eeMark Manual

Daniel Molka

June 20, 2012

SPONSORED BY THE

Federal Ministry
of Education
and Research

Contents

1 Introduction

The energy consumption of compute clusters has become a major concern over the last years. Therefore, methods to efficiently operate HPC centers continue to become more important as the focus is shifting from raw performance towards performance per Watt. The peak power consumption per node is no longer increasing while the idle power continuously decreases due to the adaption of power saving mechanisms. However, the proper use of hardware efficiency features under partial load remains challenging. Idle or low-power states of the processor can reduce performance due to wake-up latencies of the devices, invalidated caches, etc. On the other hand, features such as Turbo Boost of newer generation Intel processors are widely used, even if the performance gain is overshadowed by the non-proportional power consumption increase. This results in a high power consumption of processors during phases of high utilization, that are unfortunately also generated by busy-waiting based synchronization mechanisms. The challenge is to identify the appropriate energy saving mechanism for a specified workload to reduce the power consumption with little or now performance decrease.

Therefore, energy efficiency benchmark `eeMark` was developed in order to analyse the energy efficiency of clusters and investigate the effectiveness of power saving mechanisms. A configurable source code generator is used to create various computationally intensive kernels that can be tailored to different processor architectures. These compute kernels are combined with MPI communication and I/O operations to create parallel workloads that stress all components of an HPC installation. Different phases are typically bottlenecked by few components, allowing to conserve energy in other, not fully utilized components. Multiple power measurement tools are supported to determine the energy efficiency under those well-defined workloads.

Section 2 describes the high-level benchmark design. Section 3 shows how the source code is generated for a certain platform. The usage of the benchmark is explained in Section 5. Section 6 illustrates the results generated by the benchmark.

2 eeMark Overview

The benchmark is designed to determine the performance and efficiency of HPC systems. In order to support multiple instruction set architectures, the benchmark is implemented in the high level programming language C. Furthermore, HPC systems typically are distributed memory systems with multiple compute nodes that are connected via a network. MPI is used as communication and synchronization library between the participating nodes and processes, as it is architecture independent and commonly used in the HPC community.

2.1 Kernels to stress individual system components

Processors, memory, network, and file system are the major contributors to the total energy consumption. Therefore, three classes of benchmarking kernels are available to stress the different components:

- compute kernels that generate high load on CPUs and/or memory,
- communication kernels that stress the network between the nodes,
- I/O kernels that use the file system.

Figure 1 shows the dataflow through the kernels. A detailed description of the kernels can be found in Section 3.2.

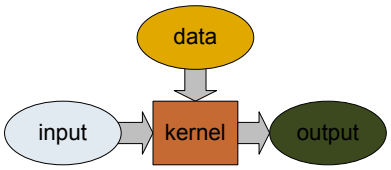

Figure 1: Dataflow of individual kernels. Compute kernels use *input* and *data* to calculate the *output*. Communication and I/O kernels only use *input* and *output* buffers as source and destination, respectively

2.2 Kernel Sequences and Their Parallel Execution

Applications usually have multiple phases during their execution. To reflect that in the benchmark, each MPI rank performs a sequence of kernels that stress different components. Two buffers are allocated per rank that alternatingly act as *input* and *output* buffers for the kernels as depicted in Figure 2.

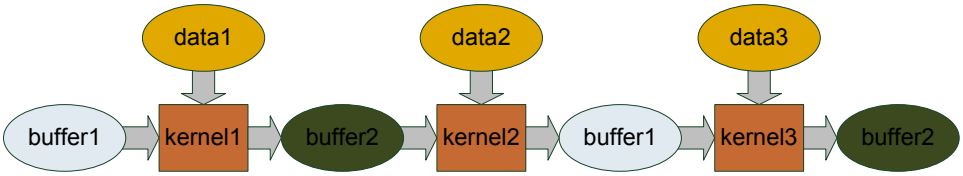

Figure 2: Kernel sequence dataflow. The output of each kernel in the sequence becomes the input of the kernel that follows. The *data* buffer is only allocated with the size required by the respective kernel.

