Leibniz-Rechenzentrum
der Bayerischen Akademie der Wissenschaften

# Decision Criteria and Benchmark Description

**for the Acquisition of the**
**European High Performance Computer**
**SuperMUC at LRZ**

(23. September 2010)

**Sep 2010** **LRZ-Bericht 2010-03**

Direktorium:
Prof. Dr. A. Bode (Vorsitzender)
Prof. Dr. H.-J. Bungartz
Prof. Dr. H.-G. Hegering
Prof. Dr. D. Kranzlmüller

Leibniz-Rechenzentrum
Boltzmannstraße 1
85748 Garching

UST-ID-Nr. DE811305931

Telefon: (089) 35831-8000
Telefax: (089) 35831-9700
E-Mail: lrzpost@lrz.de
Internet: http://www.lrz.de

Öffentliche Verkehrsmittel:
U6: Garching-Forschungszentrum

# Content

# 1 Decision Criteria

Aside from adjustments to enable the **evaluation of qualitative factors**, the decision will be based on the **compute performance offered for a fixed total budget** (which includes investment as well as energy, air conditioning, and maintenance costs). The offered compute performance or ranking number R will be determined as described in chapter 1.2. The **benchmark** programs used to measure the compute performance include kernel benchmarks, programs from users of the HLRB II, and programs from the benchmark activities of DEISA and PRACE.

## 1.1 Benchmarks

### 1.1.1 Aggregate compute performance

The term *aggregate compute performance* refers to the method to calculate the performance of a **fully loaded system** for any given benchmark. In simplified terms, the method can be described as follows:

The system has to be **loaded with as many as possible identical copies** of a given benchmark. The vendor ascertains the **average compute performance per core**[1] for a given benchmark. LRZ then **multiplies** this per-core performance value with the total number of compute cores, taking a common correction factor for the fat nodes into account (see below). This yields the **aggregate compute** performance of the system.

The rationale behind this is that under certain circumstances (e.g., because of bottlenecks of memory access or of the interconnect) the aggregate performance for running **n** identical copies of a program is not always simply **n** times the performance of a single program run.

It may happen that a number of cores of the system cannot be used by the benchmark configuration. In order to account for these left-out cores in the aggregate performance, the vendor must

- run a version of the program with a smaller problem size on the remaining processors as a dummy program to produce some artificial workload, *and/or*
- run appropriate copies of the benchmark program DENSE_EIG and/or LINPACK as a dummy program to produce some artificial workload.

The dummy programs are only used to model memory and/or interconnect bottlenecks and do not themselves contribute to the performance calculations.

Nodes or cores purely dedicated for service purposes cannot contribute to the aggregate benchmark performance. The same applies for cores which are not used for computational work, while their attached physical memory is used by other cores running a benchmark program.

For simplicity, the command lines provided for the individual benchmarks indicate the requirement that the system must be filled with identical copies of a benchmark by using the flags "-C ncopies".

### 1.1.2 Performance assessment of fat nodes

For ease of handling of the performance estimates, no separate benchmark results need to be provided for thin and fat nodes. It is assumed that the majority of nodes will be thin nodes. The performance of a fat node will be evaluated to be that of a thin node multiplied with a single correction factor. The ratio of the performance of the fat nodes vs. thin nodes will be obtained solely from the SIPBENCH program.

Let $P_{sip,fat}$, $P_{sip,thin}$ and $P_{i,thin}$ be the per-core performance of SIPBENCH and benchmark i respectively, and $n_{fat}$ and $n_{thin}$ be the **number of cores** of the fat and thin nodes, respectively. The performance for benchmark i is then evaluated as:

[1] In practice, it may be sufficient to evaluate only the performance of one of the simultaneously running applications e.g., if the wallclock timings of all copies are approximately equal, to take the longest running one.

$$P_i = P_{i.thin} n_{thin} + P_{i.thin} \frac{P_{sip,fat}}{P_{sip,thin}} n_{fat} = P_{i.thin} n_{thin} (1 + \frac{P_{sip,fat} n_{fat}}{P_{sip,thin} n_{thin}}) = P_{i.thin} n_{thin} r$$

Thus, a common correction factor r will be applied to all compute-related benchmarks and the benchmark efforts can be focused on the thin nodes.

The common correction factor must be specified in the benchmark section and will be verified by LRZ.

The LINPACK benchmark for the complete system is the only exception from this rule; there the committed performance for the complete system will be measured directly.

### 1.1.3 Weights for Benchmarks

| i | Benchmark | Weight, $g_i$ (%) | Reference |
|---|---|---|---|
| | **Interconnect related benchmarks** | | |
| 1 | Link bandwidth of a node, intra island | 2 | |
| 2 | Saturated node bandwidth, intra island | 2 | |
| 3 | Bisection bandwidth , intra island | 4 | |
| 4 | Bisection bandwidth , inter island | 3 | |
| 4 | Latency, intra island | 4 | |
| 5 | Latency, inter island | 2 | |
| 6 | Collective communication, Barrier | 3 | |
| 7 | Collective communication, Allreduce | 5 | |
| | SUM | *25* | |
| | **Kernel Benchmarks** | | |
| 8 | APEX | 2 | |
| 9 | DENSE_EIG, inter island | 3 | |
| 10a | LINPACK, intra island | 2 | |
| 10b | LINPACK, whole system | 5 | |
| 12 | TRIADS | 7 | |
| 13 | SIPBENCH | 3 | |
| 14 | SPARSE_EIG, inter island | 3 | |
| | SUM | *25* | |
| | **Applications** | | |
| 15 | BQCD | 6 | |
| 16 | CP2K | 6 | |
| 17 | GADGET | 6 | |
| 18 | GENE | 5 | |
| 19 | LB-DC | 5 | |
| 20 | NAMD | 6 | |
| 21 | SEISSOL | 6 | |
| 22 | WALBERLA | 5 | |
| | SUM | *45* | |
| | **Energy efficiency** | | |
| | TFlop/s / MW | 5 | |
| | SUM | *5* | |
| | **IO** | | |
| | IOBench 1: multi-stream read/write for parallel file system | *0* | |
| | IOBench 2: multi-stream read/write for home file system | *0* | |
| | Metadata Benchmark | *0* | |
| | SUM | *0* | |

I/O benchmarks will be only used to verify the requirements described in section 3.6.

## 1.2 Procedure for ranking the performance of the offered systems

The procedure aims at ranking the offered systems according to their aggregate compute performance and evaluating their relative compute performance when compared with each other. The following evaluation scheme ensures this:

To normalize the different characteristics of the individual benchmarks, the value $V_i$ for **aggregate compute performance** for each individual benchmark ($P_i$) is defined as the ratio with respect to the best-performing one ($P_{i,best}$) among all offers.

If performance data (like GFlop/s, GByte/s, 1/s) are compared ("higher is better"), then

$$V_i = \frac{P_i}{P_{i,best}}$$

If latencies etc. are compared ("lower is better"), then

$$V_i = \frac{T_{i,best}}{T_i}$$

Thus, for each offered system, numerical values between 0 and 1 are obtained for each individual benchmark. These ratios will be multiplied by the weight factors $g_i$ stipulated for the benchmarks in section 1.1.3 and subsequently summed up:

$$R = \left( \sum_i V_i \cdot g_i \right)$$

R is denoted as the ranking number and is considered a measure for the relative strength of an offered system with respect to all offered systems. The performance of the combined Phases 1+2 is included in these considerations by prescribing a fixed improvement ratio (see *Description of Goods and Services*, section 2.4.3 as well as section 2.2.3 of this document).

## 1.3 Qualitative evaluation and final ranking

To obtain a qualitative overall impression of the system all benchmarks as well as other aspects of the offer will be examined for conspicuous characteristics that may have an impact on the achievable compute performance or the usability and manageability of the system.

For characteristics that conspicuously deviate from the average, a corresponding evaluation weight will be assigned. This will usually happen if a system differs substantially from the other systems or a clear failure to fulfil the requirements stated in this document is observed. Qualitative corrections of this kind are performed carefully by LRZ by judging against the state-of-the-art, and will be justified and explained.

Differences in the committed aggregate compute performance will not be evaluated in this step, since this information is already contained in the benchmark results themselves. However the scaling behaviour will be taken into account.

The criteria for applying corrections are described in the following section.

The ranking number R determined according to the description given in the previous section will be multiplied with a factor $1+\eta$ in order to obtain the final qualified ranking number Q.

$$Q = R\,(1+\eta)$$

The range of $\eta$ for particular feature groups is given below. The final qualitative correction by LRZ is restricted to a final range of $-0.25 \leq \eta \leq 0.25$, even if the summation over all groups would lead to lower or higher values.

Because the vendor must only provide overall commitments (red-labelled boxes) in Chapter 3 and details are only optional, those details given will be honoured only positively. However, it may happen that all offers which include such details will be positively honoured, and those which do not will not get an extra bonus.

## 1.4 Criteria for the qualitative evaluation

In many cases, besides commitments to be made, additional questions have to be answered. These questions will help to clarify essential facts relevant for the operation and usage of the offered system. The evaluation of the answers compared with other offers can result in a higher or lower ranking of the tenderer.

Aspects which will be evaluated include, but are not limited to:

**Characteristics of the benchmarks**

- scaling behaviour
- the programming effort for porting and/or optimisation
- reasonableness of the predictions and projections
- outstanding absolute performance
- commensurate and balanced performance for all benchmarks

The vendor's response to Chapter 3 of this document ("Benchmarks") is evaluated for the criteria mentioned above:

**Characteristics of the hardware**

- Whether the configuration is balanced and appropriate for the expected usage with respect to
  - CPUs, processors or nodes
  - main memory
  - mass storage
  - internal interconnect
- the relative and absolute bandwidths and latencies of the memory hierarchy, of the interconnect, and of the I/O subsystem
- topology of the internal network
- the number of cores in a shared memory node
- monitoring capabilites (e.g., component power consumption, thermals and errors, network and I/O traffic)
- capabilities to monitor the system and user behaviour and the capabilities to optimise, control and steer the usage of system resources.
- characteristics of the migration system

Primarily, the vendor's response to the following items in the document "Description of Goods and Services for the European High Performance Computer SuperMUC at LRZ" will be evaluated:

- Chapter 2.2.1
- Chapter 2.3.1
- Chapter 2.3.3 - Chapter 2.3.4
- Chapter 2.5.1
- Chapter 2.15
- Chapter 3

**Reliability, resiliency, redundancy, usability, flexibility and scalability**

- the capability of the system to support the scaling of applications to high core counts and high performance
-
- the size of the usable memory of a node
- the expected stability of hard- and software
- scalability of monitoring and administration tools
- ease of use of the system, its tools and its programming environment
- the expected downtimes caused by software maintenance and upgrades of the operating system
- seamless integration and usage of Phase 2
- versatility and general purpose characteristics of the system and its software
- handling of replacement parts
- integrity and safety of data
- re-routing and dead-link detection features within the interconnect
- diversity and synergy of system architectures within the Gauss Centre for Supercomputing

Primarily, the vendor's response to the following items in the document "Description of Goods and Services for the European High Performance Computer SuperMUC at LRZ" will be evaluated:

- Chapter 2.2.2
- Chapter 2.3.1

- Chapter 2.3.3 - Chapter 2.3.6
- Chapter 2.5.4 - Chapter 2.5.8
- Chapter 2.6
- Chapter 2.7.4
- Chapter 2.9.2
- Chapter 2.10.1

**Quality and features of operating system, programming environment, batch system, file systems, data and system management software**

- the flexibility in job administration and management
- the quality and availability of compilers, debuggers, test aids, and tools for performance analysis
- the possibilities provided by the system software to ensure high usability, a well-balanced load distribution, and operational stability as well as a high energy efficiency for the expected application profile
- the interoperability with the rest of the LRZ environment, particularly for visualisation, archiving and backup
- features available for
- monitoring and control of system usage
- the scope and availability of optimized scientific libraries and applications (including third party software)
- quality of documentation
-
- the programming effort for porting and/or optimisation
- automatically achievable compute performance (autoparallelisation/autovectorisation)
- software developed by the vendor and the measure of control exerted over it
- well-defined processes for fixing bugs in the OS, compilers and tools
- capabilities and efficiency of the batch system
- capabilities for managing the interconnect fabric

Primarily, the vendor's response to the following items in the document "Description of Goods and Services for the European High Performance Computer SuperMUC at LRZ" will be evaluated:

- Chapter 2.5.1
- Chapter 2.5.7
- Chapter 2.7.3
- Chapter 2.8 - Chapter 2.10

**Support,cooperation and petascale references**

- quality of vendor support for the operation of the system
- possibilities for cooperations
- number and size of petascale installations

Primarily, the vendor's response to the following items in the document "Description of Goods and Services for the European High Performance Computer SuperMUC at LRZ" will be evaluated:

- Chapter 2.5.1
- Chapter 2.5.7 - Chapter 2.5.8
- Chapter 2.5.11
- Chapter 2.11
- Chapter 2.13 - Chapter 2.14

**Promising or advanced technologies; energy efficiency; infrastructure**

- cooling concept
- extent of the free cooling capability for the Munich area
- possibilities for the reuse of waste heat
- additional operation and infrastructure costs (i.e., costs that are not part of the procurement like cooling devices)

Primarily, the vendor's response to the following items in the document "Description of Goods and Services for the European High Performance Computer SuperMUC at LRZ" will be evaluated:

- Chapter 2.1.4
- Chapter 2.2.1
- Chapter 2.5.1
- Chapter 2.8 - Chapter 2.9

| **Feature group** | **Range for η** |
|---|---|
| Characteristics of benchmarks, scaling | 0.00 … +0.10 |
| Characteristics of the hardware | -0.10 … +0.10 |
| Reliability, resiliency, redundancy, flexibility and scalability of the system | -0.10 … +0.10 |
| Quality and features of operating system, programming environment, batch system, software | -0.10 … +0.10 |
| Support,cooperation and petascale references | -0.10 … +0.10 |
| Promising technologies; energy efficiency; infrastructure requirements; sustainably low operation costs. | -0.10 … +0.10 |
| **Final Range:** | **-0.25 …+ 0.25** |

# 2 Rules for the benchmarks and verification

## 2.1 General Rules

### 2.1.1 Confidentiality

Under no circumstances shall any information related to the design, algorithms, or source code of the benchmarks be disclosed to a third party without written consent of LRZ and the authors of the particular benchmark.

