEISA_BENCH/application/SU3
bash-2.05a$ perl ../../bench/jube -update -result <ID>
```

The ID is the reference number the benchmarking environment has assigned to this run. The performance data will then be output to stdout, and can be post-processed from there.

1.4 Architecture

This section is still under construction.

1.5 Configuration

1.5.1 Defining a platform

The JUBE system helps bringing a set of applications to a new platform with the introduction of several levels of configuration options. The highest of these levels is the platform itself. Here, all library paths and configuration switches for all applications in the benchmark a given for a specific platform. This also means, that all configuration parameters have to be held as general as possible to allow different applications to use application specific library and compilation parameters. The next listing shows an example platform definition for the IBM SP4 Regatta system JUMP at the Research Centre J\"ulich.

Listing of the platform.xml containing one platform:

```
<platforms>
    <platform name="IBM-SP4-Jump">
        <params
            make            = "gmake"
            rm              = "rm -f"
            ar              = "ar"
            arflags         = "-rs"
            ranlib          = "/usr/bin/ranlib"

            cpp             = "/usr/lib/cpp"
            cppflags        = "-P"

            f77             = "xlf_r"
            f77flags        = "-qtune=pwr4 -qarch=pwr4"

            f90             = "xlf90_r"
            f90flags        = "-qtune=pwr4 -qarch=pwr4"

            cc              = "xlc_r"
            cflags          = "-qtune=pwr4 -qarch=pwr4"

            cxx             = "xlC_r"
            cxxflags        = "-qtune=pwr4 -qarch=pwr4"

            mpi_f90         = "mpxlf90_r"
            mpi_f77         = "mpxlf_r"
            mpi_cc          = "mpcc_r"
            mpi_cxx         = "mpCC_r"

            ldflags         = "-qtune=pwr4 -qarch=pwr4"

            mpi_dir         = ""
            mpi_lib         = ""
            mpi_inc         = ""
            mpi_bin         = ""

            blas_dir        = ""
            blas_lib        = "-lessl"

            lapack_dir      = "-L/usr/local/lapack/LOCALlib"
            lapack_lib      = "-llapack -lessl"

            fftw3_dir       = "-L/usr/local/fftw/LOCALlib"
            fftw3_lib       = "-lfftw3 -lfftw3_threads -lm"
            fftw3_inc       = "-I/usr/local/fftw/LOCALinclude"

            fftw2_dir       = "-L/usr/local/fftw/LOCALlib"
            fftw2_lib       = "-ldfftw -ldrfftw -ldfftw_threads -ldrfftw_threads -ldfftw_mpi -ldrfftw_mpi -lm
            fftw2_inc       = "-I/usr/local/fftw/LOCALinclude"

            netcdf3_dir     = "-L/usr/local/netcdf/LOCALlib"
            netcdf3_lib     = "-lnetcdf"
            netcdf3_inc     = "-I/usr/local/netcdf/LOCALinclude"

            module_cmd      = "module load"
        />
    </platform>
</platforms>
```

The platform tag defines a new platform. Its name is used in different configuration steps later on. For each platform defined, a subdirectory to platform/ should hold any platform specific files, such as templates to batch queue submission scripts, etc..

The name of the platform will be used throughout the benchmark run for configuration purposes, thus a meaningful name can ease later maintenance of the benchmark suite.

1.5.2 Top-Level configuration of a benchmark

The top level configuration file defines the benchmark runs for a specific benchmark program. The outermost element <bench> has only two parameter, the benchmark name (here PEPC) and the platform on which the benchmark will be performed. Both values are used for building the unique identifier, and the platform name has to be one of the platforms defined in platform.xml. The bench-element can contain one or more <benchmark> elements describing a set of benchmark runs. This element has also an attribute for specifying a unique name. The second attribute 'active' can be used to manage different benchmark run sets. Only benchmark elements which are active will be processed by JUBE.

Listing of an example top-level configuration:

```
<!--
    DEISA BENCHMARK SUITE

    NBench benchmark configuration schemata for: PEPC

    Contact: w.frings@fz-juelich.de
-->
<!--
```

```
   Choose one of the platforms defined in the toplevel platform.xml

    IBM-SP4-Jump
    IBM-SP4-Zahir
    IBM-PowerPC-MareNostrum

    wscheme (walk_scheme) -> 0 (isend/irecv) or 1(coll)
    nt                    -> number of timesteps
    npart                 -> number of particles (ions)