The kernel sequences are defined for groups of ranks that perform the same workload (see Section 3.3). The size of the groups can be calculated dynamically at runtime in order to enable running the benchmark with different numbers of processes However, groups with a fixed size are supported

as well to allow e.g. a single master that collects all data from a variable number of workers. Three algorithms as depicted in Figure 3 are available to assign ranks to groups. Communication can be done between ranks of a single group or between ranks of different groups. The available patterns are listed in Section 3.2.

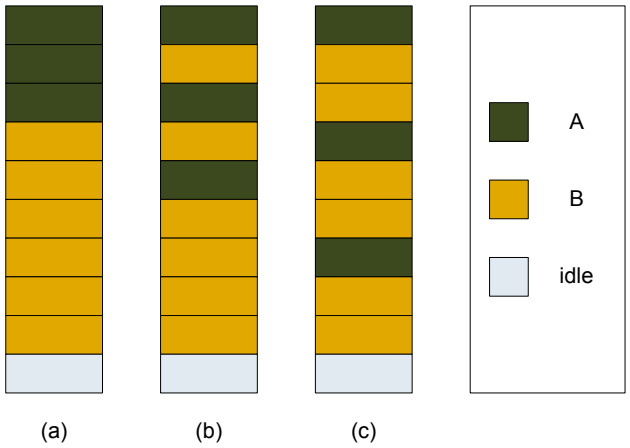

Figure 3: Group assignments of 10 ranks to two groups of variable size with a 1:2 ratio. Groups: A (3 ranks), B (6 ranks), and idle (1 rank). (a) continuous blocks, (b) round robin, (c) weighted round robin

3 Source Code Generation

The eeMark source code is generated from templates. Generating the source code requires `python`, the `python-cheetah` package, and the program `date` to be installed. By default the python script only generates the source code. However, it can be used to build and run the benchmark as well.

```
Usage:
 ./eeMark.py [options]
options:
 -a|--action {source|build|run}:  select operation
    source:      only generate source code (default)
    build:       generate code and compile
    run:         execute benchmark (requires -b)
 -c|--config:    configuration file (default: config.cfg)
 -b|--benchspec: benchspec to run
 -h|--help:      print help
```

3.1 Configuration

A configuration file is used to select the compiler, tailor the benchmark code to the architecture, and select the kernels that are included in the binary. It includes the following options:

- compiler settings:

cc	C compiler to use (e.g. gcc, mpicc)
ccflags	compiler flags (e.g. -mavx, -std=c99)
ldflags	linker flags (e.g. -lmpi)

- MPI settings:

mpicmd	command to run MPI applications (e.g. mpirun) [1]
np_param	parameter to specify number of processes (e.g. -np)
np_default	default value for number of processes

- Source code optimization options:

simd_width	register width in Byte, all buffers will be aligned accordingly
unroll	level of loop unrolling
blocksize	blocksize used by some compute kernels to maximize reuse of level 1 cache content.

- Miscellaneous settings:

prefix	optional prefix for binary name to distinguisch multiple versions that are optimized for different systems
energylib	selected power management library (see Section 4)

Furthermore, additional pragmas can be specified in the section `[pragma]` in the form `"PRAGMA_<NAME>="#pragma <value>"`. This adds the respective pragma before loops if `#include "pre_loop_pragma.tmpl"` is used in the template. Multiple pragmas can be defined. Each prgama can be disabled by removing `-DPRAGMA_<NAME>` from the Makefile before compiling the benchmark.

3.1.1 Kernel Definitions

The available kernels are configured in individual sections. Changes to the kernel definitions are only needed if kernels are added or removed. This is typically not required.