LRZ may disclose the optimisations for the application benchmarks to the respective authors of the particular program and will discuss the modifications with them.

### 2.1.2 Code modifications

Any modifications that do not change the intention, functionality and complexity order of the underlying algorithms or lower the precision of the calculations are permitted. The changes must be at least as robust as the baseline algorithm. Typical examples of allowable modifications include:

- Insertion of compiler directives
- Loop unrolling or fusion
- Blocking, including the variation of block size
- Changes of data layout or of the alignment of data
- Calls to library subroutines
- Inlining
- Reversing outer and inner loops
- Use of threads (implicit or explicit)
- Pattern matching techniques for replacing original code with calls to libraries
- Replacement of message passing constructs by shared memory constructs or by one-sided communication constructs.

Any change of the source code must be fully disclosed. A short rationale for every change must be provided.

Compilation or linkage flags that are generally supported and documented are permitted. Linking to optimized versions of vendor libraries is permitted and encouraged. The usage of all libraries must be disclosed together with the submission of the results. Source code directives which are supported and documented may be used. Language extensions instead of directives are also permitted.

If **multithreading** is used, the vendor must ensure that the size of the benchmark remains unchanged and performance values are reported on a per core basis.

### 2.1.3 Limitations of optimisation

We take it for granted that the vendor understands the intention of the loop kernels (especially those of the low-level benchmarks) and undertakes no action to circumvent the intended measurements.

The following optimisations are **not** permitted:

- **Code to circumvent the actual computation:** Any modification of the code to circumvent the actual computation is not permitted, e.g., it is not permitted to use Strassen's algorithm for DGEMM matrix multiplication. This prohibition should of course not cover standard optimisation techniques, whether by the compiler or carried out manually, which pull redundant code out of the loops and precompute it.
  Additional results using such modified codes may be supplied for qualitative evaluation.
- **Eliminate code or bypass code** which is not covered in the actual benchmark case but may be covered in other benchmark cases.
- **Optimized assembly modules:** Substituting any part of the code by optimized assembly modules or modification of compiler generated assembly code or executables is discouraged and should be disclosed when reporting the results. In this case the performance of the non-substituted code must also be disclosed. The only exception from this rule is the APEX benchmark.
- **Pre-supposing the knowledge of the solution:** Any modification of the code or input data set which makes use of known properties of the solution is not permitted.

### 2.1.4 Conversion of timing to performance values

For some programs timing values are converted to floating point operations per second by dividing a predefined operation count by the execution time (wall clock time). However, in some cases only parts of the program are measured e.g., omitting the initial phase or the final phase.

It is not allowed to change the conversion factors, even if they may appear wrong or inappropriate.

For any given benchmark program we consider the converted values as just another measurement unit of execution time, which serves to rank the results against the best performance obtained among vendors.

### 2.1.5 Rules for systems which are not available for benchmarking

The description of each benchmark program specifies which results must be delivered. In case that a benchmark program cannot be run on the proposed system, either because there is no system of the offered size available or because the proposed system hardware or software can not yet be benchmarked at all, predictions have to be made by the vendor.

These predictions are considered as committed minimal performance results. They should be based on measurements done on a roughly comparable system which is presently available and is similar in architecture to the one offered. Results obtained on smaller systems need to be carefully scaled to the predicted values for the proposed system size or to the required benchmark size. Together with the predictions, the results of actual measurements must also be disclosed in order to enable LRZ to perform a qualitative evaluation of the characteristics of the offered system. The reasoning behind all estimates and predictions should be explained. A rationale for the assumptions underlying the predictions should be given.

### 2.1.6 Commitments by the vendor and submission of results to LRZ

| |
|---|
| Commitments labeled in red must be filled in.<br>Committed values have to be demonstrated by the vendor during the acceptance procedure for the system.<br>Failure to reproduce the committed performance requires that appropriate countermeasures need to be taken by the vendor. |

Other fields in the benchmark result tables (not marked in red) should be filled in which case they may lead to an additional positive qualitative correction.

All required results and the source code of the programs should be supplied to LRZ on a DVD. The tenderer should keep an exact copy and/or checksum of this DVD.

### 2.1.7 Delivery of the benchmark sources by LRZ

The final version of the benchmark source codes as well as additional input files for the SuperMUC procurement is delivered by LRZ through

**http://www.lrz-muenchen.de/services/compute/hlrb/benchmark/**

All preliminary versions of the benchmarks and the accompanying documentation are then considered obsolete.

If it is necessary to provide fixes to the benchmarks, we will inform the tenderer by email. The fixes themselves will be available for download from the above location.

### Frequency and Power Envelope Settings

Benchmarks for the performance commitments may be run at any frequency and power envelope setting which will be available on the proposed system. However, the peak eclectrical power consumption of total system specified by the vendor in the “Description of Goods and Services” must not be exceeded.

The determination for the Energy Capping Limit will be performed in a different way and is described in chapter 3.7.

### 2.1.9 Delivery of updated commitments (compared to intermediate offer)

Updated commitments are possible. Typically, only adjustments upward should be made. This includes

- delivering better benchmark results
- delivering more hardware (which will increase the aggregate performance).

If downward adjustments are performed for a benchmark, an irrefutable reason must be provided, and compensation is required. In this case the intermediate offer is considered as a competing one. The two offers are compared against each other according to the rules given in Chapter 1.2. The benchmark ranking number of the final offer must be greater or equal the ranking number of the intermediate offer.

$$R_{final} \geq R_{intermediate}$$

Delivery of output files and documentation may contain results unchanged from the intermediate proposal. Files and documentation that have changed against the intermediate offer should be clearly marked.

## 2.2 Verification of Benchmark Commitments

### 2.2.1 Verification for Phase 1

During the **acceptance test for Phase 1,** the performance of the individual benchmarks has to be demonstrated. For each benchmark (i) the relative deviation from the committed values is computed:

$$D_i = \frac{P_{i,committed} - P_{i,demonstrated}}{P_{i,committed}} \quad \text{or} \quad D_i = \frac{T_{i,demonstrated} - T_{i,committed}}{T_{i,committed}}$$

The performance of individual benchmarks may fail to reach the committed values by a margin of 15%. Such deviations must be compensated by achieving better-than-committed values in other benchmarks. The weighted sum of all deviations of the individual benchmarks is computed as follows (see section 1.1.3 for the values of $g_i$):

$$D = \sum_i g_i \cdot D_i$$

| | |
|---|---|
| M 1: | The relative deviation for each individual benchmark ($D_i$) must be less than 15 %.<br>The weighted sum of all relative deviations (D) must be less than or equal to zero.<br>To confirm your accordance with this, check here **[ ]** |

### 2.2.2 Verification for an optional upgrade of Phase 1

If an upgrade of Phase 1 is performed, the performance of the individual benchmarks has to be demonstrated. It must be ensured that the performance after an upgrade is not less than before the upgrade: For each benchmark (i) the relative deviation from the values **before** the upgrade is computed as follows:

$$\delta_i = \frac{P_{i,before} - P_{i,after}}{P_{i,before}} \quad \text{or} \quad \delta_i = \frac{T_{i,after} - T_{i,before}}{T_{i,before}}$$

The performance of individual benchmarks may fail by a margin of 15%. Such deviations must be compensated by other benchmarks. The weighted sum of all deviations with the weights of the individual benchmarks is computed as follows (see section 1.1.3 for the values of $g_i$):

$$\delta = \sum_i g_i \cdot \delta_i$$

| | |
|---|---|
| M 2: | The relative deviation for each individual benchmark ($\delta_i$) before and after an optional upgrade must be less than 15 %<br>The weighted sum of all relative deviations ($\delta$) must be less than or equal to zero.<br>To confirm your accordance with this, check here **[ ]** |

### 2.2.3 Verification of the improvement ratio for Phase 2[2]

The benchmark set for the verification of the improvement ratio (see *Description of Goods and Services* in section 2.4.3) consists of the following benchmarks:

- LINPACK for Phase 2
- TRIADS, aggregate performance Phase 2

[2] For commitments relating to Phase 2 see “Anschreiben

- Aggregate Bisectional Bandwidth for Phase 2
- **Four** benchmarks selected out of the set of application benchmarks: aggregate performance Phase 2

The individual improvement ratio $IR_i$ is calculated as the ratio of the aggregate performance of Phase 2 and the aggregate performance of Phase 1 (as measured during the acceptance test of Phase 1):

$$IR_i = \frac{P_{i,Phase2}}{P_{i,Phase1}}$$

$$IR = \sum_i IR_i / 7 \geq 1.4$$

| | |
|---|---|
| I 1: | For Phase 2[3], the sum of all individual improvements divided by the number of benchmarks (7) should be greater than or equal to the required overall improvement ratio stipulated in *Description of Goods and Services* in section 2.4.3.<br><br>To confirm your accordance with this, check here **[ ]** |

[3] For commitments relating to Phase 2 see "Anschreiben

# 3 Benchmarks

## 3.1 Auxiliary scripts and routines

### 3.1.1 Running the Benchmarks, $RUN

A sample script `$BENCH/bin/run` is provided which can be used to start up the benchmark programs. In the benchmark description the script is denoted as $RUN:

RUN=`$BENCH/bin/run`

In addition to the internal time measurement within the benchmark programs this script performs an external time measurement.

The script takes at least five input parameters beyond the executable to be run:

| | |
|---|---|
| -n: | the (total) number of MPI processes |
| -N: | the number of nodes |
| -I: | the number of islands |
| -t: | the number of threads |
| -C: | the number of copies |

**Modifications:** Modify the script according to your needs e.g., insert calls to *mpiexec* to start the executables. Using other calls than the *time* command which provide additional information about the performance of the system (e.g., Flop-counters, hardware performance monitors/counters) is appreciated, but these commands must at least report the elapsed (real) time, user time, and system time.

This script can also be modified to fill the system with identical copies of the program, as stipulated in section 0.

### 3.1.2 Low-level library used by the benchmark programs

The directory `$BENCH/src/aux` contains routines used by more than one benchmark program e.g., timing routines for wall clock time and CPU time. Modification to the files `time.c`, `time.body.c`, and `time.body1.c` may be necessary to include machine-specific timing routines.

To build the library, type: *make*.

## 3.2 Interconnect-related benchmarks

**Precautions regarding cache**

For evaluation of the interconnect properties version 3.2 of the Intel MPI benchmark is used, which provides a highly configurable set of MPI kernels. Apart from minor changes to the build system, the only change to the benchmark source is the increase of MAXMSGLOG from 22 to 24 in **`IMB_settings.h`**. In the multi-process group version of the benchmark, disjoint groups of 2, 4, 8, etc. processes will be formed, which will all simultaneously run the benchmark routines.

Please consult the User's Guide provided with the benchmark sources, which is located at *bench/src/low_level/imb/doc/IMB_Users_Guide.pdf* in the LRZ benchmark source tree.

**To ensure that the cache is invalidated** for each measurement iteration of the program, it is required to use the `-off_cache` switch of the executable. The environment variable `CCONF` referenced in the command lines for executing the benchmark programs must contain the following two numbers, separated by a comma:

- The size of the last level cache in MBytes
- The cache line size for that cache in Bytes

Example: CCONF=3,128

### 3.2.1 MPI Benchmark: Link bandwidth of a node, one MPI task per node, intra-island

Purpose: Measure the bandwidth available to applications that use only a single MPI task per compute node.

