# JuBE Tutorial

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 Jülich, Germany.

This tutorial intends to give a guideline for integrating software packages in JuBE using the example of Intel® MPI Benchmarks 3.2 (IMB). This benchmark is freely available and can be downloaded from the Intel webpage ( http://software.intel.com/en-us/articles/intel-mpi-benchmarks/). If you are not familiar with this software we recommend to compile it and to try out some of its features before reading this tutorial. In this manner it is more easy for you to understand the features of JuBE because you recognize which parts are JuBE specific and which parts are allocated to IMB.

We will show step by step what has to be done to let the program run under the control of JuBE. We describe the integration on an Intel Nehalem cluster, so please be aware that the settings given in this tutorial will most likly not function on your machine. If you are not familiar with XML it would be a good idea to read a tutorial about that topic before you start but don't worry: for a successful use of JuBE only the very basic concepts of XML are necessary.

JuBE shows the following general structure

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

The applications/ subdirectory contains the individual benchmark applications. The bench/ subdirectory contains the benchmark 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. All information that are essential for the integration process are organized in several XML files which will be introduced in this tutorial.

# 1. Define the Platform and the Job Script

First of all we have to define the platform on which our software should run. We have to make clear to JuBE which compilers are available on the machine under consideration, which libraries are available and so on. These kind of information has to be included in the file "platform.xml" which is located in "platform/". The following figure shows the general settings for the Nehalem cluster:

```
<platforms>
  <platform name="Intel-Nehalem-HPC-FF">

            mpi_f90          = "mpif90"
            mpi_f77          = "mpif77"
            mpi_cc           = "mpicc"
            mpi_cxx          = "mpicxx"

            cxxflags         = "-fast"
            cflags           = "-fast"
            f77flags         = "-fast"
            f90flags         = "-fast"

            ldflags          = "-i-dynamic"
            mpi_dir          = ""
            mpi_lib          = ""
            mpi_inc          = ""
            mpi_bin          = ""

            blas_dir         = ""
            blas_lib         = "-L/opt/intel/Compiler/11.0/074/mkl/lib/em64t"

            mkllib           = "mkl"
            mkl_lib        = "-L/opt/intel/Compiler/11.0/074/mkl/lib/em64t"
            mkl_inc        = "-I/opt/intel/Compiler/11.0/074/mkl/include"
            mkl_version    = ""

            fftlib         = "mkl"
            fft_lib        = "-L/opt/intel/Compiler/11.0/074/mkl/lib/em64t"
            fft_inc        = "-I/opt/intel/Compiler/11.0/074/mkl/include"
            fft_version    = ""

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

Above all you have to choose a name for your platform, in our example we chosed:

```
<platform name="Intel-Nehalem-HPC-FF">
```

Later on this platform will be invoked by the top level file which we discuss later. You are also able to define more than one platform in platform.xml and that's what you probably will do when you are interested in benchmark runs. To make the available compilers and libraries known to JuBE, just define some arbitrary names and apply the corresponding compilers as well as the libraries to them, like:

```
mpi_f90    = "mpif90"
blas_lib   = "-L/opt/intel/Compiler/11.0/074/mkl/lib/em64t
```

The same can be done with any other information like compiler flags, tools that have to be used (make, ar ...) etc. But keep in mind that it is important to consider the structure of the XML files, otherwise you very likely provoke errors. In the next step we have to create a template file for the job script which will be processed by the resource manager. The definitions hereby depends on your resource management system. On our platform we use Torque, which is a free available version of PBS (Portable Batch System). The template file has to be created in a subdirectory of the platform/ directory:

- JuBE/
    - platform/
        - Intel-Nehalem-Cluster/intel_PBSsubmit.job.in

That's the content of "`intel_PBSsubmit.job.in`"