-->
<bench name    = "PEPC" platform= "IBM-SP4-Jump" >

<benchmark name="scaling_1" active="1">
    <!-- version="reuse|new" -->
    <compile     cname="$platform" version="new" />
    <tasks       threadspertask="1" taskspernode="32" nodes="1"  />
    <params  npart="2000000"
                 nt="10"
         wscheme="0,1" />
    <prepare     cname="PEPC_sphere" />
    <execution   iteration="1" cname="$platform" />
    <verify      cname="PEPC" />
    <analyse     cname="$platform" />
</benchmark>

</bench>
```

The compile tag describes the compile steps. The cname attribute is a reference to an entry in compile.xml, one of the additional xml-files. The attribute version defines if the executable should be generated (new) or if an executable from a former benchmark run with the same configuration can be used (reuse).

The tasks tag is the definition of the number of cpus used for the benchmark run. Threadspertask is the number of OpenMP-Threads, taskspernode the number of MPI task per node and nodes the number of nodes. The total number of used processors is the product of these three values. In the first example, the attribute taskpernode contain a comma-seperated list of values. The benchmark run will be performed for each of this values.

The params tag defines the input parameters which should be used in the benchmark runs. The attributes of this element depends on the benchmark program and will be used in the preparation step for building the program input file. Each of this attribute van contain a comma separated list of value. JUBE will automatically generate benchmark runs for all possible combinations of these parameters and the tasks ranges given in the tasks element.

The prepare tag describes the preparation step. The corresponding element can be found in the file prepare.xml.

The execution tag contains a reference (cname) to a element in execute.xml. This file should contain for each batch system one entry which can be referenced here. The attribute iteration defines the number of repetitions of a benchmark run. This can be used for measuring random runtime variations between different runs.

The postprocess tag is currently not used. This step could be used for collecting data, e.g. gathering data from output files of each task.

The analyse tag contains a reference (cname) to a element in analyse.xml. In the analyse step the output file of the benchmark run will be scanned in respect to the patterns defined in analyse.xml.

1.5.3 The compile step

This file defines all relevant parameters needed for the configuration and compilation of the source code. This file can contain, like the other second-level XML files, one or more different elements. The attribute cname must be unique in the file and defines the name under which it can be selected in the top-level XML file.

Example compile step definition:

```
<compilation>

<!-- predefined vars:
   $outdir -> output directory for temporary compile files
   $id     -> identifier of this benchmark run
-->

<compile cname="IBM-SP4-Jump">
    <!-- Specification of source files to copy into temporary build
         directory -->
    <src directory="./src" files="configure.in configure config.h.in makefile.in lpepcsrc pepc-b src-config c

    <!-- Create Makefile and substitute parameters -->
    <substitute infile="compile.sh.in" outfile="compile.sh">
        <sub from="#EXECNAME#"      to="$execname" />
        <sub from="#OUTDIR#"        to="$outdir" />
        <sub from="#CPP#"           to="$cpp" />
        <sub from="#CPPFLAGS#"      to="" />
        <sub from="#CC#"            to="gcc" />
        <sub from="#CFLAGS#"        to="" />
        <sub from="#FFLAGS#"        to="-q64 $f90flags" />
        <sub from="#MPI_F90#"       to="$mpi_f90" />
        <sub from="#LDFLAGS#"       to="$ldflags" />
        <sub from="#CONFIG_SHELL#"  to="/usr/bin/bash" />
        <sub from="#MPI_DIR#"       to="" />
        <sub from="#MPI_LIBS#"      to="" />