Source: $BENCH/src/low_level/imb/src

Executable: IMB-MPI1

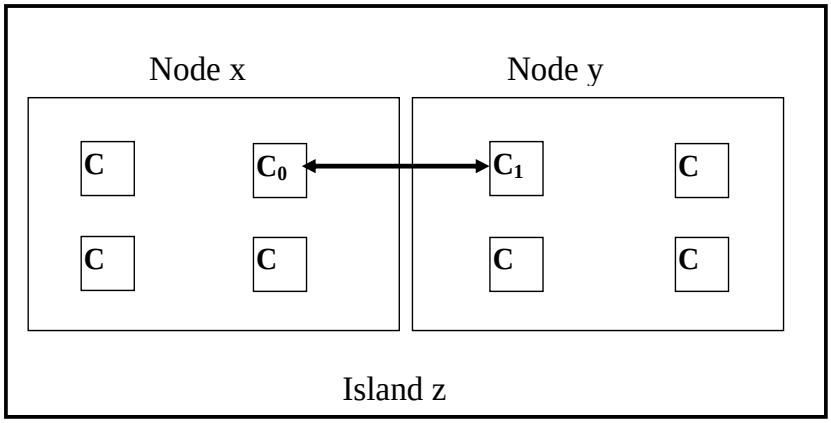
Compile: make MPI1

Procedure: Run the executable IMB-MPI1 between two arbitrarily (i.e., worst case) chosen thin nodes of an installation phase. Run **exactly one** MPI task on each node. Let the other cores of the node idle.

Command line:

```
$RUN -C 1 -I 1 -N 2 -n 2 -t 1 ./IMB-MPI1 \
   -msglen ../etc/Lengths -off_cache $CCONF \
   pingpong pingping sendrecv exchange \
      | tee > mpi.single.out
```

Example: Two nodes, each having 4 cores. Only one core of each node takes part in the communication:




Results: Deliver **`mpi.single.out`**

Report the bandwidth value for the message size of 16 MBytes (=16777216 Bytes) for the four cases specified above.

Commitment:

| **Benchmark** | **Bandwidth [GByte/s]** |
|---|---|
| pingpong | |
| pingping | |
| exchange | |
| sendrecv | |
| (these values are for reference and qualitative evaluation, take the maximum of the above values as commitment) | |
| **Maximum MPI link bandwidth for one MPI task per node pair** | |

Aggregation (will be performed by LRZ): This number will be multiplied by the number of node pairs of the offered system.

### 3.2.2 MPI-1 Benchmark: Saturated node bandwidth of a node, intra island

Purpose: Measure the aggregate bandwidth for a program which uses multiple MPI tasks per node. Effectively, the saturation bandwidth of a node is measured.

Source etc.: see above

Procedure: Run the executable IMB-MPI1 between an arbitrarily chosen Node 0 and a sufficient number of arbitrarily chosen other nodes to saturate the links of Node 0.

All processes with even process IDs must run on Node 0. In the case of using exactly 2 nodes, this can be accomplished by using the **`-map`** option of the benchmark, as indicated in the command line given below. This assumes that the MPI implementation provides a task numbering consistent with filling up nodes in sequence of their interconnect topology. Otherwise specific options of the mpiexec command must be used e.g., the multi-clause (MPMD-style) MPI start-up command.

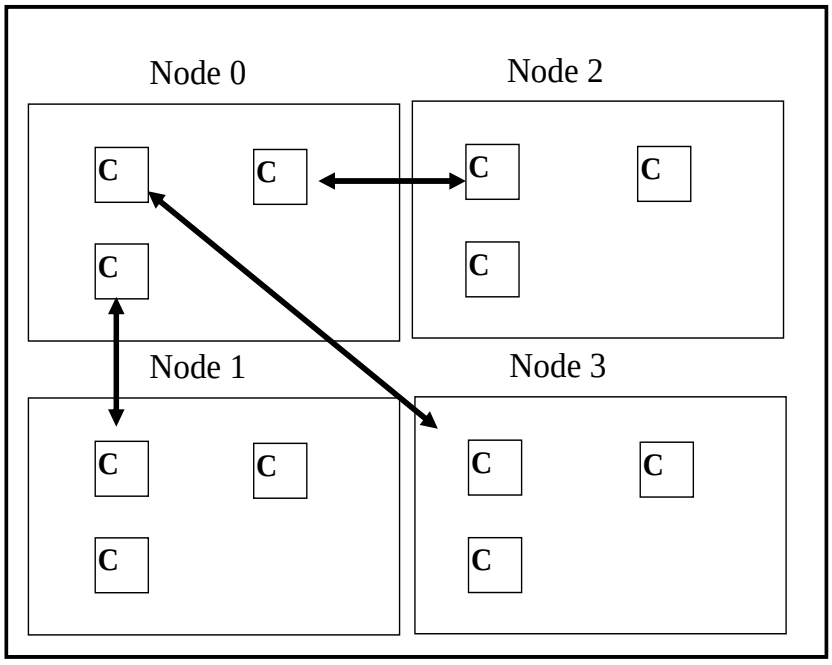
The number of processes taking part in this benchmark run is denoted NCORES; this is an even number which is chosen for optimal performance by the vendor.

Since we run the "multi" version of the benchmark, disjoint groups of processes are formed. Therefore, the reported bandwidth must be scaled with the number of processor pairs; this is half the number of MPI processes taking part in the benchmark.

Command line:

```
$RUN -C 1 -I 1 -N <nodes> -n $NCORES -t 1  \
    ./IMB-MPI1 -msglen ../etc/Lengths -multi 0 \
    -off_cache $CCONF -map $(($NCORES/2))x2 \
    pingpong pingping sendrecv exchange \
        | tee mpi.multiple.out
```

Example: Three cores on Node 0 communicate with one core each on Node 1, Node 2 and Node 3. Thus, the reported bandwidth values have to be scaled by a factor of 3.




.

If more than two nodes participate, it may be necessary to use a multi-clause (MPMD-style) MPI start-up command.

Results: Deliver `mpi.multiple.out`

Take the bandwidth value for the first part of the output, where NCORES/2 groups of 2 tasks are formed, i.e., MPI Communication Group 0 consists of the processes 0 and 1 etc.

Report the bandwidth value for the message size of 16 MBytes (=16777216 Bytes) for the four cases specified above and multiply the value by half the number of processors taking part in the communication. The maximum value must meet the commitment by the vendor for the **saturation bandwidth of one node**.

Commitment

| **Benchmark** | **Bandwidth [GByte/s]** |
|---|---|
| pingpong *** (NCORES/2)** = | |
| pingping * **(NCORES/2)** = | |
| exchange * **(NCORES/2)** = | |
| sendrecv * **(NCORES/2)** = | |
| **Number of nodes used:** | |
| **Maximum saturated MPI link bandwidth of a node set** | |

Aggregation (which will be done by LRZ) is performed by multiplying the saturated link with the number of nodes in the offer.

### 3.2.3 MPI Benchmark: Bisection bandwidth and latency, intra-island and inter-island

Purpose: Measure the bisection bandwidth and latency.

The bisection bandwidth is the minimum bandwidth over all possible bisection bandwidths. Here, the worst-case bisectional configuration within all thin nodes must be measured. Both MPI-1 and one-sided MPI-2 calls are considered.

Source: see above

Executable: IMB-MPI1, IMB-EXT

Compile: make MPI1 EXT

Procedure: The benchmark must be executed on the following *node groupings* of every installation phase:

**Case 1, intra-island: ALL** thin nodes within an island of an installation phase

**Case 2, inter-island: ALL** thin nodes of an installation phase

Divide all nodes of a *node grouping* into two equally sized sets, called “left” and “right” in the following. Use as many cores on a node as you need to reach the maximum aggregate bandwidth of the internal network, but **at least one core of each node** must take part in the communication.

Run the benchmark with pairs of cores, where the two tasks of a pair are on different sides of the configuration.

All MPI tasks with even IDs must run in the “left” set, and all MPI tasks with odd IDs in the “right” set. This is accomplished by using the **`-map`** option of the benchmark, as indicated in the command line given below. This assumes that the MPI implementation provides a task numbering consistent with filling up nodes in sequence of their interconnect topology.

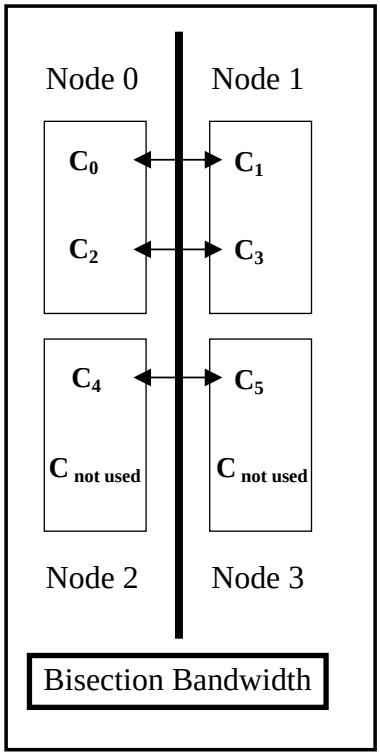
The number of processes taking part in this benchmark run is denoted NCORES.

Command lines:

```
$RUN -C 1 -I 1 -N <nodes> -n $NCORES -t 1 \
    ./IMB-MPI1 -msglen ../etc/Lengths -multi 0 \
    -off_cache $CCONF -map $(($NCORES/2))x2 \
    pingpong pingping sendrecv exchange \
      | tee mpi.bisection.case1.mpi1.out

$RUN -C 1 -I 1 -N <nodes> -n $NCORES -t 1 \
    ./IMB-EXT -msglen ../etc/Lengths -multi 0 \
```

```
    -off_cache $CCONF -map $(($NCORES/2))x2 \
    bidir_put bidir_get \
      | tee mpi.bisection.case1.mpi2.out


$RUN -C 1 -I <number of islands> -N <nodes> -n $NCORES -t 1 \
    ./IMB-MPI1 -msglen ../etc/Lengths -multi 0 \
    -off_cache $CCONF -map $(($NCORES/2))x2 \
    pingpong pingping sendrecv exchange \
      | tee mpi.bisection.case2.mpi1.out

$RUN -C 1 -I <number of islands> -N <nodes> -n $NCORES -t 1 \
    ./IMB-EXT -msglen ../etc/Lengths -multi 0 \
    -off_cache $CCONF -map $(($NCORES/2))x2 \
   bidir_put bidir_get \
      | tee mpi.bisection.case2.mpi2.out
```

Example: Four nodes with 6 MPI processes take part in this example:




Results: Deliver mpi.bisection.*.out.

Take the bandwidth value for the first part of the output, where "NCORES/2 groups of 2 processors" are formed (i.e., MPI Communication Group 0 consists of the processes 0 and 1 etc) and for the message size of 16 MBytes (=16777216 Bytes).

Take the timings for the first part of the output where "NCORES/2 groups of 2 processors" are formed, for a message size of 1 Byte. This is used as definition of the MPI latency.

Commitments:

| **Benchmark** | **Bandwidth** | |
|---|---|---|
| | **Case 1 [GByte/s]** | **Case 2 [GByte/s]** |
| pingpong | | |
| pingping | | |
| exchange | | |
| sendrecv | | |
| bidir_put aggr | | |
| bidir_put nonaggr | | |
| bidir_get aggr | | |
| bidir_get nonaggr | | |
| Take maximum of the above values | | |
| **Bisection bandwidth** | | |
| **Number of MPI tasks used** | | |
| **Number of MPI tasks per node** | | |

Aggregation of numbers to the whole system will be performed by LRZ, using the following prescription:

- Aggregation for Case 1: Bisection * (number of MPI tasks/2) * (number of islands)
- Aggregation for Case 2: Bisection *( number of MPI tasks/2)

| **Benchmark** | **Latency** | |
|---|---|---|
| | **Case 1 [µs]** | **Case 2 [µs]** |
| pingpong | | |
| pingping | | |
| exchange | | |
| sendrecv | | |
| bidir_put aggr | | |
| bidir_put nonaggr | | |
| bidir_get aggr | | |
| bidir_get nonaggr | | |
| Take minimum of the above values | | |
| **Latency** | | |

### 3.2.4 Collective Communication

Purpose: Measure MPI collective execution time as a function of task count. The following collective routines are measured: MPI_Barrier and MPI_Allreduce.

Source etc.: see above

Procedure: The benchmark must be executed with 4096, 16384, 65536 tasks on an installation phase of the offered system. A subset of nodes should be chosen which contains at least as many cores as MPI tasks are needed for the benchmark. For the reduction operation, a size of 8 Bytes is chosen. The task layout can be optimized for the target system and must be disclosed to LRZ.

Command lines:
```
$BENCH/bin/run -C 1 -I <islands> -N <nodes> -n $NCORES -t 1\
    ./IMB-MPI1 -msglen ../etc/Lengths_collective \
       -off_cache $CCONF -npmin 1024 -multi 0 \
        barrier allreduce
    | tee mpi.collective.out
```

Results: Deliver **`mpi.collective.out`**. These results are used for qualitative evaluation.
From the average timing values $t_{avg}$ contained in the output file, calculate the quantity

$$AVG = \frac{\sum_n L(n)}{3} = \frac{\sum_n \frac{t_{avg}}{\log_2(n)}}{3}, n \in \{4096,16384,65536\}$$

for the Barrier and Allreduce MPI call.