```
#!/bin/bash -x

#### CPU time limit
#PBS -l cput=#TIME_LIMIT#

#PBS -S /bin/bash

#PBS -l nodes=#NODES#:ppn=#NCPUS#

#PBS -M #NOTIFY_EMAIL#
#PBS -N #BENCHNAME#
#PBS -o #STDOUTLOGFILE#
#PBS -e #STDERRLOGFILE#

cd ${PBS_O_WORKDIR}
#ENV#
echo "<jobstart at=\"`date`\" />" >> #OUTDIR#/start_info.xml
#PREPROCESS#
#MEASUREMENT# #STARTER# #ARGS_STARTER# #EXECUTABLE# #ARGS_EXECUTABLE#
cd ${PBS_O_WORKDIR}
#POSTPROCESS#
echo "<jobend at=\"`date`\" />" >> #OUTDIR#/end_info.xml
```

You are completely free concerning the naming of the subdirectory and the template file. By convention the name of any template files ends with the extension ".in". The variable parts of the file which have to be substituted when JuBE is invoked are labeled by a hash (#) at the beginning and at the end of the variable. In order to avoid errors arising from case sensitivity it is a good idea to use only capital letters for these variables. Please note that the hashes before the PBS command don't indicate variables but mark the commands for PBS.

## 2. Integrate the Software

After defining the platform and the template for the batch system we can go on with the integration of the software package. To do so we swap to the applications/ directory. Here we have to create a subdirectory which includes all the necessary files for the integration process. Since we want to integrate IMB we name the directory "IMB/":

- JuBE/
    - applications/
        - IMB/

· .bench_current_id.dat
· jube-Intel-Nehalem-HPC-FF.xml
· compile.xml
· execute
· prepare.xml
· analyse.xml
· result.xml
· verify.xml
· src/
  - IMB_3.1
  - IMB_3.2
· input/
· run/

Before we start to modify our XML files we copy the source code to the src/ directory and since there could exist several versions of the software it makes sense to create subdirectories for each version. In our case we want to integrate version 3.2 of IMB so we create a subdirectory called IMB_3.2 just in the src/ directory and store the source code there. Now let's have a closer look to the files given above.

## .bench_current_id.dat

This file just keeps the id number of the latest JuBE run, that is used for the creation of distinct filenames and directory name for the benchmark run procedure. JuBE creates this file automatically.

## jube-Intel-Nehalem-HPC-FF.xml

The top level file "`jube-Intel-Nehalem-HPC-FF.xml`" comprises all steps that have to be triggered for a full benchmark run. This file doesn't follow any naming convention so that you can name it as you like. But it's best practice to refer the naming to the platform for which it is used since there could be more than one kind of such a file in this directory and following the suggested naming convention makes it easy to distinguish between them. The top level file shows the following structure:

```
<bench name    = "IMB"
       platform= "Intel-Nehalem-HPC-FF" >

<!-- ******************************************************* -->

<benchmark name="IMB_3.2_latency" active="1">
 <!-- version="reuse|new" -->
 <compile     cname="$platform" version="new" />
 <tasks       threadspertask="1" taskspernode="2" nodes="1"  />
 <params      ncpus="$taskspernode"
              type=""
              multi=""
              npmin=""
              msglen=""
              input=""
              iter=""
              time=""
              mem=""
              mapx="$nodes"
              mapy="$taskspernode"
              />
 <prepare     cname="IMB_3.2" />
 <execution   iteration="1" cname="$platform" />
 <verify      cname="IMB_3.2" />
 <analyse     cname="IMB_3.2" />
</benchmark>

<!-- ******************************************************** -->
</bench>
```

It starts with the root element called "`<bench>`". There are two attributes, namely "`name`" and "`platform`". For "`name`" we choose the name of the software which we want to integrate in JuBE. "`platform`" specifies the platform settings which should be taken and which is included in "`JuBE/platform/platform.xml`". Within the top level file it is allowed to include an arbitrary number of benchmark run configurations, they are distinguished by different benchmark names. In our case we only have one benchmark run which we call "`IMB_3.2_latency`". This run can be activated through the `active` attribute, whereby "1" stands for "let it run" and "0" indicates that the configuration must not be considered. If you have more than one benchmark configuration for IMB defined within the top level file you can switch on and off the several configurations by "`active`". Hint: Never turn on more than one benchmark run configuration because only the very last configuration in the file will be considered by JuBE.

The "`<compile>`" section triggers the compilation of the software. "`cname`" invokes a compilation configuration set included in compile.xml. There could exist different definitions of compilation sets so it is important to define a name at this point. In "`cname=$platform`" $platform will be substituted by the setting in "`platform`" above which results in "`cname=Intel-Nehalem-HPC-FF`" in our case. "`version`" can be set to "new" or "reuse". "new" triggers the compilation step while "reuse" causes that the compilation step will be skipped. If there doesn't exist an executable the compilation will start anyway. The executable will be stored in run/ which will be created in the first run.

In the "`<tasks>`" section all information are set which are needed to let the executable run on the platform. The settings are very descriptive, threadspertask defines the number of threads which should be used per task and so on. This piece of information will automatically be included in the job file when JuBE starts its work. One of JuBE greatest advantages is that it is possible to span a multidimensional parameter space. For example, if you want to let the executable run on a different number of nodes you can define this within one step:

```
<tasks       threadspertask="1" taskspernode="2" nodes="1,2,3,4"  />
```

If you additionally want the program run on a different number of nodes with a different number of tasks per node, just define:

```
<tasks       threadspertask="1" taskspernode="2,4,8" nodes="1,2,3,4"  />
```

This settings result in 12 different benchmark runs which are consistently managed by JuBE. More details about the storage mechanism within JuBE will follow later.

The "`<params>`" section is meant for settings concerning the integrated software. Here, ncpus, multi, etc. are options that are dedicated to IMB.

The prepare step ensures that all additional files essential for the benchmark run are available and that the substitutions in this files are processed by JuBE. The execution step is triggered by the "`<execution>`" section. As in all sections the configuration set within the corresponding file (for example execute.xml for the execution step) can be reached by "`cname`". In the verify step a script is invoked which checks if the results make sense and if the run terminated normally. This verify script is individually and has to be adapted for each integrated software package. Last but not least in the end we want to extract the most important results from our output. This will be done by special patterns which are defined in analyse.xml. The analyse step is invoked by the "`<analyse>`" section within the top level file, of which more later. We see that the top level file brings together all steps that are necessary to let the benchmark run.

## compile.xml

All important information for the compilation are stored in compile.xml. For IMB a reasonable configuration could be:

```
<compilation>
  <!-- Version 3.2 -->
  <compile cname="Intel-Nehalem-HPC-FF">
    <src directory="./src/IMB_3.2" files="*.c *.h GNUmakefile Makefile.base make_ict.in make_mpich.in" />
    <substitute infile="make_ict.in" outfile="make_ict">
      <sub from="#MPI_CC#"      to="mpicc" />
      <sub from="#OPTFLAGS#"    to="-O3" />
      <sub from="#LDFLAGS#"     to="$ldflags" />
    </substitute>

    <substitute infile="make_mpich.in" outfile="make_mpich">
      <sub from="#LIBS#"        to="-lm" />
      <sub from="#MPI_CC#"      to="mpicc" />
      <sub from="#CPPFLAGS#"    to="$cflags" />
      <sub from="#LFLAGS#"      to="$lflags" />
      <sub from="#OPTFLAGS#"    to="$cflags" />
      <sub from="#OUTDIR#"      to="$outdir" />
      <sub from="#EXECNAME#"    to="$execname" />
      <sub from="#LIB_PATH#"    to="" />
      <sub from="#LIBS#"        to="" />
      <sub from="#LDFLAGS#"     to="$ldflags" />
      <sub from="#CPPFLAGS#"    to="" />
      <sub from="#MPI_HOME#"    to="" />
      <sub from="#MPIINCLUDE#"  to="/usr/lpp/ppe.poe/include" />
      </substitute>
      <command>(gmake -f GNUmakefile all; cp -p IMB-MPI1 $execname)</command>
  </compile>
</compilation>
```

The configuration is labeled with the name that is given in the top level file, namely "`Intel-Nehalem-HPC-FF`". The "`<src>`" element is supposed to define which files are relevant for the compilation. These files will be copies to a distinct place with a distinct naming along with all other files that are needed for the benchmark run. We'll take a look at this directory in short. The "`<substitute>`" and the "`<sub>`" elements organize the substitution mechanism for the template file. As previously mentioned we mark the template files with the ".in" extension. In our example above JuBE invokes "make_ict.in" and "make_mpich.in" and substitutes the variables in these files concerning the definitions given by the "`<sub>`" elements and creates the output files. The following example makes this point clearer:

| | |
|---|---|
| `LIB_PATH =` | `LIB_PATH =` |
| `LIBS =` | `LIBS =` |
| `CC = #MPI_CC#` | `CC = mpicc` |
| `ifeq (,$(shell which ${CC}))` | `ifeq (,$(shell which ${CC}))` |
| `$(error ${CC} is not defined through the PATH environment variable` | `$(error ${CC} is not defined through the PATH environment variable` |

| | |
|---|---|
| setting. Please try sourcing an Intel(r) Cluster Tools script file such | setting. Please try sourcing an Intel(r) Cluster Tools script file such |
| as "mpivars.[c]sh" or "ictvars.[c]sh") | as "mpivars.[c]sh" or "ictvars.[c]sh") |
| endif | endif |
| OPTFLAGS = #OPTFLAGS# | OPTFLAGS = -fast |
| CLINKER = ${CC} | CLINKER = ${CC} |
| LDFLAGS = #LDFLAGS# | LDFLAGS = -i-dynamic |
| CPPFLAGS = | CPPFLAGS = |
| export CC LIB_PATH LIBS OPTFLAGS CLINKER LDFLAGS CPPFLAGS | export CC LIB_PATH LIBS OPTFLAGS CLINKER LDFLAGS CPPFLAGS |
| include Makefile.base | include Makefile.base |

On the left hand we have our template file "make_mpich.in" and on the right hand the consequent makefile "make_mpich". Whenever JuBE encounters a variable labeled by the hashes it looks it up in the substitution section in compile.xml and performs the desired changes. If you want to try out different settings for the compilation step of your software package you only need to adapt the parameters in compile.xml. The same mechanism will also be applied for the execution step.
Last but not least the "`<command>`" element gives you the opportunity to define how you want to start the compilation. In most cases this is likely done by the invokation of the *`make`* command.

## execute.xml

The concepts and mechanisms for the execution step and the compilation step are substantially the same:

```
<execution>
  <execute cname="Intel-Nehalem-HPC-FF">
    <input files="../../platform/Intel-Nehalem-HPC-FF/intel_PBSsubmit.job.in" />
    <substitute infile="intel_PBSsubmit.job.in" outfile="intel_PBSsubmit.job">
      <sub from="#OUTDIR#"               to="$outdir" />
      <sub from="#LOGDIR#"               to="$logdir" />
      <sub from="#STDOUTLOGFILE#"        to="$stdoutlogfile" />
      <sub from="#STDERRLOGFILE#"        to="$stderrlogfile" />
      <sub from="#BENCHNAME#"            to="$benchname" />
      <sub from="#TIME_LIMIT#"           to="00:02:00" />
      <sub from="#NODES#"                to="$nodes" />
      <sub from="#NCPUS#"                to="$ncpus" />
      <sub from="#TASKSPERNODE#"         to="$taskspernode" />
      <sub from="#THREADSPERTASK#"       to="$threadspertask" />
      <sub from="#EXECUTABLE#"           to="$executable" />
      <sub from="#NOTIFY_EMAIL#"         to="a.schnurpfeil@fz-juelich.de" />
      <sub from="#NOTIFICATION#"         to="never"/>
      <sub from="#ARGS_EXECUTABLE#"      to="$input $msglen $multi"/>
      <sub from="#PREPROCESS#"           to="" />
      <sub from="#POSTPROCESS#"          to="" />
      <sub from="#STARTER#"              to="mpiexec" />
      <sub from="#MEASUREMENT#"          to="" />
      <sub from="#ARGS_STARTER#"         to="-np `$nodes * $ncpus`"/>
    </substitute>
    <command>qsub intel_PBSsubmit.job</command>
  </execute>
</execution>
```

First you invoke the template file - in the example given above it's *intel_PBSsubmit.job.in* - than JuBE will perform the substitutions defined by <*sub*> elements and stores the changes in the output file, namely *intel_PBSsubmit.job*. Besides the variables which are defined in other XML files JuBE has some internally defined variables like $outdir and $logdir, which ensure that your benchmark run will be stored in a distinct place with a distinct naming.

## prepare.xml

During the preparation procedure all adaptations are considered which are important for the benchmark run but which are not included in the compile or execution step. IMB provides many different MPI tests like PingPong?, Reduce, Scatter etc. Here we are only interested in the Pingpong test so the input file for IMB should only contain the keyword `PingPong` (if this is not clear than please have a look at the IMB documentation). This kind of configuration could be done by the preparation step:

```
<preparation>
 <prepare cname="IMB_3.2">
      <input files="input/benchmarks.dat.in run/imb_postprocess.pl input/lengths.dat" />
      <substitute infile="benchmarks.dat.in" outfile="benchmarks.dat">
         <sub from="#ROUTINE#"       to="pingpong" />
      </substitute>
   </prepare>
</preparation>
```

With these settings we create an input file for IMB which only contains the keyword for addressing the Pingpong test.

## verify.xml

The verification step tests your results and will be triggered by a second invokation of JuBE. We give some examples of JuBE invokations with different sets of options later in the tutorial. Here you typically call a perl script which do the testing of your results. Because this is an individual step which can't be automated the script has to be programmed by the user. But don't worry there are some examples available. For IMB the invocation step could be triggered by the following setting:

```
<verification>
  <verify cname="IMB_3.2">
    <command>
      run/check_results_imb.pl $subdir/verify.xml $stdoutfile $stderrfile
    </command>
  </verify>
</verification>
```

*check_results_imb.pl* expects the standard output file and the standard error file and greps for certain information. It's not really necessary to have such a verification script but it's vital to define at least a "pseudo" verification step because of technical reasons.

## analyse.xml and result.xml

In the first step JuBE performs the compiling and the execution as described above and in the second step - indeed JuBE has to be invoked a second time - the results can be analysed and included in a table. The analyse procedure is organized by analyse.xml and result.xml. Since these two files show a strong interaction, we consider them simultaneously at this point.

In order to make clear which results have to be extracted out of the output you can define search patterns based on regular expressions in analyse.xml. Here we give a simple example:

```
<analyzer>
  <analyse cname="IMB_3.2">
     <parm name="PingPong"      unit="MBytes/sec"    mode="line" type="float">
        \s+100000\s+$patnint\s+$patnfp\s+$patfp
     </parm>
     <parm name="test"      unit=""    mode="derived" type="float">
       $PingPong*$taskspernode
     </parm>
  </analyse>
</analyzer>

==================================================================================

# PingPong

#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000         0.35         0.00
            1         1000         0.33         2.93
           10         1000         0.33        29.03
          100         1000         0.38       251.97
         1000         1000         0.56      1704.30
        10000         1000         2.24      4263.26
       100000          419        11.72      8139.87


# All processes entering MPI_Finalize
```

With the pattern defined in the analyse section we match the last row in the table of the `PingPong` results and extract the last number, i.e. 8139.87. How does this regular expression work? To make this point clear we only consider the regular expression:

```
\s+100000\s+$patnint\s+$patnfp\s+$patfp
```

\s matches one or more free spaces at the beginning of a line in the output. The next pattern, 100000, than matches the last row of the table. Since we are interested in the last number of that line, we have to skip two values, i.e. "`419`" and "`11.72`". This is fulfilled by "`$patnint`" and "`$patnfp`". These patterns, which are defined within JuBE, match an arbitrary integer number and floating point number respectively. Of course you are free to define such patterns by yourself. The last pattern "`$patfp`" matches our final number and stores it in the variable `PingPong`, the name that was chosen for the *name* attribute of the "`<parm>`" element. Please recognize that "`$patfp`" differs from "`$patnfp`". In order to extract a peace of data you have to put your corresponding pattern in braces (brace-matching for Perl regular expressions, that's the mechanism behind `$patfp`). This kind of searching can be done in the *line* mode set in the "`<analyse>`" section of the example above. *mode="derived"* allows to create derived numbers from known values. The example hereto makes not much sense but shows how it works.

The entries for the table of the final results are defined in result.xml:

```
<result>
```

```
  <show active="1" colw="22" >
    taskspernode, PingPong, test
  </show>

  <sort active="1">
    PingPong
  </sort>
</result>
```

Here we define via "`<show>`" which results should be represented in our result table. The sort section in result.xml gives the opportunity to choose wich column should be used for the ordering of the values in the table. This make sense if you include results from more than one calculation. In our example the results would be ordered with descending values of the `PingPong` variable.

Now everything is done what is needed to integrate IMB in JuBE. We have defined a platform and a job script, we did the settings for the compilation and the execution step and finally we described how we can extract our results and organize them in tables. By the way we learnt the meaning of the prepare and verify step.

# 3. Using JuBE

Having integrated IMB in JuBE we are now able to perform some test runs. We will go through the procedure again step by step to make clear what JuBE actually does and how the single benchmark runs are organized and stored. JuBE is invoked by the command *jube*. If you don't set any options you will get a short overview of the available options:

```
schnural@zam009:~/SUBVERSION/JuBE/applications/IMB> jube
Usage: /home/schnural/bin/jube <options> <xml-file> <id-range>

                -start, -submit *  : submit new set of benchmark runs (defined in xml-file)
                -update         +  : scans for results of finished jobs
                -result         +  : shows results of benchmark runs (tables)
                -force          +  : force a rescan of benchmark output files for new results
                -cdir <dir>        : directory containing the xml files
                                     (default: ./)
                -pdir <dir>        : directory containing platforms definition XML files
                                     (default: ../platforms)
                -tmpdir <dir>      : directory which is used for running the job in,
                                     please use only an absolute path
                                     (default: tmp in benchmark directory)
                -verbose level     : verbose
                -dump              : dump XML-file structure
                -showall           : shows all results, incl. failed and queued runs
                -debug             : don't submit jobs
                -rmtmp             : remove temp directory directly
                -cmpdir <dir>      : directory which is used for running the compile step in,
                                     please use only an absolute path
                -Version           : prints out the current version
      * : needs XML top level file  <xml-file>
      + : a range of benchmark run ids can be specified <id-range>
```

If everything is correct, we should have the following files and subdirectories in our *IMB* directory:

- JuBE/applications/IMB
    - .bench_current_id.dat
    - analyse.xml

- compile.xml
- execute.xml
- input/
- jube-Intel-Nehalem.xml <--- top level file
- prepare.xml
- result.xml
- run/
- src/
- verify

Maybe you want to begin with the checking of your configuration and you start the procedure with:

```
jube -start <top level file> -debug -verbose 5
```

*-debug* or *-de* invokes the debug mode included in JuBE which triggers the compilation and the setup of the environment but it doesn't submit the job. The *-verbose [1..5]* or *-ve [1..5]* option gives additional information. Now let's start our first run:

```
xml -debug -verbose 5
-----------------------------------------
16 09:09:44 2009
-----------------------------------------

-----------------------------------------
1
-----------------------------------------
5
jube-Intel-Nehalem-HPC-FF.xml
$PWD/compile.xml
$PWD/benchlog/benchlog_000001.log
$PWD
/home/schnural/SUBVERSION/JuBE/bench/../platform
-----------------------------------------
HPC-FF.xml ...
in 0.0963 sec
.

.

.

ERSION/JuBE/bench/../platform/platform.xml ...
/bench/../platform/platform.xml in 0.4419 sec

ehalem-HPC-FF:
e=$platform  (Intel-Nehalem-HPC-FF) -> Identifier=IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001
enerating temporary directory $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001
enerating run step $platform
  1 : 1 nodes 2 tasks 1 threads
 nodes = 1
 taskspernode = 2
 taskspernode = 2
 nodes = 1
 taskspernode = 2
 taskspernode = 2
      1: [iter-> ][msglen->-msglen lengths.dat][input->-input benchmarks.dat][mapx->1][time-> ][ncpus->2][npmi
 platform = Intel-Nehalem-HPC-FF
            -> generating temporary directory $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_
```