A kernel is defined in a section `[func_<name>]` that can include the following parameters:

[1] must not contain submission command of batch system, see section 5.2

type	type of kernel: compute | convert | comm | io | mpi_io
	compute computation kernel
	complex computation kernel
	convert data conversion kernel
	comm communication kernel
	io POSIX I/O kernel
	mpi_io MPI I/O kernel
tmpl	name of the function template for this kernel (default: func_<name>.tmpl)
init	name of the data initialization template for this kernel (default: init_<name>.tmpl)
datatypes	available data types int32, int64, single, double (default: all)
ops_per_byte	list of available versions with different amount of operations per byte, only needed for compute kernels

3.2 Available Kernels

Only kernels that are listed in the configuration file are included in then source code. Kernels that support the `ops_per_byte` parameter will be generated in multiple versions - one for every listed amount of operations per byte. Compue kernels with type `compute` share one initialization function for all versions, kernels of type `complex` have seperate initialization functions for each version of the kernel. The workload profiles (see section Section 3.3) must not include any undefined kernel or unsupported amount of `ops_per_byte`.

3.2.1 Compute Kernels

The simple kernels (type=compute) perform vector operations using different arithmetic operations (addition, multiplication, multiply-add, division, square root, etc). They can perform a variable amount of operations per byte in order to generate compute bound or memory bound workloads. Many compilers support automatic loop vectorization to utilize SIMD extensions such as SSE or AVX, that increase performance of such data parallel operations. However, only simple loops are covered by a wide range of compilers. To enable vectorization on many systems, the following restrictions are met by the simple compute kernels:

- simple loop form: `for (i=0;i<size;i++)`, not `for (i=0,j=n;i<size;i++,j++)`

- no manual unrolling of loop iterations as auto-vectorization merges multiple loop iterations into one (operations on consecutive data elements within one iteration are not vectorized by some compilers).
- access arrays solely with the loop index, i.e. always use `ptr[i]` (no `ptr[i+n] or j=i+n;...ptr[j]`)

However, modern superscalar processors with pipelining and out-of-order execution typically require a certain amount of independent operations to fully utilize their execution units. A coarse-grained loop unrolling (enabled with the `unroll` parameter in the configuration file) can provide these independent operations without breaking the loop form restrictions. In this case data is divided into multiple blocks with separate pointers. Each loop iteration processes one element of every block, thereby creating independent operations while retaining the possibility for the compiler to combine consecutive iterations into one iteration using SIMD instructions.

A matrix multiplication kernel is available as well (type=complex). Unfortunately, the nested loops and the required more complex data addressing violate above restrictions what can result in low utilization of the SIMD units.

3.2.2 Data Types and Convert Functions

Most kernels are available in 32 and 64 bit integer as well as single and double precision floating point versions. However, the output af a kernel can not be processed by a following kernel that needs a different input data type. Therefore, convert kernels are available that change the data type and enable sequences that use different data types.

3.2.3 Communication Kernels

The communication kernels allow to exchange data between ranks within a group and between ranks in different groups. Available kernels (see Figure 4) are:

(a) global broadcast, reduce, and allreduce that involves all ranks in all groups,

(b) global broadcast, reduce, and allreduce that involve only the first rank of every group,

(c) broadcast, reduce, and allreduce within a group,

(d) send and receive between groups that exchange data between ranks with the same group rank

(e) rotate up and down within a group.

Care has to be taken that all groups participate in global operations (a, b), and that inter-group communication (d) consists of deadlock-free matching pairs of `sends` and `receives`, as otherwise the execution would be blocked.