Commitment:

| **N** | **t [µs]** | **/ log2(n)** | **L(n) for Barrier [µs]** | **L(n) for Allreduce [µs]** |
|---|---|---|---|---|
| 4096 | | /12 | | |
| 16384 | | /14 | | |
| 65536 | | /16 | | |
| (these values are for reference and qualitative evaluation, take the average of the above values as commitment) | | | | |
| **AVG for collective operations** | | | | |

## 3.3 Low Level and Kernel Benchmarks

### 3.3.1 APEX

Purpose: The Apex-MAP benchmark (**A**pplication **pe**rformance **ch**aracterisation project -- **M**emory **A**ccess **P**robe) is a simple tuneable synthetic benchmark that simulates typical memory access patterns of scientific applications. The benchmark was originally developed by E. Strohmeier and H. Shang from the Future Technology Group at the Lawrence Berkeley National Lab (LBNL), California. A paper describing the benchmark is available at https://ftg.lbl.gov/ApeX/mascots.pdf. The original code provided at https://ftg.lbl.gov/ApeX/ApeX.shtml was modified by LRZ to include both random and strided memory access patterns and to add calls to increase the computational intensity.

Parameters of the benchmark are tuned to model the characteristics of the workload on the present supercomputer at LRZ.

The benchmark has the following input parameters:

| | |
|---|---|
| `access` | The regularity of the memory access, `S` for strided access patterns, `R` for random access patterns |

| | |
|---|---|
| M | the total size of the allocated memory block `data[]` in which data accesses are simulated (`M*sizeof(double)` Bytes) |
| L | the vector length of data access, (sub-blocks of length `L<M` starting at `ind[i]` are accessed in succession), describes the **spatial locality** |
| K | the shape parameter of a power distribution function (`0 ≤ K ≤ 1`), determines the random starting addresses `ind[i]`, describes the **temporal locality** |
| I | the length of the index buffer `ind[]` |
| S | the stride width |
| C | a parameter used to increase the **computational intensity** by calling the subroutine `compute(C)` |
| times | the number of times to repeat the whole kernel |

In the case of strided access, only the parameters `M, S, C` and `times` are relevant. The kernel routine for strided access sums up every S-th element of the allocated memory block `data[]`:

```
for (j = 0; j < M/S; j++) {
     W0 += c0*data[j*S];
     W0 += compute(C);
}
```

In the case of random access patterns M, L, K, C, times and I are the relevant parameters. The kernel routine for random memory access is:

```
for (i = 0; i < I; i++) {
     for (j = 0; j < L; j++) {
          W0 += c0*(data[ind[i]+j]);
          W0 += compute(C);
    }
}
```

In this mode `I` subblocks of length `L` are accessed. The starting addresses stored in the array `ind[]` are random numbers drawn from a power distribution function which is characterized by the shape parameter `K`. The smaller `K` is, the higher the temporal reuse of data is. For `K=0` always the same starting address is used, while for `K=1` the starting addresses are uniform deviates in the range `0 ≤ ind[i] < M-L`.

Braces and calls to a `dummy` routine have been inserted into the `compute` routine to assure that the floating point operations are really executed and not cancelled by optimisations of the compiler. The `dummy` routine may be removed if it is assured that the number of floating point operations is not changed by the compiler. Only modifications of the `compute` routine that enforce that the 128 floating point operations in the loop body are really executed and give the same results are allowed.

Various specific combinations of the input parameters together with weighting factors have been selected to cover the current CPU and memory usage profile of typical scientific applications.

Source: $BENCH/src/low_level/apex/

Hints: The routine compute may be inlined by defining –DINLINE as compiler flag (default); the macro definition –DDUMMY may be removed if the routine **`dummy()`** is not to be executed (see above).

Executable: Apexbm-mpi (MPI version)
*(Apexbm (serial), Apexbm-omp (OpenMP) are not used here.)*

Procedure: The input parameters of the benchmark are stored in the file Apexbm.in.

For quantitative evaluation the MPI version of the benchmark should be run (one process per core).
To ensure that the system delivers the performance homogeneously, MPI barriers have been inserted before and after each kernel, prior to taking the time steps. For each MPI task, the performance of a specific combination of the input parameters is measured and weighted to yield one overall performance number. The output file Apexbm.out contains the performance data for each parameter set measured in the MPI process with rank 0 and a mean value averaged over all MPI processes.

Run the benchmark with as many MPI tasks as there are cores within a thin node:

```
$RUN  -C <ncopies> -I 1 -N 1 -n <# of cores of thin node> -t 1 ./Apexbm-mpi
```

Results Deliver the result file Apexbm.out.

Multithreading: not permitted for this benchmark

Reference: see Apexbm.out.hlrb2

Commitment: The required mean performance and memory bandwidth per core is given in the output lines:

LRZ SUPERMUC BENCH: Mean Performance per Core [GFlop/s]: ...
LRZ SUPERMUC BENCH: Mean Memory Bandwidth per Core [GByte/s]: ...

Fill in the following table:

| | |
|---|---|
| **Mean Performance per Core [GFlop/s]** | |
| **Mean Memory Bandwidth per Core [GByte/s]** | |
| **Number of cores of Phase 1 on which benchmark can run** | |

### 3.3.2 DENSE_EIG

Purpose: The Scalapack routine PDSYTRD is a computational routine for dense symmetric eigenproblems. It reduces a symmetric matrix to real symmetric tridiagonal form by an orthogonal similarity transformation. Evaluate the performance and scaling behaviour.

Source: src/kernels/DENSE_EIG

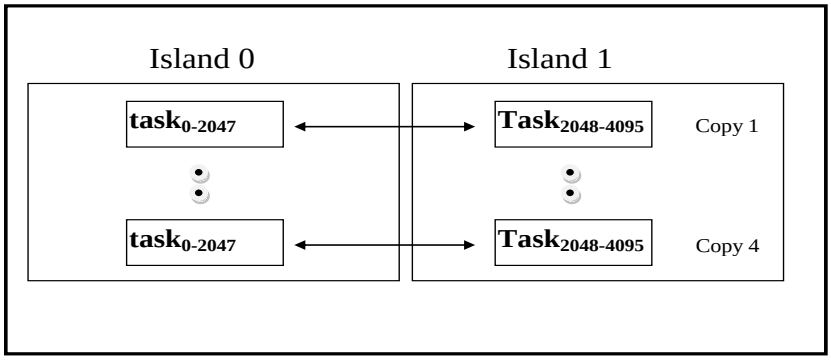
The sources for the ScaLAPACK routine itself, for BLACS, BLAS, and ScaLAPACK in general are not provided; it is expected that the vendor uses optimized version of these.

Testing: Internal testing is switched on by setting the threshold in the input file greater than zero, e.g. 10.0

Scaling: Weak/Isogranularity: The linear problem size is scaled with the square root of the number of MPI tasks

Multithreading: may be used

Procedure: Run the benchmark between *two* islands with as many copies as possible, each of which is comprised of 4096 MPI tasks, where 2048 tasks run on one island and 2048 tasks on the other. Any remaining cores on the two islands must run a smaller dummy version of the benchmark. All other island pairs of the system must run the application in a similar way (see figure).




```
#############################################################
# see also file: JOB
#############################################################

SCALE=2048
SIZE=64
PROCS=`expr  $SIZE  \* $SIZE`
N1=`expr $SCALE \* $SIZE`
cat >TRD.dat <<EOD
'ScaLAPACK TRD computation input file'
'MPI machine'
'TRD.out'         output file name
6                 device out
'L'               define Lower or Upper
```

```
1                number of problems sizes
$N1              values of N
1                number of NB's
64               values of NB
1                Number of processor grids (ordered pairs of P&Q)
$SIZE            values of P
$SIZE            values of Q
00.0             threshold
EOD
#Tests will be bypassed if threshold is set to ZERO

$RUN  -C <ncopies> -I 2 -N <# of nodes> -n 4096 -t 1\
      ./xdtrd |tee out.split.xdtrd.$PROCS.$N1.$SCALE
```

Modifications: The block size NB may be modified.

Any replacement of the Lapack routine PDSYTRD with similar calling interface and equivalent numerical results (like PDSYTTRD or PDSYNTRD) is valid.

Reference: RESULTS.HLRBII and REFERENCES

For the characteristics of the communication see:
REFERENCES/CODE_CHARACTERISTICS_*

Results: Results are reported in the line
LRZ SUPERMUC BENCH: GFLOPS/CORE
of the output file

Multithreading: If multithreading is used, the above script including the call of $RUN must be suitably modified.

Commitments:

| SIZE = sqrt(mpiprocs) | Problem size N1 | # of MPI tasks | # of Threads | # of cores (=MPI tasks * Threads) | GFLOPS/CORE |
|---|---|---|---|---|---|
| 64 | 131072 | | | 4096 | |
| Number of cores the benchmark can use on the Phase 1 offer | | | | | |

### 3.3.3 LINPACK

Purpose: The standard parallel LINPACK benchmark is used to obtain an estimate of the peak performance of the system; this benchmark also gives an impression of the quality of the vendor's BLAS implementation. For a given problem size N, $\alpha N^2$ bytes of memory storage are required; ideally, $\alpha$ should not be much larger than 8 if eight-byte floating-point words are used. If this storage is distributed across P tasks, the amount of memory per task required will be

$$M = \frac{\alpha N^2}{P}.$$

For execution on a large parallel system, a compromise may need to be made between the long run time needed versus the high fraction of peak performance achieved for large problems.

The performance L(P, N) is determined by

$$L(P,N) = \frac{\frac{2}{3}N^3 + 2N^2}{T_{measured}},$$

where $T_{measured}$ is the execution time for problem size N and task count P.

Note that the implementation is required to preserve the operation count specified above, i.e. use of a Winograd Strassen or related algorithm for performing matrix multiplications is prohibited.

Otherwise, vendor-specific implementations with respect to coding, communication mechanism and (possibly multi-threaded) BLAS library implementation are encouraged. The reference implementation HPL, available at http://www.netlib.org/benchmark/hpl/ has recently been updated; the new 2.0 release contains improvements of the scalability of the initialisation phase to systems with many hundred thousand cores of memory, bug fixes for systems with large memory and improvements in the verification algorithm. The reference implementation uses MPI and standard C. It contains a BLAS implementation, but a vendor specific C BLAS library may replace the integrated BLAS calls.

Porting the reference implementation to any multi-purpose platform typically only requires modification of an include file for Make to adjust compiler options, compiler call, MPI linkage options and, optionally, a BLAS library implementation.

Procedure: Choose the problem size as described above and measure the total performance $R_{max}$.

**Case1**: throughput (with the number of cores equal to the preferred island size, see *Description of Goods and Services* section 1.2)

```
$RUN  -C <number of copies> -I <# islands> -N <# nodes> \
      -n 8192 -t 1 ./linpack.exe
```

**Case2**: measure performance for the complete system, including all thin and all fat nodes.

```
$RUN  -C 1 -I <number of islands>  -N <number of nodes of island> \
      -n <#cores of system>   ./linpack.exe
```

Multithreading: If multithreading is used, the call of $RUN must be suitably modified.

Commitment: The performance per core has to be committed.

| | **Case 1: intra island** | **Case 2: complete system** |
|---|---|---|
| Number of cores used | | |
| Problem Size N | | |
| Total execution time [s] | | |
| $R_{max}$[TFlop/s] | | |
| Performance **per core** : = $R_{max}$/ncores **[GFlop/s]** | | |

For case 1, the following must also be provided:

| | |
|---|---|
| Number of islands of the offered Phase 1 system on which the benchmark can be run | |

### 3.3.4 TRIADS (rinf1)

This benchmark tests the floating point and integer performance of various one-dimensional loop kernels; depending on the access pattern, various aspects of the processor architecture and the memory hierarchy are tested. This benchmark is derived from the TRIADS Genesis benchmark originally developed in the HPC Department at the University of Southampton, UK.

Purpose: Of the loop types implemented, two (double precision and long integer triads) contribute to the **quantitative evaluation of the bandwidths** as well as the **degradation factor** defined as the worst-case performance ratio of strided to non-strided access (high values are worse)**.** The results for all other loop types will only be considered for **qualitative** assessment of the node architecture. The cases relevant for the quantitative evaluation are

**Case 101:** Double Precision Vector Triad

A(I) = B(I) * C(I) + D(I)

evaluated as 3 loads, 1 store and 2 floating-point operations.

Case 101 of this benchmark is very important. LRZ considers the result of this benchmark as providing the lower limit for the sustainable performance of the system.

**Case 301**: Long Integer Vector Triad

IA(I) = IB(I) * IC(I) + ID(I)

evaluated as 3 loads, 1 store and 2 integer operations.