```
           -> compile step $platform (Intel-Nehalem-HPC-FF)
 platform = Intel-Nehalem-HPC-FF
 platform = Intel-Nehalem-HPC-FF
   key= >execname< >$PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01/IMB_Intel-Nehalem
   key= >msglen< >-msglen lengths.dat< rc=0
   key= >input< >-input benchmarks.dat< rc=0
   key= >time< > < rc=0
   key= >pdir< >/home/schnural/SUBVERSION/JuBE/bench/../platform< rc=0
   key= >rundir< >$PWD/run< rc=0
   key= >npmin< > < rc=0
   key= >subid< >n1p2t1_t001_i01< rc=0
   key= >subdir< >$PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01< rc=0
   key= >cname< >Intel-Nehalem-HPC-FF< rc=0
   key= >platform< >Intel-Nehalem-HPC-FF< rc=0
   key= >id< >IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001< rc=0
   key= >mapy< >2< rc=0
   key= >iter< > < rc=0
   key= >tasks< >2< rc=0
   key= >mapx< >1< rc=0
   key= >outdir< >$PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01< rc=0
   key= >name< >IMB_3.2_latency< rc=0
   key= >benchhome< >$PWD< rc=0
   key= >ncpus< >2< rc=0
   key= >threadspertask< >1< rc=0
   key= >benchname< >IMB< rc=0
   key= >taskspernode< >2< rc=0
   key= >mem< > < rc=0
   key= >type< > < rc=0
   key= >multi< >multi 1< rc=0
   key= >nodes< >1< rc=0
 execname = $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01/IMB_Intel-Nehale ...
                  copy files/dirs: *.c *.h GNUmakefile Makefile.base make_ict.in make_mpich.in
                  executing: cp -rp ./src/IMB_3.2/*.c $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001
                  executing: cp -rp ./src/IMB_3.2/*.h $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001
                  executing: cp -rp ./src/IMB_3.2/GNUmakefile $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency
                  executing: cp -rp ./src/IMB_3.2/Makefile.base $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_laten
                  executing: cp -rp ./src/IMB_3.2/make_ict.in $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency
                  executing: cp -rp ./src/IMB_3.2/make_mpich.in $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_laten
                   sub: make_ict.in -> make_ict
          (1)  #01 #OPTFLAGS#           -> -O3
          (1)  #01 #LDFLAGS#            -> -i-dynamic
          (1)  #01 #MPI_CC#             -> mpicc
          (2)  #00 #OPTFLAGS#           -> -O3
          (2)  #00 #LDFLAGS#            -> -i-dynamic
          (2)  #00 #MPI_CC#             -> mpicc
                   sub: make_mpich.in -> make_mpich
 outdir = $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01
          (1)  #00 #OUTDIR#             -> $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB...
 cflags = -O3
          (1)  #01 #OPTFLAGS#           -> -O3
          (1)  #01 #MPI_HOME#           ->
 ldflags = -i-dynamic
          (1)  #01 #LDFLAGS#            -> -i-dynamic
          (1)  #02 #MPI_CC#             -> mpicc
          (1)  #01 #LIB_PATH#           ->
          (1)  #00 #MPIINCLUDE#         -> /usr/lpp/ppe.poe/include
          (1)  #01 #LIBS#               ->
          (1)  #01 #CPPFLAGS#           ->
 execname = $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01/IMB_Intel-Nehale ...
          (1)  #00 #EXECNAME#           -> $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB...
$lflags: lflags
          (1)  #00 #LFLAGS#             -> $lflags
 outdir = $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01
          (2)  #00 #OUTDIR#             -> $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB...
 cflags = -O3
          (2)  #00 #OPTFLAGS#           -> -O3
          (2)  #00 #MPI_HOME#           ->
 ldflags = -i-dynamic
          (2)  #00 #LDFLAGS#            -> -i-dynamic
          (2)  #00 #MPI_CC#             -> mpicc
          (2)  #00 #LIB_PATH#           ->
          (2)  #00 #MPIINCLUDE#         -> /usr/lpp/ppe.poe/include
          (2)  #00 #LIBS#               ->
          (2)  #00 #CPPFLAGS#           ->
 execname = $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01/IMB_Intel-Nehale ...
          (2)  #00 #EXECNAME#           -> $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB...
$lflags: lflags
          (2)  #00 #LFLAGS#             -> $lflags
                  executing compile command: (cd $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2
                  executing: cp -p $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01/IM
            -> prepare step IMB_3.2 (Intel-Nehalem-HPC-FF)
                  prep input files: input/benchmarks.dat.in run/imb_postprocess.pl input/lengths.dat
                    executing: cp -rp input/benchmarks.dat.in $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency
                    executing: cp -rp run/imb_postprocess.pl $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_
                    executing: cp -rp input/lengths.dat $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i0000
                   sub: benchmarks.dat.in -> benchmarks.dat
          (1)  #01 #ROUTINE#            -> PingPong
          (2)  #00 #ROUTINE#            -> PingPong
            -> execute step $platform (Intel-Nehalem-HPC-FF)
 platform = Intel-Nehalem-HPC-FF
                  copy files: ../../platform/Intel-Nehalem-HPC-FF/intel_PBSsubmit.job.in
                    executing: cp -rp ../../platform/Intel-Nehalem-HPC-FF/intel_PBSsubmit.job.in $PWD/tmp/IMB_I
                   sub: intel_PBSsubmit.job.in -> intel_PBSsubmit.job
 nodes = 1
 ncpus = 2
>`1 * 2`<
          (1)  #01 #ARGS_STARTER#       -> -np 2
 outdir = $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01
          (1)  #02 #OUTDIR#             -> $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB...
 ncpus = 2
          (1)  #01 #NCPUS#              -> 2
 stderrlogfile = $PWD/logs/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001.n1p2t1_t001_i01_stderr.log ...
          (1)  #01 #STDERRLOGFILE#      -> $PWD/logs/IMB_Intel-Nehalem-HPC-FF_IM...
 threadspertask = 1
          (1)  #00 #THREADSPERTASK#     -> 1
          (1)  #01 #STARTER#            -> mpiexec
 mapx = 1
          (1)  #00 #MAPX#               -> 1
 taskspernode = 2
          (1)  #00 #TASKSPERNODE#       -> 2
 logdir = $PWD/logs
          (1)  #00 #LOGDIR#             -> $PWD/logs
 type =
          (1)  #00 #TYPE#               ->
          (1)  #00 #NOTIFICATION#       -> never
 executable = $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01/IMB_Intel-Neha ...
          (1)  #01 #EXECUTABLE#         -> $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB...
 nodes = 1
          (1)  #01 #NODES#              -> 1
          (1)  #01 #PREPROCESS#         ->
 stdoutlogfile = $PWD/logs/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001.n1p2t1_t001_i01_stdout.log ...
          (1)  #01 #STDOUTLOGFILE#      -> $PWD/logs/IMB_Intel-Nehalem-HPC-FF_IM...
          (1)  #01 #TIME_LIMIT#         -> 00:02:00
 mapy = 2
```