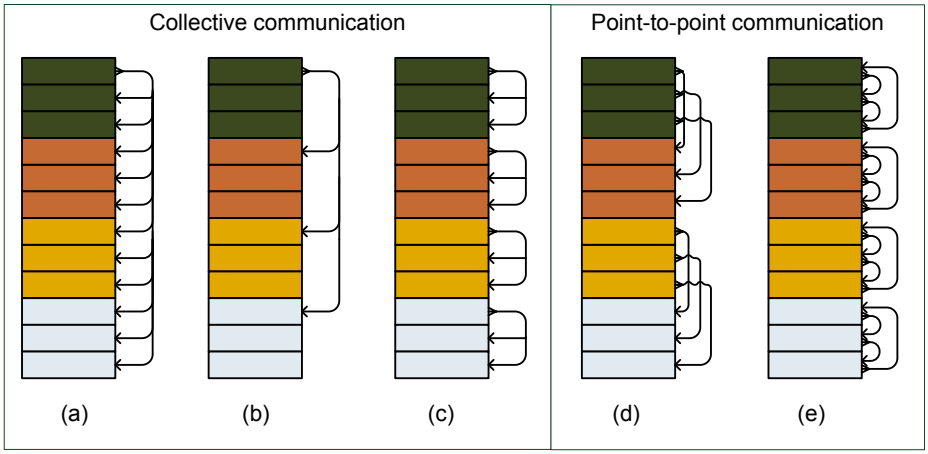

Figure 4: Available MPI communication patterns. (a) all ranks, (b) first rank of every group, (c) within group, (d) between equal group ranks, (e) rotate within group

3.2.4 I/O Kernels

The I/O kernels use POSIX or MPI I/O functions to read and write files. When using POSIX I/O Each rank has individual input and output files. The MPI I/O kernels use one file per group. The input files for the ranks that perform read operations are created and initialized with random data in the initialization phase before the actual benchmark.

3.3 Workload Profiles

The workload is specified in benchspec files (<name>.benchspec). They determine how much computation is performed and how much data is used by a benchmark run.

3.3.1 Syntax

In section `[general]` global settings are defined.

datasize amount of input and output data

computesize amount of data processed by compute kernels, has to be equal to or a multiple of `dtasize`, processing continues at

the beginning of the buffer when the end of the buffer is reached

granularity amount of data processed by a single call of a kernel, the whole sequence is performed for each block before starting to process the next block, allows to share results for parts of the data with other groups

The kernel sequences for each group (see Section 2) are defined in individual `[GroupN]` sections. Each section includes the following parameters:

size fixed|dynamic: determines if the group has a fixed amount of ranks or uses as many ranks as available

num_ranks for groups with fixed size this is the exact number of ranks, for dynamic groups ranks are assigned to the group in multiples of the number, the ratio between different groups with dynamic size is kept during the assignment of ranks (e.g. if one group has size 2 and another size 3 up to 4 ranks can be idle as only multiples of 5 can be assigned to the two groups in order to keep the 2:3 ratio)

function sequence of kernels performed by the group in each iteration, sequences are defined as comma separated lists of functions in the form: <function name>_datatype[_ops_per_byte] (e.g. random_gen_int64, to_double_int64, mul_double_128, io_write_double)

3.3.2 Restrictions

The following restrictions have to be met by each sequence:

- output data type of each function equals the input data type of the following function
- first kernel does not need input, i.e. generate random numbers, read from file system, or receiving MPI function
- last kernel is check_output of matching data type that checks for illegal values (not a number, infinite, or zero)

4 Energy Measurement

Power is measured using external tools. The following methods are supported by eeMark:

PowerTracer is developed at the University of Hamburg. It can be used to store the measured power consumption and corresponding timestamps in a database for post-processing which is used

by eeMark to collect power consumption information after the benchmark is finished.

Dataheap (http://tu-dresden.de/zih/dataheap) is a distributed monitoring and data collection infrastructure developed at Technische Universität Dresden. It consists of a central manager service which is connected to data sources and data collectors via network. Power meters can be connected as data sources. eeMark can be connected as collector in order to access the recorded power consumption values.

PTDaemon (ptd), the SPEC power and temperature daemon has been developed by the SPEC open system group. It supports many power analyzers from several vendors and provides a simple interface to access this information.