Source: src/low_level/triads

Hints: The benchmark code is written in standard Fortran, with some infrastructure as well as kernel code in standard C. Support for C interoperability and dynamic memory management from the Fortran 2003 standard is required. The MPI variant of the code presumes the availability of a standard-conforming MPI implementation including the Fortran module **mpi**; but only functionality from MPI-1 is needed. Facilities to configure and evaluate the benchmark run are included; these require the following tools: GNU Make, bash (3.0), perl (5.8), gnuplot (4.2). Due to the efforts invested in making the code standard-conforming, any porting effort should be minimal.

Procedure: **Variant: multi-threaded**

For ***quantitative*** evaluation, an instance of the program needs to run on all cores of a node. Only a single MPI task is started which spawns OpenMP threads. Performance values are determined by running with at least 16 threads; apart from this constraint there is freedom to choose the number of threads. For the performance measurement, a logarithmically integrated average is formed for vector lengths starting with 100,000 * OMP_NUM_THREADS, and ranging up to 16 Million. Hence, cache-based systems gain a performance advantage if the cache size is larger than ~3.2 MBytes.

```
make config-mpi-multi
make

OMP_NUM_THREADS=16
$RUN -C <ncopies> -I 1 -N 1 -n 1 -t $OMP_NUM_THREADS ./triads.exe
done
```

Please consult the **doc/README.RUN** file in the source directory for instructions on how to configure and build the executables for the various runs to be performed.

Modifications: Allowed modifications include:

Choice of compiler and optimal compiler flags

Optimisation directives (compiler-specific and OpenMP) in source code. The kernels are located in the subdirectory **`loops`** (or in the case of C kernels) in the file **`triads_c_calls.c`**

Using an alternative loop: instead of Case 101, the results of Case 102 or 103 may be used. Instead of Case 301, the results of Case 302 or 303 may be used.

Reference: see subdirectory **`reference_output`**.

Commitment: Deliver all `triads*.res` output file. Write down the per-core performance indicated in the `triads_summary.*.*.res` output file. From the values in that file, the performance for contiguous and strided access are extracted, and weighted averages for floating point and integer performance must be calculated as follows:

$$P_{TRIADS} = 0.7\left(\frac{1}{2} + \frac{1}{2 * Degradation_{101}}\right) * Performance_{101} + 0.3\left(\frac{1}{2} + \frac{1}{2 * Degradation_{301}}\right) * Performance_{301}$$

| | Case 101 | Case 301 |
|---|---|---|
| Number of Threads used | | |
| Logarithmic weighted average (MFlop/s) | | |
| **$Performance_{case}$ per-core (MFlop/s)** **(also contained in the above line)** | | |
| Stride of Degradation at (len=16M, stride=<x>) (stride is automatically determined by program) | | |

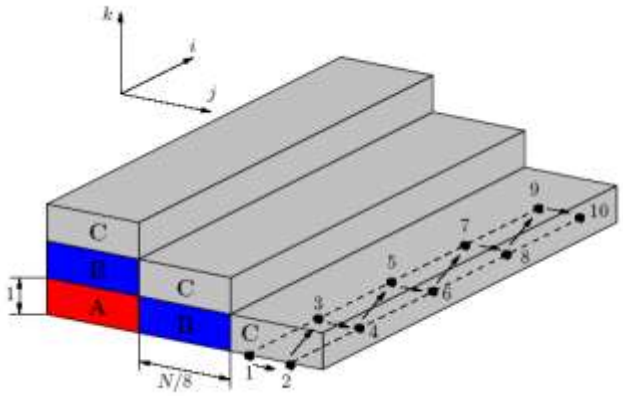
| | | |
|---|---|---|
| **Degradation$_{case}$<br>- loss of spatial locality at (len=16M stride ….<br>(stride is automatically determined by program)** | | |
| **-Averaged and weighted Performance: $P_{TRIADS}$** | | |
| **Number of cores usable for running the benchmark on Phase 1** | | |

### 3.3.5 SIPBENCH

Procedure: This is a multi-threaded strongly-implicit procedure (SIP) solver, suitable for solving systems of linear equations resulting from a discretisation of partial differential equations. It is widely used in fluid mechanics and therefore of great practical importance. Regarding the implementation, different approaches are possible. The original paper by Deserno et al. from RRZE is available at http://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/Projekte/OptGuide.pdf

The following versions of the solver can be used:

**Case 101 (**SipThreeDSolver): a straightforward way is to iterate over all nodes (i, j, k) in 3 do-loops.
**Case 102** (SipThreeDSolver_regopt): same as case 101, but one loop is split in multiple parts to improve register usage.
**Case 103** (SipThreeDSolver_ppp, pipeline parallel processing): Data dependencies may prohibit automatic parallelisation of the 3D-Version of the SIP-Solver. However, the Fortran90 compiler on the Hitachi SR8000 was able to resolve the dependencies with its specific pipeline parallel processing technique. This technique is re-programmed here with explicit OpenMP statements. The system is divided up into chunks of a certain size (see the following figure). Parallelisation is applied to the loop along the j-direction (the middle loop). Calculation of any chunk is delayed by a barrier until the chunk left of it has been processed. Consequently, blocks with equal colour in the figure are calculated concurrently. This leads to load imbalance in the “wind-up” and ”wind-down” phases of this pipeline since some CPUs have to wait for the first few chunks to be calculated; this effect is negligible for a sufficiently large lattice.




**Case 104** (SipHyperplaneSolver): A hyperplane is defined as L = i + j + k = const.
**Case 105 (SipHyperplaneSolver_sr8k):** Pipeline-parallel variant of case 104 optimized for Hitachi SR8000.
**Case 106 (SipHyperLineSolver):** Similar to hyperplanes one can define hyperlines for which L = j + k = const.

Hints: Probably Case 103 will be the best-performing one.

For getting good performance, it may be necessary to ensure that the data layout in the initialisation phase is the same as during the computation phase. For Shared Memory systems the allocation policy (e.g., First Touch) determines the appropriate initialisation strategy.

**Current versions of the Intel MKL libraries may interfere with the threading library of the compilers** (libiomp5.so). Avoid this by ensuring that any MKL specific entries in LD_LIBRARY_PATH appear after those for the compilers.

Source: $BENCH/src/low_level/sip

Procedure: Execute the benchmarks on both thin and fat nodes using 16 threads, filling the nodes with copies of the program.

**Case 1**: Thin node filled with one copy (or multiple copies, if number of cores of the node is >16)

```
OMP_NUM_THREADS=16
$RUN -C <ncopies> -I 1 -N 1 -n 1 -t $OMP_NUM_THREADS ./SipBench.exe
```

**Case 2**: Fat node filled with 2 copies (or more than 2 copies, if number of cores of the node is >32)

```
OMP_NUM_THREADS=16
$RUN -C 2 -I 1 -N 1 -n 1 -t $OMP_NUM_THREADS ./SipBench.exe
```

Modification: To avoid implications from memory layout on the performance, the problem size may be chosen by the vendor in the range between 395 and 405. The default problem size in the input file `SipInput.dat` was set to 401.

Reference outputs: see directory REFERENCE-OUTPUT.

Results: Take the maximum performance over the six code versions. In the box below, please enter the performance numbers for a single instance (running on a filled node).

Commitment: The measured performance for each case is given in the output line:

```
"Performance per core (MFlop/s):"
```

| | $p_{sip,thin}$ [MFlop/s] (Per core) | $p_{sip,fat}$ [MFlop/s] (Per core) |
|---|---|---|
| 101: Sip3DSolver | | |
| 102: Sip3DSolver_regopt | | |
| 103:Sip3DSolver_ppp (probably the best perfoming variant) | | |
| 104: SipHyperPlSolver | | |
| 105: SipHyperPlSolver_sr8k | | |
| 106: SipHyperLineSolver | | |
| (The above values are just for reference) | | |
| Max. of above Performance per core | | |

Calculation of the correction factor *r* for fat nodes (see 1.1.2)

| | |
|---|---|
| Number of thin **cores** $n_{thin}$ usable for benchmark in Phase 1 | |
| Number of fat **cores** $n_{fat}$ usable for benchmark in Phase 1 | |
| Correction factor for fat nodes: $r = (1 + \frac{P_{sip,fat} n_{fat}}{P_{sip,thin} n_{thin}}) =$ | |

where $n_{thin}$ and $n_{fat}$ are the total number of **cores** in thin and fat nodes, respectively.

### 3.3.6 SPARSE_EIG

This benchmark iteratively solves the standard symmetric eigenvalue problem corresponding to the Laplacian operator in one dimension using PETSC and SLEPC.

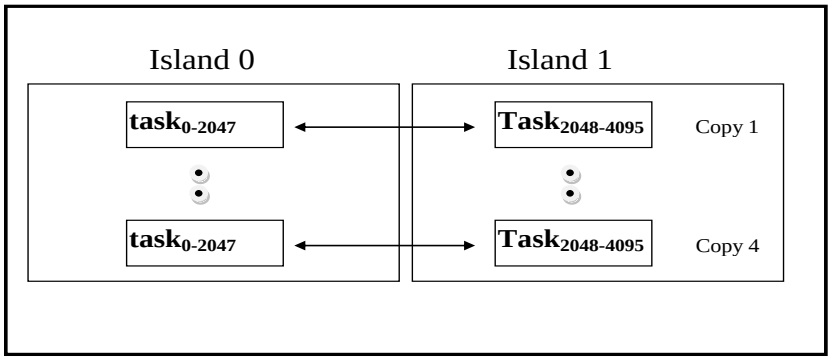
Source: $BENCH/src/kernels/SPARSE_EIG

The sources for the PETSC, SLEPC etc. are not bundled. See: http://www.mcs.anl.gov/petsc and http://www.grycap.upv.es/slepc. Version 3.0 or higher of both packages is required.

Building: The makefile depends on the correctly set-up PETSC and SLEPC environment: the `SLEPC_DIR` and `PETSC_DIR` environment variables must be set and point to the respective installation directories.

Procedure: **Case: inter-island**

Run the benchmark between *two* islands with as many copies as possible, each of which is comprised of 4096 MPI tasks, where 2048 tasks run on one island and 2048 tasks on the other. Any remaining cores on the two islands must run a smaller dummy version of the benchmark. All other island pairs of the system must run the application in a similar way (see figure).




```
NPROCS=4096
SCALE=4096
N1=`expr $SCALE \* $NPROCS`
N2=`expr $N1 \* 2`
$RUN -C <ncopies> -I 2 -N <nnodes> -n $NPROCS -t 1 \
   ./ex1f -n $N1  -eps_max_it 20000 -eps_non_hermitian \
          -eps_nev 1 -eps_tol 1E-6  -mat_view_info \
   | tee ex1f.$NPROCS.out
```

Commitment: Deliver ex1f.*.out; the required information is in that output file.

| | |
|---|---|
| LRZ SUPERMUC BENCH: Number of MPI Tasks: | 4096 |
| LRZ SUPERMUC BENCH: Iterations: | |
| LRZ SUPERMUC BENCH: Wallclock Time: | |
| LRZ SUPERMUC BENCH: Iterations per Second: | |
| Number of cores the benchmark can use on complete Phase 1 | |
| LRZ SUPERMUC BENCH: Iterations per Second and Task<br>(this is the performance per core) | |

## 3.4 Application Benchmarks

### 3.4.1 BQCD

BQCD (Berlin QCD Program) is a hybrid Monte-Carlo program that simulates Quantum Chromodynamics with dynamical standard Wilson fermions. The computations take place on a four-dimensional regular grid with periodic boundary conditions. The updates are local (i.e., only nearest neighbors are needed). The kernel of the program is a standard conjugate gradient solver with even/odd preconditioning. As a consequence, all arrays are stored in an even/odd ordered fashion and the four indices are collapsed into a single one. The access to neighbors is handled by lists.

The parallelisation is done through a regular grid decomposition in the highest 3 dimensions. The values from the boundaries of the neighboring processors are stored in the same array as the local values. The local values have indices 1, ..., volume/2. The boundary values have indices greater than volume/2.

The total domain size in the example input files is 48 x 48 x 48 x 96. The memory for the arrays is dynamically allocated during initialisation.

Apart from rounding errors the program gives identical results for any grid decomposition.

Source: $BENCH/src/apps/BQCD

Building: see: README

make prep-<platform>
make

Executable: `bqcd`

For more information about the compiled executable type: `mpiexec -n 1 ./bqcd -V`

Modifications: libd21.a (set by libd=21 in Makefile.var) may be replaced by a different version of the library. Which library is used must be disclosed.

Porting to and optimisation for your hardware may be required. Choice of compilers and flags will vary from system to system.

In addition to the usual standard optimisation techniques mentioned in 2.1.2 you are allowed to perform the following modifications for the routines **d** and **d_dag**:

- the sequence of floating point operations may be changed according to the rules of algebra
- the sequence of communication operations may be changed as long as remote data is available at the right point in time
- the data layout may be changed
- the subroutine structure within d() and d_dag() may be modified
- Fortran, C,or C++ may be used as programming languages
- any communication library (MPI, SHMEM. OpenMP) may be used
- If you change the data layout you have to make sure that outside cg() the original data structures can be used i.e., at the beginning of cg(), data would have to be copied from the original data structures to the new ones and at the end of cg() data would have to be copied back correspondingly. At a few locations outside of cg() the subroutines d() and d_dag() are called. Here necessary copy operations are also permitted.