        <sub from="#MAKE#"          to="$make" />
        <sub from="#RM#"            to="$rm" />
        <sub from="#AR#"            to="$ar" />
        <sub from="#ARFLAGS#"       to="-X64 $arflags" />
        <sub from="#RANLIB#"        to="$ranlib" />
        <sub from="#F77#"           to="$f77" />
        <sub from="#F90#"           to="$f90" />
        <sub from="#F90FLAGS#"      to="-q64 $f90flags -O3 -qsuffix=cpp=F90" />
        <sub from="#CXX#"           to="$cxx" />
        <sub from="#CXXFLAGS#"      to="-q64 $cxxflags" />
        <sub from="#MPI_F77#"       to="$mpi_f77" />
        <sub from="#MPI_CC#"        to="$mpi_cc" />
        <sub from="#MPI_CXX#"       to="$mpi_cxx" />
        <sub from="#LD#"            to="$mpi_f90" />
        <sub from="#MPI_DIR#"       to="$mpi_dir" />
        <sub from="#MPI_LIB#"       to="$mpi_lib" />
        <sub from="#MPI_INC#"       to="$mpi_inc" />
        <sub from="#MPI_BIN#"       to="$mpi_bin" />
        <sub from="#BLAS_DIR#"      to="$blas_dir" />
        <sub from="#BLAS_LIB#"      to="$blas_lib" />
        <sub from="#LAPACK_DIR#"    to="$lapack_dir" />
        <sub from="#LAPACK_LIB#"    to="$lapack_lib" />
        <sub from="#FFTW_DIR#"      to="$fftw3_dir" />
        <sub from="#FFTW_LIB#"      to="$fftw3_lib" />
        <sub from="#FFTW_INC#"      to="$fftw3_inc" />
        <sub from="#NETCDF_DIR#"    to="$netcdf3_dir" />
        <sub from="#NETCDF_LIB#"    to="$netcdf3_lib" />
        <sub from="#NETCDF_INC#"    to="$netcdf3_inc" />
        <sub from="#MODULE_CMD#"    to="$module_cmd" />
        <sub from="#MODULE_FILES#"  to="" />

<!-- "-qautodbl=dbl4" flag overwrites "-qrealsize=8" for "mpi_times.f90" module file. -->

    </substitute>

    <!-- issue build command -->
    <command>sh compile.sh</command>
</compile>

</compilation>
```

The param tag defines parameter for the compilation. Typical parameters are the compiler flags, the compiler name and loader name. The values specified in this element will also be part of the XML file containing the results of a benchmark program.

The src tag describes the source needed for the compilation. All files specified by the two attributes directory and files are copied to the temporary working directory of the benchmark run. The src element can specified more than once for copying files from different directories.

The substitute tag describes the replacement of the parameters in a template file. Each substitute can handle the replacement in one file. This element can also be iterated. The template has also to be copied by the src element. The placeholders are defined in the from attribute of the sub element. It will be replaced by the value of the to attribute. Possible variable names which can be used here are the attribute names defines in the param element or the attributes of the elements in the top-level XML file. There are also some pre-defined variables like $execname for the path and name of the executable.

The command tag defines the command which should be used to start the compilation. Typically this will be a call of make or gmake. It is also possible to start a shell script or to run configure in this place.

1.5.4 The prepare step

The prepare control file defines how the input file of benchmark should be generated. In principal this is defined by a substitution of placeholders in a template file.

Example prepare step definition:

```
<preparation>

<prepare cname="PEPC_sphere">
  <mkdir directory="data" />
  <mkdir directory="dumps" />
  <mkdir directory="log" />
  <mkdir directory="fields" />
  <mkdir directory="fields_pp" />

  <input files="input/sphere.h.in input/sphere_start.h.in" />
  <substitute infile="sphere.h.in" outfile="run_bench.h">
    <sub from="#NPART#"   to="$npart" />
    <sub from="#WSCHEME#" to="$wscheme" />
    <sub from="#NT#"      to="$nt" />
  </substitute>

  <substitute infile="sphere_start.h.in" outfile="run_start.h">
    <sub from="#NPART#"   to="$npart" />
    <sub from="#WSCHEME#" to="$wscheme" />
    <sub from="#NT#"      to="$nt" />
  </substitute>

  <command>
  ( cd $rundir;
    perl $benchhome/run/create_pelist.pl $ncpus;
    $benchhome/run/make_pes;
    $benchhome/run/clean_pes;
  )
  </command>