```
          (1)  #00 #MAPY#               -> 2
          (1)  #01 #MEASUREMENT#        ->
 msglen = -msglen lengths.dat
 input = -input benchmarks.dat
 multi = multi 1
          (1)  #01 #ARGS_EXECUTABLE#    -> -input benchmarks.dat -msglen lengths.dat multi 1
          (1)  #01 #POSTPROCESS#        ->
 mapx = 1
 mapy = 2
          (1)  #00 #MAP#                -> -map 1x2
 benchname = IMB
          (1)  #01 #BENCHNAME#          -> IMB
          (1)  #01 #NOTIFY_EMAIL#       -> a.schnurpfeil@fz-juelich.de
 nodes = 1
 ncpus = 2
>`1 * 2`<
          (2)  #00 #ARGS_STARTER#       -> -np 2
 outdir = $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01
          (2)  #00 #OUTDIR#             -> $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB...
 ncpus = 2
          (2)  #00 #NCPUS#              -> 2
 stderrlogfile = $PWD/logs/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001.n1p2t1_t001_i01_stderr.log ...
          (2)  #00 #STDERRLOGFILE#      -> $PWD/logs/IMB_Intel-Nehalem-HPC-FF_IM...
 threadspertask = 1
          (2)  #00 #THREADSPERTASK#     -> 1
          (2)  #00 #STARTER#            -> mpiexec
 mapx = 1
          (2)  #00 #MAPX#               -> 1
 taskspernode = 2
          (2)  #00 #TASKSPERNODE#       -> 2
 logdir = $PWD/logs
          (2)  #00 #LOGDIR#             -> $PWD/logs
 type =
          (2)  #00 #TYPE#               ->
          (2)  #00 #NOTIFICATION#       -> never
 executable = $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001/n1p2t1_t001_i01/IMB_Intel-Neha ...
          (2)  #00 #EXECUTABLE#         -> $PWD/tmp/IMB_Intel-Nehalem-HPC-FF_IMB...
 nodes = 1
          (2)  #00 #NODES#              -> 1
          (2)  #00 #PREPROCESS#         ->
 stdoutlogfile = $PWD/logs/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000001.n1p2t1_t001_i01_stdout.log ...
          (2)  #00 #STDOUTLOGFILE#      -> $PWD/logs/IMB_Intel-Nehalem-HPC-FF_IM...
          (2)  #00 #TIME_LIMIT#         -> 00:02:00
 mapy = 2
          (2)  #00 #MAPY#               -> 2
          (2)  #00 #MEASUREMENT#        ->
 msglen = -msglen lengths.dat
 input = -input benchmarks.dat
 multi = multi 1
          (2)  #00 #ARGS_EXECUTABLE#    -> -input benchmarks.dat -msglen lengths.dat multi 1
          (2)  #00 #POSTPROCESS#        ->
 mapx = 1
 mapy = 2
          (2)  #00 #MAP#                -> -map 1x2
 benchname = IMB
          (2)  #00 #BENCHNAME#          -> IMB
          (2)  #00 #NOTIFY_EMAIL#       -> a.schnurpfeil@fz-juelich.de
                  -> submit job command: [debug] qsub -q sun intel_PBSsubmit.job

------------------------------------------
         1
------------------------------------------
```