Multithreading: Multithreading is permitted and may be useful, particularly for large core counts. By running several benchmark configurations, we compared the effect of two communication modes, pure MPI and hybrid OpenMP+MPI. We observed that the pure MPI version of the code is faster up to approx. 4096 cores. Above this number, the hybrid approach shows better performance.

Please report the number of nodes, processes and threads used.

Procedure: The input for the benchmark run is stored in the file TEST/`input.BENCH`. In `input.BENCH` the domain size is defined in the line:

```
lattice LX LY LZ LT
```

The actually used number of MPI processes per x-, y-, z- and t-direction has to be defined in the line:

```
processes NPEX NPEY NPEZ NPET
```

The total number of MPI processes is

```
NPEX * NPEY * NPEZ * NPET
```

NPEX NPEY NPEZ and NPET must be dividers of LX, LY, LZ and LT, respectively. They may be changed as long as they fit the number of cores required for benchmarking. Examples for input files are also available as `TEST/input.BENCH-*`.

Run the program for lattice size (48 x 48 x 48 x 96) with an increasing number of processes and/or threads (starting from the lowest possible number) up to the required number to show the scaling behaviour.

Command Line: Prepare the six input data sets with the desired settings for the layout (48 x 48 x 48 x 96) lattice size, number of the MPI-processes and threads

```
$BENCH/bin/run -C … -I … -N … -n … -t 1 ./bqcd  input.BENCH.512 >out.BENCH.512
etc.
```

Testing**:** Smaller test cases may be generated by modifying LX, LY, LZ, LT in `input.BENCH` to fit to the number of processors and memory

Examples: Reference input and output can be found in the directory `Reference`.

Results: Please deliver all result files out.BENCH.*

Commitments and requirements:

The performance per core of the **CG-solver** can be obtained from the output. It is contained in the table with the following heading:

```
                                                Performance
region          #calls     time       mean      min      max      Total
```

Go to the line "CG" and report the mean values for timer [s] and performance [MFlop/s].

| **Number of MPI tasks** | **Number of Threads** | **Number of cores** | **Timer value [s]** | **CG Performance per core [MFlop/s]** |
|---|---|---|---|---|
| | | 512 | | |
| | | 1024 | | |
| | | 2048 | | |
| | | 4096 | | |
| | | 8192 | | |
| | | 16384 | | |

Please also specify how many copies can run on the Phase 1 installation:

| | |
|---|---|
| Number of copies of the benchmark on the Phase 1 system, executed with 8192 cores per copy | |

### 3.4.2 CP2K

CP2K is a suite of modules containing a variety of molecular simulation methods at different levels of accuracy, from ab-initio DFT through semi-empirical NDDO approximation to classical Hamiltonians. It is used routinely for predicting energies, molecular structures, vibrational frequencies of molecular systems, reaction mechanisms, and is ideally suited for performing molecular dynamics studies.

Source: CP2K is a freely available program, written in Fortran 95, and versions of the code are available in $BENCH/src/apps/CP2K (given as reference) or from CVS:

```
touch $HOME/.cvspass
cvs -d:pserver:anonymous@cvs.cp2k.berlios.de:/cvsroot/cp2k login
cvs -z3 -d:pserver:anonymous@cvs.cp2k.berlios.de:/cvsroot/cp2k co cp2k
```

To update the source to the latest version,

```
cd cp2k
cvs update -dAP
```

Compiling: To compile CP2K you need to change into the makefiles subdirectory and most likely you will also have to adapt your architecture file in the `CP2K/arch` directory. The command `CP2K/tools/get_arch_name` will tell you which architecture will be used on your platform by default. By setting the environment variable FORT_C_NAME you can change the preferred compiler on some architectures. The architecture flag can be selected at compile time with: `make ARCH=my_arch_name` in case the guess is wrong or you want to compile versions with multiple compilers or different settings. The kind of build (sopt, popt) can be changed with `VERSION=build_type`.

Executable: The executable cp2k.popt (parallel) will be found at `./exe/your_arch/cpek.<build_type>`

Prerequisites: BLAS, LAPACK and libint version 1.1.2 or 1.1.4 should be available; MPI and SCALAPACK are needed for parallel runs.

The libint library is used to evaluate the traditional (electron repulsion) and certain novel two-body matrix elements (integrals) over Cartesian Gaussian functions used in modern atomic and molecular theory. The idea is to to generate optimized code for such integrals. See: http://www.files.chem.vt.edu/chem-dept/valeev/software/libint/libint.html

FFTW can be used to improve FFT speed, (recommended is FFTW3).

Executable: cp2k.popt

Testing: Most inputs in any of the cp2k/tests/*regtest*/ directories are tested on a daily basis:

Procedure: The input for the benchmark run is stored in the file `UO2-2x2x2.inp`; also the file `t_c_g.dat` is needed. The MAX_MEMORY parameter in the HF section of the input file can be adapted to the available core memory; this parameter specifies the memory per MPI task which is not used by the MPI implementation, the operating system and other parts of the CP2K program itself:

MAX_MEMORY ≤ physical core memory – memory(MPI + OS) – memory(CP2K)

The presently chosen value is 1700 MBytes.

```
$RUN -C …-I 1 -N … -n 1024 -t 1 ./cp2k.popt UO2-2x2x2.inp >cp2k.out.1024
```

Please deliver all result files.

Commitment: At the end of the output a table with the timing results is printed. Report the:

TOTAL TIME MAXIMUM (last column) of the line "CP2K"

The performance per core is computed as:

$$Performance\_per\_core = \frac{1}{Walltime * number\_of\_cores}$$

Fill in the following table:

| Number of cores | Walltime [sec] | Performance_per_core [1/sec] (provide enough significant digits) |
|---|---|---|
| 1024 | | |
| 2048 | | |

Please also specify how many copies can run on the Phase 1 installation:

| | |
|---|---|
| Number of copies of the benchmark on the Phase 1 system, executed with 2048 cores per copy | |

### 3.4.3 GADGET

GADGET is a code for cosmological N-body simulations. It was designed primarily for computers with distributed memory, uses MPI for explicit parallelism and can be used from personal workstations, clusters and massive parallel computers with several thousands of cores.GADGET computes gravitational forces with a hierarchical tree algorithm and represents fluids by means of smoothed particle hydrodynamics (SPH). The code can be applied to studies of isolated systems or to simulations that include the cosmological expansion of space, both with or without periodic boundary conditions. In all these types of simulations, it describes the evolution of a self-gravitating collisionless N-body system, and allows gas dynamics to be optionally included. The code can be used to address a wide array of problems in astrophysics and with the inclusion of additional physical processes such as radiating cooling and heat-

ing, it can also be used to study the dynamics of the gaseous intergalactic medium, or to address star formation and its regulation by feedback processes.

Source: $BENCH/src/apps/gadget

Executables: P-Gadget3, N-ICs

Modifications: Porting to and optimisation for your hardware may be required.

Requirements: GSL (gnu scientific library) version 1.09 -1.11.
Newer versions might also work: see http://www.gnu.org/software/gsl/
GNU Multiple Precision Arithmetic Library: see http://gmplib.org/
FFTW version 2.1.5: http://www.fftw.org/
HDF5 (optional): http://www.hdfgroup.org/HDF5/
PAPI (optional): only required for performance measurements. http://icl.cs.utk.edu/papi/.

Directory Structure: Ideally you need to have this directory structure:

Tailored files for I/O are provided in the following fashion:

| | |
|---|---|
| Benchmark: | contains the output of Gadget3. |
| bin: | the binary programs will reside here. |
| input: | input data set , parameter files, etc. |
| input/ICs: | directory containing the initial conditions generated by N-GenIC. |
| input/ics.param: | parameter file needed by N-GenIC. |
| input/param.txt | parameter file for P-Gadget3. |
| N-GenIC: | source code of Initial Condition generator. |
| P-Gadget3: | source code for gadget3. |
| Reference: | Reference information. |

Compiling: The best way to generate the binaries for your architecture is going to each source sub-folder (N-GenIC, P- Gadget3) and look for the `Makefile.def` within this folder. Those files are very similar or might be even the same, depending on what you are testing.

Based on how you build FFTW you can comment or uncomment for double/single precision or with/without prefix.

Compiling is in most cases performed by typing `make build` in the root directory.

This step will generate two binaries, which are located in the `bin` sub-directory: `bin/N-GenIC`, `bin/P-Gadget3`.

Note: The source contain a parameter macro PMGRID which controls slab parallelization of part of the applications. This parameter must not be changed.

Memory requirement: The memory requirement of the code can be adjusted by setting `PartAllocFactor` (in param.txt) to a any value not greater than 3.

Initial Conditions:

You need to edit the file `input/ics.param`, look for the entry indicated below and edit according to your architecture:

```
input/ics.param:
```

```
GlassFile input/grid_little_endian.dat
or
GlassFile input/grid_big_endian.dat
```

The input used in this benchmark consists of a set of **4096** particles (*grid_little_endian.dat* or *grid_big_endian.dat*).

If you want to generate other data sets, you can modify *input/ics.param.txt*

```
input/ics.param
```

```
…
GlassTileFac           32
Nmesh                 512
Nsample               512
…
```

For strong scaling set *GlassTileFac* to 32 (default). Modification of this value is not allowed.

It is desirable to have *Nmesh* and *Nsample* as powers of 2 since it uses FFT. *GlassTileFac* repeats the initial conditions given by the `grid_*_endian.dat` over each dimension. The size of the input and the number of particles follow a power-of-3 law:

```
Number of Particles = 4096 * GlassTileFac^3
Data Size = 0.229 * GlassTileFac^3 (MB)
```

Initial conditions are generated with:

```
mpiexec –n 32 bin/N-GenIC input/ics.param
```

The initial conditions are stored in the subdirectories: `input/ICs` (total size 7.6 GBytes)

Execution: To execute GADGET use:

```
$RUN –C <ncopies> -I 1 -N <nodes> -n <mpitasks> -t 1 bin/P-Gadget3 input/param.txt
```

Measurements: The information about the wall time is found in:

`Benchmark/run_[number_of_MPI_tasks]/cpu.txt`.

`Benchmark/run_<mpitasks>/cpu.txt:`

```
Step 0, Time: 0.02, CPUs: 128
total            81.46   100.0%
...
...

Step 49, Time: 0.020308, CPUs: 128
total            494.39   100.0%
LRZ timing value:   412.75 seconds
```

Due to initialisation overhead, step 0 is not accounted.

Reference: To check the correctness of your result compare `Reference/energy.txt` with

`Benchmark/run_<mpitasks>/energy.txt`

`Benchmark/run_<mpitasks>/energy.txt`

```
0.02 1.35258e+07 0 3.238e+11 1.35258e+07 0 4.31734e+10 0 0
2.80627e+11 0 0 0 0 0 0 0 0 0 0 0 0 8.8816e+06 5.77304e+07 0 0
0 0
```

The second value has to be equal to *1.35258e+07* with 3 or 4 digits of accuracy. The other numbers must not be NaN or Inf.

Strong Scaling: Run the benchmark with copies sized at 512, 1024, 2048, 4096 and 8192 cores, respectively.

Multithreading: is allowed

Commitments: The performance per core is computed as:

$$Performance\_per\_core = \frac{1}{Walltime * number\_of\_cores}$$

Fill in the following table:

| Number of MPI tasks | Number of threads | Number of cores | Walltime (LRZ timing value) [sec] | Performance_per_core [1/sec] (provide enough significant digits) |
|---|---|---|---|---|
| | | 512 | | |
| | | 1024 | | |
| | | 2048 | | |

| | | | | |
|---|---|---|---|---|
| | | 4096 | | |
| | | 8192 | | |

The entry marked in red is used for the quantitative evaluation and must be filled in, the other entries are for qualitative evaluation. Please also specify how many copies can run on the Phase 1 installation:

| | |
|---|---|
| Number of copies of the benchmark on the Phase 1 system, executed with 4096 cores per copy | |

### 3.4.4 GENE

The gyrokinetic plasma turbulence code (GENE) is a software package dedicated to solving the nonlinear gyrokinetic equations in a flux-tube domain. Alternatively, it can be operated in a linear mode, thus calculating the properties (like complex frequencies, parallel mode structures, and quasilinear transport coefficients) of the microinstabilities driving the turbulence. GENE can be run on a large number of different computer architectures, including Linux clusters and various massively parallel systems, using anything between a single and tens of thousands of processors. The code itself is written in Fortran90/95, the package also includes an IDL-based tool for data visualisation and analysis.

source: $BENCH/src/apps/GENE

source : ./src
IDL-based postprocessing routines: ./diagnostics
documentation: ./doc
testsuite: ./testsuite

as well as the `newprob` script, a generic `makefile` and several machine-specific makefiles. If you are planning to use GENE on a different architecture, you will have to create a new machine-specific `makefile`.