All methods use the same basic approach. The power consumption is monitored during the execution and the average power during the measurement interval can be queried using an API after the execution. The required timestamps are collected by rank 0 for each iteration. One of the power measurement tools needs to be available in order to run eeMark. Wrapper libraries that implement the actual power measurement using one of those tools are included in eeMark (subdirectory `lib`). Support for other power measurement equipment can be added by providing a library that implements the API defined in the `src/energymeter_interface.h`. The library to use is selected in the configuration file, `energylib=`

powertracer Use PowerTracer: The libglib2.0-dev package is requiered to use PowerTracer. The `INCLUDE` and `LIB` path of PowerTracer have to be configured in `lib/powertracer_compile.sh` before building eeMark. The used nodes are specified in environment variable: PTLIB_NODES.

dataheap Use Dataheap: Copy libexpec/ folder from your Dataheap installation to eeMark/lib/ directory. Set DHLIB_SERVER to the dataheap server (host:port) and DHLIB_COUNTER to the conter of the power meter connected to the system under test.

specpower Use PTDaemon from SPEC: Set SPLIB_SERVER to the server (host:port) running `ptd`.

dummylib Run eeMark without power measurement.

5 Using eeMark

5.1 Preparation

The following steps are needed to get the benchmark operational:

1) setup one of the power measurement tools (see Section 4)
2) edit power measurement, compiler, and MPI settings in configuration file (default: config.cfg)
3) optionally edit optimization parameters as well
4) generate source code (see Section 3)
5) compile the benchmark on the system under test using `make [-j n]`
6) make sure that the command line tool `bc` is installed, furthermore `wc`, `cut`, `sed`, `echo`, `expr` will be requiered by shell scripts.

Steps 2,3, and 4 can be performed on another computer if the requiered software is not available on the system under test. A script (run.sh) that can be used to start the benchmark is created in step 4 as well. However, manual adaption can be necessary for using batch systems.

5.2 Running the Benchmark

Benchmarks can be run individually or in predefined sets:

- `./run.sh [options] <benchspec>` runs a single workload profile.
- `./runset.sh [options] <benchset>` starts a set of benchspecs.

Options for run.sh:

-i|--iterations	number of iterations
-d|--distribution	distribution of ranks (see Figure 3)
-h|--hugetlbfs	use hugepages mounted at specified directory for `input`, `output`, and `data` buffers.
--version	print version information
--help	print help
<**benchspec**>	benchmark specification file

The `runset.sh` script uses the list of `benchspecs` from the `benchset` file to sequentially start individual benchmarks using `run.sh`. The corresponding parameters for `--iterations` and `--distribution` are taken from the `benchset` file as well. Additional options (e.g. -hugetlbfs) will be passed to all calls of `run.sh`.

The `runset.sh` directly uses the output of the individual benchmark runs. Thus, if a batch system is used the execution of runset.sh has to be submitted as a single job, i.e. the `mpicmd` setting in the config file must not include the submission command of the batch system.

A collection of `benchspecs` and reference `benchsets` can be found in the <`eeMark-inst-dir`>`/benchspec` directory. However, custom `benchspecs` and `benchsets` can be created easily.

6 Benchmark Result

The benchmark determines performance and efficiency ratings as well as a combined score. Detailed reports are generated for individual workload profiles (`benchspecs`). This is especially useful to compare different power management configurations on a single system under the same workload. However, comparing different systems requires to take multiple scenareos into account. Each system can have its strength and weaknesses in different components (e.g. capacity vs. capability systems). Thus, a workload can for example be limited by the network on one system while being compute bound on another. This easily leads to wrong conclusions when attempting to compare the efficiency of different systems based on a single workload. Therefore, `reference benchsets` are provided that cover a wide range of workloads with various utilization levels of the different components in order to facilitate comparisons between system for different classes of applications.