```
 09:10:20 2009
-----------------------------------------
ications/IMB>
```

JuBE gives a lot of information about the substituted data and the settings and thankfully terminates normally. The compilation step shows no errors and the configurations in the XML files are accepted by JuBE. Only some substitutions went wrong:

```
unknown vars in $lflags: lflags
                                                   (2)  #00 #LFLAGS#           -> $lflags
```

JuBE wasn't able to substitute #LFLAGS# because we didn't define $lflags before. But in this case it doesn't matter, because this value isn't needed for the benchmark run. When JuBE runs the first time it creates a couple of directories whithin *JuBE/application/IMB*:

- benchlog/
- logs/
- results/
- tmp/
- xmllogs/

The *benchlog/* directory includes files with general information about the benchmark settings. It's very similar to the information given on the command line when you start JuBE (see the example above). Each file has a special labeling including the identifier. As mentioned before the identifier is stored in *.bench_current_id.dat* and will be increased by 1 whenever JuBE is invoked for starting the debug procedure and a benchmark run respectively. 'logs/' includes the standard output and the standard error output, *results/* includes the results from the analyse step, we come back to this later. The *xmllogs/* directory inlcudes XML files that contain all information of the corresponding benchmark run. So it is theoretically possible to reproduce the full run by just considering the XML logfiles. You are free to have a look at the logfiles in your text editor but it is really difficult to read. To make things easier we also provide a XSL template file and a CSS file in *xmllogs/*. So it is possible to regard the logfiles *<whatever>.longlog* with your browser in a formatted manner. The *longlog*-files will be created when the analyse step is triggered (more on that below). However this procedure doesn't work for all browsers. While *konqueror* doesn't provide XSLT, the files can be loaded in *firefox* without problems. If the browser refuses to load the *longlog*-files just append *.xml* at the filename.

Now we invoke JuBE without the debug options and than we will have a look at the logs/ and tmp/ directory:

```
jube -start jube-Intel-Nehalem-HPC-FF.xml -verbose 5
```

JuBE compiles the software package and executes the job. If you want to avoid the compilation for each run just change the version attribute in the top level file from new to reuse. The main results are stored in the *log/* directory:

- IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000002.n1p2t1_t001_i01_stderr.log
- IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000002.n1p2t1_t001_i01_stdout.log

Opening IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000002.n1p2t1_t001_i01_stdout.log in a text editor gives:

```

/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000002/n1p2t1_t001_i01/IMB_Intel-Nehalem-HPC-FF_cname_Intel-Nehalem-
```

Now we swap to the tmp/ directory which includes all files that are necessary for the corresponding benchmark runs in unique identified subdirectories:

- JuBE/applications/IMB/tmp/IMB_Intel-Nehalem-HPC-FF_IMB_3.2_latency_i000002/n1p2t1_t001_i01/
    - IMB_Intel-Nehalem-HPC-FF_cname_Intel-Nehalem-HPC-FF.exe

- benchmarks.dat
- benchmarks.dat.in
- compile_err.log
- compile_out.log
- end_info.xml
- execute_err.log
- execute_out.log
- intel_PBSsubmit.job
- intel_PBSsubmit.job.in
- lengths.dat
- src/
- start_info.xml

It's a good idea to collect the templates as well as the substitutes files here so can easily check if all substitutions were fulfilled if the preocedure terminates with an error. It's not necessary to consider each file. They are mentioned in order to give you a guideline and hints if you do the integration by yourself.

At the end of the tutorial we want to give you a short introduction in how JuBE can be used to extract and organize your results, so we go back to the main *IMB/* directory:

```
JuBE/applications/IMB/
```

Now let us trigger the analyse step and explain the options:

```
jube -result -update 2 -verbose 5 -force -showall
```

The results should look like this:

```
tion,jobenddate,ldflags,make,mapx,mapy,mem,mkl_inc,mkl_lib,mkl_version,mkllib,module_cmd,mpi_bin,mpi_cc,mpi_cx
```

The tabel at the end of this output will also be stored in the result/ directory. If it's not clear to you, why we get the displayed table please have a look at the definitions given in *analyse.xml* and *result.xml*. *-result* tells JuBE to start the analyse step and the *-update* option has to be set if you want to create the result table a second time. Normally it is not necessary to set *-update* in the first time but nevertheless we recommend always to set this option to make things consistent. If you use *-update* without an argument the update will be applied to all available results. If you want to update the results only for a specific run, let's say the run with the identifier 2, than please use *-update* followed by the identifier like in the example above.
The *-force* option makes sure that the results will be extracted when JuBE is invoked more than one time for the analyse step.
At long last if you don't have a pattern for the walltime JuBE will assume that the benchmark run fails and refuses to create the result table. In order to avoid this behaviour you have to set the *-showall* or *-sh* option.

This tutorial intends to give you a short introduction in JuBE. There are a lot of more features which are not covered by this tutorial. But nevertheless it gives you a profound basis for your work with this nice peace of software and it will simplify your live concerning benchmarking. We highly recommend to reproduce the integration given in this tutorial and to play around with configuration settings, the patterns and the options that come along with JuBE.

Good luck!