The external software you need to specify in order to compile GENE are:

Fortran90/95 compiler
MPI message passing interface
FFT routine (ESSL, MKL, FFTW, optional support for NAG libraries).
The BLAS/LAPACK library (also contained in ESSL, MKL, ACML)

Additional software packages that can extend the functionality of GENE are OpenMP and the PETSC/SLEPC packages for eigenvalue computations, available at http://www.mcs.anl.gov/petsc and www.grycap.upv.es/slepc.

In order for the makefile to work properly, choose a name for your machine and rename the directory and makefile:

```
./src/svprob/new_machine/new_machine.mk  to ./src/svprob/mach/mach.mk
```

The code must be compiled with PRECISION=double set in the *.mk file i.e., using the compiler flags **–r8** and the pre-processor flags **–DDOUBLE_PREC**

Some machines are automatically recognized by the values of certain predefined environment variables. If you have written a new machine-specific makefile, you either have to extend these routines (located in the main makefile in the GENE base directory) or to set an environment variable MACHINE to `mach` before you can compile or test your installation.

Testing: Once you have a (preliminary) makefile for your machine, go to the `./testsuite` directory and check whether you can compile with:

```
make -f ../makefile
```

Note that the first call to `make` copies the machine-specific makefile to `./testsuite/mach`, which is then used for subsequent compilations. This local version of the makefile can then be improved and tested before it is finally transferred back to `./src/svprob/mach`.

After you managed to compile without errors, type:

```
./testsuite
```

This will compute test problems with increasing complexity on an increasing number of processors and will compare the results and time needed for the computation with reference values. By default, the maximum number of MPI processes used is 8. OpenMP parallelisation can be tested by setting the OMP_NUM_THREADS environment variable to a value greater than 1.

Setting up a computation: To create a problem folder `./prob01` in which the parameters for a computation are defined, execute: `./newprob` in the GENE base directory. Successive calls to `newprob` will generate new problem folders: `./prob02/, ./prob03/` …etc. The problem folders may be renamed, the directory structure must not be changed, however.

The next step is to adapt the switches for libraries, precision, optimisation etc. in the machine-specific makefile Compile GENE by typing:

```
cd ./prob01
make -f ../makefile
```

This action should create an executable called:

```
prob01/mach/gene
```

which can then be used for both linear and nonlinear runs.

The other file to be adapted is the input file `parameters`. Here, one can specify various physical and numerical parameters, and choose between several options concerning the input and output of data.

Measurements: The only input file is `parameters`. It specifies Fortran namelists.
For a run with nprocs MPI-tasks, copy the file `../input/parameters.[nprocs]` to `parameters`, optionally modify it and run the executable:

```
cp ../../input/parameters.[nprocs]  parameters
$RUN -C <ncopies> -I 1 -N <nodes> -n nprocs -t <threads> ./gene  > output
```

Reference: reference outputs are in the directory `./reference`. **Note that differences in results for relatively small numbers (1.0E-15 and smaller) can be ignored.**

Multithreading: is permitted and may be useful. Please report the number of nodes, processes and threads that are used.

Hints: The number of tasks is determined by nproc_s, nproc_v, nproc_w, nproc_y, nproc_z. The parameters used in the example were chosen to yield the best results on LRZ's Altix 4700. On any other system, different setting might be needed depending on the interconnect topology and hardware.

Output: The following output files stored in the run directory should be delivered:

| | |
|---|---|
| nrg.dat | is used to verify the correctness of the benchmark run |
| parameters.dat | contains the parameters of the benchmark run |
| s_alpha.dat | contains information about the gridpoints |
| output | contains logging information, especially the result of the time measurement. |

Commitment: The required timing information is in the line: `Wall clock time of the time loop`

$$Performance\_per\_core = \frac{1}{Walltime * number\_of\_cores}$$

Fill in the following table:

| Number of MPI tasks | Number of threads | Number of cores | Walltime [sec] | Performance_per_core [1/sec] (provide enough significant digits) |
|---|---|---|---|---|
| | | 1024 | | |
| | | 2048 | | |
| | | 4096 | | |

The entry marked in red is used for the quantitative evaluation and must be filled in, the other entries are for qualitative evaluation. Please also specify how many copies can run on the Phase 1 installation:

| | |
|---|---|
| Number of copies of the benchmark on the Phase 1 system, executed with 4096 cores per copy | |

### 3.4.5 LB-DC

The LB-DC is a benchmark kernel based on a full-featured Lattice-Boltzmann-Solver. This benchmark consists of two complementary programs: a preprocessor and a solver. The preprocessor creates the input data files, which are used by the solver.

Source: $BENCH/src/apps/LB-DC-Kernel

Executable: Prepartitioner: `./Exe/preproc.<name_of_architecture>-mpi`

Solver: `./Exe/vector.<name_of_architecture>-mpi`

Building: Metis 5.0 with 64-bit index integers. Compiler options etc. are provided by architecture or machine specific include files located in the `./Sys` directory.

Exactly one include file has to be activated in the Makefile by uncommenting the appropriate SYSTEM definition.

The preprocessor is built with the command

```
make preproc LIBMETIS=$METIS_LIBDIR/libmetis.a
```

The solver is built by simply running

```
Make
```

Further information on building and running this benchmark can be found in the file `./Documentation/userguide.pdf`.

Multithreading: Multithreading is permitted and may be useful. Please report the number of nodes, processes and threads that are used.

Procedure: The directory `./ParameterSets/` contains several subdirectories `<number_of_processes>` containing config files for runs with the corresponding number of MPI tasks.

Run the preprocessor within the directories1000, 2000, 4000, and 8000 to create the input files for the solver. The preprocessor needs the file `preproc_multi.par` only.

After that, run the solver itself within these directories and extract the performance data from its output. The solver needs the config files `flow.par, output.par,` and `vector.par`, as well as the data files created by the preprocessor.

Command Line: Prepartitioner: **The preprocessor is not parallelized and may take some time to complete.**

```
preproc.<name_of_architecture>-mpi
```

Solver:

```
$RUN –C <copies> -I 1 –N <nodes> -n nprocs –vector.<name_of_architecture>-mpi
```

Testing: Small test cases may be generated by adjusting the lattice size in the file `preproc_multi.par` and the number of iterations in `vector.par`.

Commitments: Results are given in the output file in terms of:

```
million lattice site updates (fluid cells)/ELAPSEDsecond
```

Divide this number by the number of cores used. The wall time in seconds is characterized by the line

```
elapsed-time for main loop
```

For each run, supply the output of the program in the following table:

| Number of MPI Tasks | Number of | Number of Cores | Elapsed-time [s] | million lattice site updates (**fluid cells**)/ELAPSED | per core: million lattice site updates/(Elapsed second* |
|---|---|---|---|---|---|

| | Threads | | | second | number of cores) |
|---|---|---|---|---|---|
| | | 1000 | | | |
| | | 2000 | | | |
| | | 4000 | | | |
| | | 8000 | | | |

Please also specify how many copies can run on the Phase 1 installation:

| | |
|---|---|
| Number of copies of the benchmark on the Phase 1 system, executed with 4000 cores per copy | |

### 3.4.6 NAMD

NAMD is a widely used molecular dynamics application designed to simulate bio-molecular systems on a wide variety of computing platforms. NAMD is developed by the Theoretical and Computational Biophysics Group at the University of Illinois at Urbana Champaign (`www.ks.uiuc.edu/Research/namd/`). In the design of NAMD, particular emphasis has been placed on scalability when utilising a large number of processors. NAMD is noted for its parallel efficiency, and is often used to simulate large systems (millions of atoms).

NAMD automatically adjusts the load balance during the first part of the simulation; the computational load is measured for each patch and patches are moved between the processors if necessary. Most of the code required for the load balancing is part of Charm++ which is used as a component. The performance improvement due to the dynamical loadbalancing is largest when using a very large number of processors. The application itself is written in C++ using Charm++ parallel objects.

The required Charm++ can be built on a wide variety of communication protocols. The source of Charm++ is distributed with the NAMD source. Building a production version of NAMD also requires the libraries TCL, FFTW (version 2.1.5, single precision).

Building: For a large number of platforms, binary executables can be downloaded from the NAMD website. However these will only work if everything required by the executables is installed in the standard place. If these executables do not work on a given system or no binary is supplied for your architecture you need to build NAMD from source. To support this, the NAMD source distribution contains a number of architecture specific files for NAMD and Charm++.

```
cd $BENCH/src/apps/namd
```

first you have to build charm++

```
cd ./charm-6.1
./build charm++ net-linux-x86_64 --no-build-shared -O -DCMK_OPTIMIZE=1
```

now you are ready to build namd

```
./config Linux-x86_64-g++ --charm-arch net-linux-x86_64 --without-tcl \
   --without-fftw
cd ./Linux-x86_64-g++
make
```

finally test the installation:

```
cd $BENCH/src/apps/namd
./namd2 src/alanin
```

Extra Download: The required input files (see below) are provided as a separate download from the URL specified in section 2.1.7

Hints: Running NAMD on many thousands of processors may fail due to the MPI-library running out of resources. This typically happens during the load balancing step, when all processors need to share their performance data with rank 0. In this situation rank 0 quite often runs out of space to buffer the large number of unexpected messages. In this case it may help to set implementation specific environment variables (e.g., MPICH_UNEX_BUFFER_SIZE=100M and MPICH_PTL_SEND_CREDITS=-1) in the job submission script. If the library still runs out of buffer space, one typically receives a clear error message explaining the problem and suggesting the relevant environment settings which need modifying. Quite generally, it may be advisable to provide the rank 0 task (the master) with considerably more memory resources (up to a factor 4) than the other tasks. This can be achieved by using e.g. only one active core on one of the compute nodes used for the benchmark run, which is assigned to the master task.

Source: $BENCH/src/apps/namd

Version: 2.7b1

Executable: namd2

Testing: A test case with about 10,000 atoms (called `water_0.01mio`) is provided; it runs about half a minute on 8 cores.

Input: The actual benchmark input set is `water_6mio`. The input data consists of two files with the extension ".pdb" (coordinates of every atom) and ".psf" (structure file, describing chemical bonds). The simulation run is controlled through an ASCII input file with the extension ".namd". This file has several sections, all indicated by a comment sign ("#" in the first column). You may need to change this input file, if you use other file paths.

Procedure:
```
cd water_6mio
$RUN –C <ncopies> –I 1 –N <nodes> -n <ncores> -t 1 namd2 wa-
ter_6mio_dynamics.namd
```

Output: The required performance can be obtained from the output printed to stdout. This output should contain something similar to this:

```
TIMING: 1000  CPU: 298.363, 0.116917/step  Wall: 298.363, 0.116917/step, 0.032477
hours remaining,  4264608.375000 MB of memory in use.
......
......
TIMING: 2000  CPU: 415.546, 0.117269/step  Wall: 415.546, 0.117269/step, 0 hours
remaining, 4264608.375000 MB of memory in use.
ETITLE:      TS           BOND          ANGLE          DIHED          IMPRP
ELECT
         VDW       BOUNDARY           MISC        KINETIC               TOTAL
TEMP
POTENTIAL         TOTAL3        TEMPAVG

ENERGY:    2000   1391354.0423   1005152.3559         0.0000         0.0000      -
29401649.5532
3444612.4046         0.0000         0.0000   4005373.4237      -19555157.3269
213.0212 -2356
0530.7505 -19485956.8918       212.8747

WRITING EXTENDED SYSTEM TO OUTPUT FILE AT STEP 2000
WRITING COORDINATES TO OUTPUT FILE AT STEP 2000
WRITING VELOCITIES TO OUTPUT FILE AT STEP 2000
=================================================

WallClock: 859.916931  CPUTime: 859.916931  Memory: 4265024.984375 MB
End of program
```

Make sure that the simulation finished correctly ("End of program") and provide the walltime from the timing information printed at time step 1000 and at time step 2000. Look for the line beginning with "TIMING: 1000" that gives the timing information after 1000 time steps and take the value for "Wall" (walltime in seconds). Next, look for the timing information at time step 2000 (line starts with "TIMING: 2000"). Subtract those two values to get the walltime:

*Walltime = [value for Wall at time step 2000] – [value for Wall at time step 1000]*

The performance per core is computed as:

$$Performance\_per\_core = \frac{1}{Walltime * number\_of\_cores}$$

Enter your result in the following table:

| **Number of cores** | **Walltime [sec] (as from output, see above)** | **Performance_per_core [1/sec]** **(provide enough significant digits)** |
|---|---|---|
| 1000 | | |
| 2000 | | |

Please also specify how many copies can run on the Phase 1 installation:

| | |
|---|---|
| Number of copies of the benchmark on the Phase 1 system, executed with 2000 cores per copy | |

### 3.4.7 SEISSOL

SeisSol is a program written in Fortran 90 and uses MPI for parallelisation. It is used for the simulations of realistic earthquake scenarios, accounting for a variety of geophysical processes affecting seismic wave propagation, such as strong material heterogeneities, viscoelastic attenuation and anisotropy.