</prepare>

</preparation>
```

The input tag describes the location of template files.

The substitute tag defines the replacements, further description see compile step.

The command tag specifies a optional command which should executed after the replacements. In the example above this is a script which creates temporary subdirectories for each processor.

1.5.5 The execution step

This file contains the description how a the parallel program will be executed. Typically this is done by a batch job which will be submitted to the local batch system. It can also be a interactive run of the benchmark program. In this case, the program will be executed directly from the bench.pl script. Otherwise the job will be submitted and the bench.pl script returns directly.

Example execution step definition:

```
<execution>

<execute cname="IBM-SP4-Jump">
    <input files="../../platform/IBM-SP4-Jump/ibm_llsubmit.job.in" />

    <substitute infile="ibm_llsubmit.job.in" outfile="ibm_llsubmit.job">
        <sub from="#OUTDIR#"            to="$outdir" />
        <sub from="#STDOUTLOGFILE#"     to="$stdoutlogfile" />
        <sub from="#STDERRLOGFILE#"     to="$stderrlogfile" />
        <sub from="#CLASS#"             to="bench" />
        <sub from="#BENCHNAME#"         to="$benchname" />
        <sub from="#NODEUSAGE#"         to="not_shared" />
        <sub from="#TIME_LIMIT#"        to="00:50:00" />
        <sub from="#NODES#"             to="$nodes" />
        <sub from="#TASKSPERNODE#"      to="$taskspernode" />
        <sub from="#NOTIFICATION#"      to="never" />
        <sub from="#NOTIFY_EMAIL#"      to="email" />
        <sub from="#THREADSPERTASK#"    to="$threadspertask" />
        <sub from="#DATA_LIMIT#"        to="3.0Gb" />
        <sub from="#STACK_LIMIT#"       to="0.5GB" />
        <sub from="#MEMORYPERTASK#"     to="`3.5*$threadspertask`Gb" />
        <sub from="#EXECUTABLE#"        to="$executable" />
        <sub from="#ENV#"               to="$env" />
        <sub from="#PREPROCESS#"        to="cp run_start.h run.h; poe $executable > preprun_out.log 2> prepru
        <sub from="#POSTPROCESS#"       to="" />
        <sub from="#STARTER#"           to="poe" />
        <sub from="#ARGS_STARTER#"      to="" />
        <sub from="#MEASUREMENT#"       to="time /usr/local/bin/hpmcount" />
        <sub from="#ARGS_EXECUTABLE#"   to="" />
    </substitute>

    <environment>
        <env var="MP_LABELIO" value="yes" />
        <env var="MP_INFOLEVEL" value="2" />
        <env var="MP_SHARED_MEMORY" value="yes" />
        <env var="MP_TASK_AFFINITY" value="MCM" />
        <env var="MEMORY_AFFINITY"  value="MCM" />
        <env var="OMP_NUM_THREADS"  value="$threadspertask" />
    </environment>

    <command>llsubmit ibm_llsubmit.job</command>
</execute>

</execution>
```

The input tag describes the input file for the batch job or an interactive started script.

The substitute tag defines the replacements, further description see compile step.

The environment tag defines additional environment variables, the variable $env will contain this.

The command tag specifies the which should be used to submit the job or to start the program interactively.

1.5.6 The verification step

Example verification step definition:

```
<verification>

<!-- predefined vars:
     $subdir      -> execution dir of benchmark run
     $stdoutfile  -> $stdout file of bnechmark run
     $stderrfile  -> $stderr file of bnechmark run
     $...         -> params from benchmark specification in toplevel dir
-->

<verify cname="PEPC">
  <command>run/check_results_pepc.pl $subdir/verify.xml $stdoutfile $stderrfile $subdir</command>
</verify>

</verification>
```

1.5.7 The analysis step

The configuration file for the analysis step contains search pattern for scanning the output files (stdout, stderr) of the benchmark run. JUBE accept regular expression as defined in Perl. It is also possible to define derived results varables as expressions of other variables of the analysis step.

Example analysis step definition:

```
<analyzer>

<!-- Input is stdout and stderr of benchmark run -->
<!-- Standard result parameter:
     - walltime
-->
<analyse cname="IBM-SP4-Jump">
    <includepattern file="./analyse-pattern-pepc.xml" />
    <includepattern file="../../skel/hpm3patterns.xml" />
</analyse>