6.1 Individual benchmark results

The performance and efficiency rating are based on the amount of operations performed by the benchmark. As this is not known at compile time the information is collected from all ranks at runtime. Therefore, all kernels report the number of performed operations in ten categories:

iops_32	32 Bit integer operations
iops_64	64 Bit integer operations
flops_sp	single precision floating point ops
flops_dp	double precision floating point ops
mem_read	bytes read from memory
mem_write	bytes written to memory
io_read	bytes read from file system
io_write	bytes written to file system
network_send	bytes send over network
network_recv	bytes received over network

In order to derive a measure for the whole workload operations are weighted according to their effort. The workload heaviness is the sum of all weighted operations of all ranks.

The performance rating is measured in bilion weighted operations per second. The efficiency rating is measured in weighted operations per watt. Additionally, a combined score is calculated, which is the square root of the product of the two ratings. The report generated for each benchmark run includes a summary of the performed operation, runtimes and energy

of every iteration, the average runtime and energy of all iterations, and the three ratings for every iteration as well as the total ratings derived from the average runtimes and energy.

6.1.1 Example:

```
Benchspec: combined1_dp.benchspec
 Operations per iteration:
  - 32 Bit integer operations:            0.00B (weighted        0.00B)
  - 64 Bit integer operations:            0.00B (weighted        0.00B)
  - single precision operations:          0.00B (weighted        0.00B)
  - double precision operations:     185542.59B (weighted   742170.35B)
  - Bytes read from memory/cache:     74217.03B (weighted   445302.21B)
  - Bytes written to memory/cache:    37417.76B (weighted   224506.53B)
  - Bytes read from file system:        309.24B (weighted    59373.63B)
  - Bytes written to file system:       309.24B (weighted    59373.63B)
  - Bytes send to other ranks:         7421.70B (weighted   178120.88B)
  - Bytes received form other ranks:   7421.70B (weighted   178120.88B)
 Workload heaviness: 1886968.112 billion weighted operations
Benchmark started: Sat Apr 28 02:39:37 2012
Iteration 1:
  start:      02:40:02.786157
  end:        03:07:53.584621
  runtime:      1670.798 s
  energy:       4381.198 kJ
  avg. power:   2622.218 W
  Results:
  - performance score:   1129.38
  - efficiency score:       0.43
  - combined score:        22.05
Benchmark finished: Sat Apr 28 03:07:53 2012
 average runtime:   1670.798 s
 average energy:    4381.198 kJ
 average power:     2622.218 W
 Results:
 - performance score:   1129.38
 - efficiency score:       0.43
 - combined score:        22.05
```

6.2 Reference benchset result

Running a `benchset` creates a summary with the individual benchmark results and total scores. The `reference.benchset` evaluates the efficiency for different classes of applications:

- compute benchmarks: These benchmarks perform no or little I/O operations and do not communicate between ranks. The three workloads have different ratios of arithmetic operations and memory accesses. The `compute1` benchspec performs many arithmetic operations with every opperand, thus it depends on high utilization of the

ALUs or FPUs to achieve high performance. The `compute2` benchspec performs only few calculations on every operand, thus it benefits from high memory bandwidth. The `compute3` benchspec uses a mix of computationally intensive and memory-bound kernels. Each workload is performed for different data types.

- communication benchmarks: These benchmarks perform no or little I/O operations. Data is exchanged frequently between ranks which perform only few calculations on the data. The `comm1` benchspec uses two equally sized groups of ranks that bidirectionally exchange data with MPI_send() and MPI_recv(). The `comm2` benchspec has a single master rank that distributes and collects data using MPI_bcast() and MPI_reduce(), respectively. The `comm3` benchspec is a producer-consumer scenario with 3 consumers for every producer. The measurements are repeated for different distributions of ranks where it makes sense.