The accurate numerical simulation of the propagation of seismic waves helps to understand complicated wave phenomena; it includes capabilities to adapt to complex 3D geometries by using of tetrahedral and hexahedral meshes. It also can handle acoustic, elastic, viscoelastic, poroelastic and anisotropic material properties to approximate realistic geological sub-surfaces. The internal algorithms use arbitrarily high-order approximations in time and space as well as an explicit local time stepping algorithm, such that each element runs with its own optimal time step length to reduce computation time.

Source: $BENCH/src/apps/seissol

Executables: Seissolxx

Directory Structure: set $BENCH/src/apps/seissol as your root directory. It is important to respect this order, many files are written and read using this directory hierarchy:

| | |
|---|---|
| bin | contains `Seissolxx` executable |
| cases | Base Directory for test cases |
| cases/EU4 | Input Data and test case* |
| cases/Maple | Auxiliary Maple files* |
| lib | Libraries created by Seissol |
| Reference | Reference for checking correctness of results |
| src | Seissol complete source code |

* Files provided according to section 2.1.7 of this document.

Building SeisSol provides a Makefile, which includes a Makefile.def ; both are found in the

`seissol/src/` subdirectory.

To compile SeisSol it may be sufficient to modify Makefile.def only:

```
F90=mpif90
F77=mpif90
...

MPI_INCL=
MPI_LIB=

EXTRA_INCL=
```

```
EXTRA_LIB=

...
```

All default REALs **must be promoted** to REAL(kind=kind(1.0D0)).
The **-DSTATISTICS** macro must be set to generate the correct benchmarking version.

This is already included in the example configuration Makefile.def:

```
FFLAGS += -fpp -r8 -DSTATISTICS -O3…
```

Extra Download: The contents of the cases subdirectory are provided as a separate download from the URL specified in section 2.1.7

Running: In the `cases/EU4` sub-directory, you will find all the necessary information to run the benchmark case with 3,702,469 cells.

Domain decomposition is provided to be run with 64, 128, 256, 510, 1020 and 2040 MPI tasks. Trying to run with other counts will produce an error. When running use `seissol/cases/EU4` directory as base directory. Many paths are relative to this directory.

For generality, it will only execute 100 time steps.

```
$RUN -C <copies> -I 1 -N <nodes> -n 1020 -t 1 \
      ../../bin/Seissolxx PAR.par >seissol.out
```

see `EU4.batch.pbs` and `EU4.batch.pbs.msub` in `cases/EU4` as examples.

Hints: reading the mesh and other preprocessing steps might take several minutes. SEISSOL will produce 3xNtask files with detailed information.

Multithreading: is probably not useful for this benchmark.

Reference: SeisSol will produce several files, look for a file with this pattern; `eu4miO4-pickpoint-*.dat`. It will generate one and only one of this "pickpoint" file for each run. It is advisable to save this file before performing a new run to avoid overwriting.

Compare your results with `Reference/eu4miO4-pickpoint-00001.dat`. This file contains the evolution of velocity components u,v,w (at times 0.1..0.7) at a certain location "x,y,z".

**All values have to match with at least 6 to 7 digits of accuracy.**

Output: The total time is given in the output file (STDOUT or BATCH output) by searching:

```
LRZ SUPERMUC BENCH: number of Tasks  :       <mpi tasks>
LRZ SUPERMUC BENCH: number of Threads:       <ntreads>
LRZ SUPERMUC BENCH: Walltime [s]     :       1204.89928
```

The performance per core is computed as:

$$Performance\_per\_core = \frac{1}{Walltime * number\_of\_cores}$$

Enter your result on the following table:

| **Number of MPI tasks** | **Number of threads (probably 1)** | **Number of cores** | **Walltime [sec] (as from output, see above)** | **Performance_per_core [1/sec] (provide enough significant digits)** |
|---|---|---|---|---|
| | | 510 | | |
| | | 1020 | | |
| | | 2040 | | |

Please also specify how many copies can run on the Phase 1 installation:

| | |
|---|---|
| Number of copies of the benchmark on the Phase 1 system, executed with 2040 cores per copy | |

### 3.4.8 WALBERLA

Particle-in-fluid dynamics plays an important role in many physical and industrial systems such as fluidisation or sedimentation processes, and a detailed modeling of the transport processes is ongoing research. The waLBerla (widely applicable Lattice Boltzmann from Erlangen) benchmark computes moving particles incorporated in a fluid flow under gravitational forces. For this purpose a lattice Boltzmann fluid solver is two-way coupled to a so-called physics engine, which handles the movement and collision of the objects. With this approach, the objects are not treated as mere point masses, but are fully resolved as individual geometric entities within the flow with an accuracy that is determined by the LBM grid.

Source: $BENCH/src/apps/WALBERLA

Executable: runs/solver

Modifications: This C++ application may be very sensitive to modifications of the underlying code structure.

Edit Makefile.inc to appropriately adjust the compiler settings.

Requirements: Parts of the `Boost` library are encapsulated in `./extern/pe/src/boost`, but all other parts of the application make extensive use of the Boost header files.

Hints: It may be necessary to increase the values of vendor-specific environment variables for controlling the MPI run time environment (see `runs/JOB`).

**Result directories must be empty before running.**

Command Line: An example is contained in `runs/JOB`.

Procedure: Strong scaling

```
for PROCS in 512 1024 2048 4096 8192
do
   # Set MPI Environment variables if needed
   cd $WALBERLA/runs
   rm -rf ${PROCS}_proc.s.results
   mkdir ${PROCS}_proc.s.results
   cd    ${PROCS}_proc.s.results
   mkdir results
   $RUN -C <copies> -I 1 -N <nodes>-n $PROCS -t 1 \
        ../solver./${PROCS}_proc.s.prm 2>&1 | \
          tee ${PROCS}_proc.s.out
done
```

Reference-output: `runs/REFERENCE`

Multi-threading: is not permitted for this benchmark

Results: Deliver the files

`${PROCS}_proc.s.results/${PROCS}_proc.s.out`

The required performance is given in the last lines labelled with:
LRZ SUPERMUC BENCH

| | |
|---|---|
| Output: | Only one of the simultaneously running WALBERLA applications must write the complete povray output. This is because otherwise, several tens of thousands of small files would be created. I.e., for all but one application “spacing” in povray.prm can be set to 1501. The timings must be taken for the instance which writes the output. |

Commitment:

| **Number of MPI tasks** | **Number of threads (probably 1)** | **Number of Cores** | **[GFLOP/s]** | **Performance per Core [GFLOP/s]** |
|---|---|---|---|---|
| | | 512 | | |
| | | 1024 | | |
| | | 2048 | | |
| | | 4096 | | |
| | | 8192 | | |

Please also specify how many copies can run on the Phase 1 installation:

| | |
|---|---|
| Number of copies of the benchmark on the Phase 1 system, executed with 2048 cores per copy | |

## 3.5 Energy efficiency

Because of the high energy prices in Europe and particularly in Germany, only highly energy efficient systems are economically viable in the future. Also considering environmental and ethical aspects, energy efficiency is considered a value itself. Therefore, energy efficiency is separately evaluated.

For the LINPACK benchmark, the power comsumption of the system has to be measured (excluding the disks and not taking into account the power for cooling and air conditioning). The rules for submission to the Green500 list apply (http://www.green500.org/). The energy efficiency is calculated as

$$eff = \frac{R_{max}\ [TFlop/s]}{Power\ \ [MW]}$$

For $R_{max}$ the configuration defined for benchmark 3.3.3 has to be used.

Commitment:

| | |
|---|---|
| LINPACK performance [TFlop/s] | |
| Power [MW] | |
| eff, Power efficiency of the system [TFlop/s / MW] | |

## 3.6 Storage subsystem benchmarks

These benchmarks are included to verify the commitments for I/O performance. Client-to-Server data integrity and replication can be disabled for execution of these benchmarks if desired.

Configuration of the file systems should follow the guidelines given in section 2.5 of the *Description of Goods and Services*.

### 3.6.1 IOBench 1: multi-stream read/write for parallel file system

Purpose: This benchmark tests the I/O performance for the parallel file system.

Source: `$BENCH/io_bench/iobench`

Procedure: The test is performed for Fortran and C by writing to and reading from a globally accessible file system in parallel by multiple processes from different nodes. Files are written and read in chunks of 32 MBytes by default.

The size of each file is 256 GBytes.

For the acceptance test the benchmark will be run with 1000 processes in Phase 1 and 2000 processes in Phase 2.

Every process must run on a separate "thin" node (this means 1000 resp. 2000 nodes will be used).

Modifications: `iobench.dat`: The paths and names of the files which are written to or read from must be specified in `iobench.dat`.

The chunk size can be modified. The number of processes and file size must not be modified by more than 10%.

The insertion of compiler directives is permitted. Other modifications of the remaining routines are not permitted.

Command lines: Run the benchmark (example for Phase 1):

```
mpirun -np 1000 iobench < iobench.dat
```

Results: Deliver the output file `iobench.res`.

A detailed description of the I/O configuration must be given; particularly, all deviations from standard OS and RAID parameters must be fully disclosed.

The results are marked by the following lines:

"Overall Results for C (aggreg. over all procs and all runs)"

The value for write bandwidth is in the line marked "Write".

The value for read bandwidth is in the line marked "Read".

"fRead" and "fWrite" are for informational purposes only.

Reference output: see subdirectory REFERENCE-OUTPUT.

Commitment:

| | |
|---|---|
| I/O bandwidth for writing: | _____ GByte/s (Phase 1) |
| I/O bandwidth for reading: | _____ GByte/s (Phase 1) |

### 3.6.2 IOBench 2: multi-stream read/write for home file system

Purpose: This benchmark tests the conventional I/O performance for the home file system. In this benchmark, each MPI process writes and reads one file.

Source: $BENCH/io_bench/iobench

Procedure: As described in section 3.6.1

For the acceptance test the benchmark will be run with 200 processes in Phase 1 and 300 processes in phase 2.

Every process must run on a separate "thin" node (this means 200 resp. 300 nodes will be used).

Modifications: As described in section 3.6.1

Command lines: Run the benchmark (example for Phase 1):

```
mpirun -np 200 iobench < iobench.dat
```

Results: As described in section 3.6.1

Commitment:

| | |
|---|---|
| I/O bandwidth for writing (Primary) | _____ GByte/s (Phase 1) |
| I/O bandwidth for writing (Secondary) | _____ GByte/s (Phase 1) |
| I/O bandwidth for reading (Primary) | _____ GByte/s (Phase 1) |
| I/O bandwidth for reading (Secondary) | _____ GByte/s (Phase 1) |

### 3.6.3 Metadata Benchmark

Purpose: This benchmark tests the metadata performance for the home file system and the parallel file system. In this benchmark (based on Bonnie++), each MPI process writes files into a separate directory.

Source: $BENCH/io_bench/mpibonnie

Procedure:

The benchmark creates a directory for every MPI process and then create 0-byte files inside the directories.

You have to provide a path where the directories will be created.

For the acceptance test the benchmark will be run with 500 processes in Phase 1 and Phase 2.

Every process must run on a separate "thin" node (that means 500 nodes will be used).

The whole benchmark (creating files, using stat() and deleting files) must be completed successfully but only the file creation performance is evaluated.

Modifications: Distributing directories to different parts of a namespace is allowed.

Command lines: Run the benchmark:

```
mpirun -np 500 mpibonnie -n 2 -d <path to filesystem> -x 1 -s 0 -f > log.out
```

Post-process the results:

```
perl bonniempi_postprocess.pl log.out
```

Results: The result is the number shown in the column "Sequential create"/"Create/sec" in the output of the post.process-script (first column).

Commitment:

| | |
|---|---|
| Home file system | _____ file creations/s (Phase 1) |
| Parallel file system | _____ file creations/s (Phase 1) |

## 3.7 Procedure for the determination of the Power/Energy Capping Limit

This procedure applies only for the determination of the power capping limit. It does not apply for the determination of the aggregate compute performance, which can be performed at different frequency and power settings.

For each of the eight application benchmarks (see 3.4), the system is filled with the particular benchmark in accordance with the rules given in 1.1.1. The benchmarks are repeatedly run for a timespan which is long enough to make reliable power measurements. All devices (storage, networks etc.) of the system must be in normal operation.

The application benchmarks are run with the processor frequency and power envelope settings which are intended by LRZ for default user operation, e.g., 95W per processor.

The average power requirement (unit: kW, Kilowatts) for each application benchmark $L_i$ is determined and from this the average for the eight application benchmarks is formed

$$L = \sum L_i/8$$

The Power Capping Limit PCL is defined as:

$$PCL = 0.9 * L \quad \text{(unit: kW)}$$

For the time interval T to be defined the contract (typically a month or a quarter of a year), LRZ assures that it will not exceed the energy capping limit ECL

$$ECL = PCL * T \quad \text{(unit: kWh)}$$

# 4 Summary of mandatory requirements