<!-- for old runs of nbench with cname PEPC -->
<analyse cname="PEPC">
    <includepattern file="./analyse-pattern-pepc.xml" />
    <includepattern file="../../skel/hpm3patterns.xml" />
</analyse>
```

```
</analyzer>
```

The name tag identifier for the varible, which will be used as a unique identifier in the result XML file

The unit tag String describing the unit of the measured value, will be used in output tables and also stored in the result XML file.

The mode tag can contain one of the following keywords: line (scan line-by-line), line,add (scan line-by-line and add values if they occurs more than once), derived (no scan, compute value as expression of other variables)

1.5.8 Generating result tables

There is one addition configuration file for describing how results tables printed by JUBE should be build.

Example result step definition:

```
<result>
    <show>
    nodes,taskspernode,threadspertask,ncpus,npart,nt,wscheme,
    walltime,
    HPMwtimeAvg,HPMressizeAvg,HPMFMPpercAvg,HPMpercPeakAvg,HPMflopsCPUglobal,vcheck,vcomment
    </show>

    <sort>
    nt,
    npart,
    wscheme,
    ncpus,
    walltime
    </sort>
</result>

  <!-- do not show
  -->

<!-- all
 HPMwtimeSum,
 HPMwtimeAvg,
 HPMutimeSum,
 HPMwtimeAvg,
 HPMstimeSum,
 HPMstimeAvg,
 HPMressizeSum,
 HPMressizeAvg,
 HPMtflopsSum,
 HPMtflopsAvg,
 HPMflopsWCT,
 HPMflopsUser,
 HPMFMPpercSum,
 HPMFMPpercAvg,
 HPMpercPeakSum,
 HPMpercPeakAvg,
 HPMflopsCPU,
 HPMflopsUserCPU,
 HPMflopsCPUglobal,
 walltime
-->
```

The name tag identifier for the varible, which will be used as a unique identifier in the result XML file

The show tag defines a list of variables for the columns of the output table. Possible variable names are the identifier specified in the analysis step and the params variables of the top-level XML file

The sort tag defines sort order by a list of variables, each variable can have a flag (-,+) indicating whether to sort in descending or ascending order.

1.6 Frequently Asked Questions

This section covers frequently asked question on how to run or configure the JUBE environment.

1.6.1 General questions

1. **When I call JUBE with the parameters `-update` and `-result` I don't get the expected results printed. Why?**

   Each individual analysis pattern has to have at least one entry for the $walltime variable. If the variable is not set after output analysis, no report on this run will be generated. The reason for an unset walltime variable can be i) the job did not run yet, ii) the job ended prematurely and did not output the appropriate line in the output, or iii) the regular expression defined for the walltime variable does not match correctly.

   First address for troubleshooting is to check the ouput of `stdout` and `stderr` in the `logs/` directory.

2. **I would like to run an application with a different numbers of cores than what is pre-configured in the official benchmark files. In addition to changing the `task` tag in the top-level benchmark file, what other modifications I need to make?**

   Detailed information on the input data for an individual benchmark is given in its README. In general the JUBE environment is capable of generating input data tailored to the task description, if the application supports this. This means no other modifications need to be done, while the tasks stay in the supported range.

1.6.2 DEISA Benchmark Suite

1. **Where can I get CPMD, DL_POLY, and Fenfloss sources and how can I import them into the JUBE environment?**

   The benchmarking harness for those applications is supplied with the benchmark suite. The README file of the corresponding application will give detailed information on how and where to obtain the source code. The harness itself is configured in a way that you only have to place the sources into the `src/` directory of the application's benchmarking harness.

2. **Can ECHAM5 benchmark be run with more cores than 496? Are there any rules (or recommendations) in choosing the parameters "nproca" and "nprocb"?**

3. **The application benchmarks are configured up to 512 processes. What do I have to do, to use more than that?**

   512 processes is the maximum number of processes the DEISA Benchmark Suite has been tested with. Using more processes will not work on applications where static input data for a given number of processes is needed. Check the `prepare.xml` to see how the input data is prepared for each run, to set it up manually.