- I/O benchmarks: Those benchmarks perform mainly I/O operations on large files. The `io1` benchspec reads files, performs a single arithmetic operation on every element, and writes the result into another file. The `io2` benchspec only reads from files and tests every element for abnormal values (nan or infinite). The `io3` benchspec writes random data to files. Each I/O benchspec is available in an MPI I/O (reference.benchset) and an POSIX I/O (reference_nompiio.benchset) variant. The MPI I/O variants use large files that are shared between all ranks in a group, whereas the POSIX I/O versions use smaller files for every rank.

- combined workloads: The combined workloads use a mix of operations from the above three categories. The `combined1` benchspec uses MPI_send() and MPI_recv() to exchange data between ranks, whereas the `combined2` benchspec uses collective MPI operations. Each workload is performed for different data types. The measurements are repeated for different distributions of ranks where it makes sense.

The final benchmark scores are the average of all executed `benchspecs` in the `benchset`. However, the total result is influenced by each and every component in the system. Thus, in order to compare different systems the individual results need to be taken into account to identify characteristic strength and weaknesses. For comparisons between systems of different size it has to be considered that the performance score scales well with system size as `eeMark` uses an weak scaling approach. However, as power consumption scales with system size as well, the efficiency score does not scale with the size of a systems. Furthermore, communication and I/O kernels will not scale linearly because of contention in the network and I/O system. Thus, the efficiency is likely to decrease if system size is increased.

6.2.1 Example:

The following results were measured on four compute nodes, each with four 16-core AMD Opteron 6274 processors (i.e. 256 processes):

Benchmark	Distribution	Iterations	Performance Score	Efficiency Score	Combined Score
compute1_dp	compact	3	2214.67	0.97	46.34
compute2_dp	compact	3	2264.79	0.74	40.95
compute3_dp	fine	3	2384.87	0.94	47.28
compute1_sp	compact	3	1125.11	0.50	23.76
compute2_sp	compact	3	2253.71	0.73	40.60
compute3_sp	fine	3	1334.56	0.56	27.24
compute1_int	compact	3	611.06	0.30	13.46
compute2_int	compact	3	2203.31	0.71	39.48
compute3_int	fine	3	834.24	0.38	17.74
comm1	fine	3	5429.13	1.70	96.13
comm1	compact	3	561.47	0.22	11.20
comm2	compact	3	878.34	0.33	16.96
comm3	fine	3	715.09	0.28	14.18
comm3	compact	3	177.12	0.07	3.58
comm3	roundrobin	3	438.70	0.18	8.81
io1_nompiio	compact	3	403.48	0.27	10.49
io2_nompiio	compact	3	298.44	0.20	7.77
io3_nompiio	compact	3	362.32	0.25	9.49
combined1_dp	fine	3	3996.43	1.33	72.89
combined1_dp	compact	3	1108.31	0.43	21.73
combined2_dp	compact	3	429.24	0.17	8.63
combined1_sp	fine	3	5158.59	1.65	92.24
combined1_sp	compact	3	1108.52	0.42	21.70
combined2_sp	compact	3	343.20	0.14	6.92
combined1_int	fine	3	1293.15	0.47	24.70
combined1_int	compact	3	864.61	0.33	16.95
combined2_int	compact	3	241.75	0.10	4.92
Result:			1445.71	0.53	27.63

The scores in the communication benchmarks `comm1` and `comm3` depend heavily on the distribution of the processes on the nodes as this influences the ratio between network traffic and intra-node communication. Another interesting observation is the difference between `compute1_dp` and `compute1_sp`. This is caused by a limitation of the used compiler that does not vectorize large loops automatically (e.g. addition with 128 operations per element in inner loop is vectorized, but inner loop with 256 operations is not). Thus, scalar instructions are often used in this case instead of packed SIMD instructions, what requires twice as many instructions for single precision.

7 Acknowledgement

The development of `eeMark` has been funded by the german federal ministry of eductaion and research (BMBF) in the project `eeClust` (grant number: 01IH08008C).