Flexible Workload Generation for HPC Cluster Efficiency Benchmarking

07.09.2011

Daniel Molka (daniel.molka@tu-dresden.de)
Daniel Hackenberg (daniel.hackenberg@tu-dresden.de)
Robert Schöne (robert.schoene@tu-dresden.de)
Timo Minartz (timo.minartz@informatik.uni-hamburg.de)
Wolfgang E. Nagel (wolfgang.nagel@tu-dresden.de)
Motivation

- Varying power consumption of HPC systems
  - Depends on changing utilization of components over time (processors, memory, network, and storage)
  - Applications typically do not use all components to their capacity
  - Potential to conserve energy in underutilized components (DVFS, reduce link speed in network, etc.)
  - But power management can decrease performance

- HPC tailored energy efficiency benchmark needed
  - Evaluate power management effectiveness for different degrees of capacity utilization
  - Compare different systems
eeMark – Energy Efficiency Benchmark

- Requirements
- Benchmark Design
  - Process groups and kernel sequences
  - Power measurement and reported result
- Kernel Design
  - compute kernels
  - I/O kernels
  - MPI kernels
- Initial results
- Summary
Requirements

- Kernels that utilize different components
- Arbitrary combinations of kernels
- Adjustable frequency of load changes
- Usage of message passing
- Parallel I/O
- Reusable profiles that scale with system size
Benchmark Design - Kernels

3 types of kernels
- Compute - create load on processors and memory
- Communication - put pressure on network
- I/O - stress storage system

Same basic composition for all types of kernels
- Three buffers available to each function
- No guarantees about input other than
  - Data has the correct data type
  - No nan, zero, or infinite values
- Kernel ensures that output satisfies these requirements as well
  - Buffer data initialized in a way that nan, zero, or infinite do not occur
Benchmark Design - Kernel Sequences

- 2 buffers per MPI process used as input and output
  - Output becomes input of next kernel
- Data buffer per kernel

- Input and output used for communication and I/O as well
  - send(input), write(input): - send or store results
  - receive(output), read(output): - get input for next kernel
Profiles

Define kernel sequences for groups of processes

- Groups with dynamic size adopt to system size
  - E.g. half the available processes act as producers, the other half as consumers
  - Different group sizes possible
  - Multiple distribution patterns

- Groups with fixed amount of processes for special purposes
  - E.g. a single master that distributes work

Define the amount of data processed per kernel

Define block size processed by every call of kernel
Example Profile

[general]
iterations= 3
size= 64M
granularity= 2M
distribution= fine

[Group0]
size= fixed
num_ranks= 1
function= mpi_io_read_double, mpi_global_bcast_double-Group0,
        mpi_global_reduce_double-Group0, mpi_io_write_double

[Group1]
size= dynamic
num_ranks= 1
function= mpi_global_bcast_double-Group0, scale_double_16,
        mpi_global_reduce_double-Group0
Power Measurement

- No direct communication with power meters
- Use of existing measurement systems
  - Dataheap, developed at TU Dresden
  - PowerTracer, developed at University of Hamburg
  - SPEC power and temperature demon (ptd)
- Power consumption recorded at runtime
  - API to collect data at end of benchmark
- Multiple power meters can be used to evaluate large systems
Benchmark Result

Kernels return type and amount of performed operations

- workload heaviness = weighted amount of operations
  - Bytes accessed in memory: factor 1
  - Bytes MPI communication: factor 2
  - I/O Bytes: factor 2
  - Int32 and single ops: factor 4
  - Int64 and double ops: factor 8

Performance Score = workload heaviness / runtime

- billion weighted operations per second

Efficiency Score = workload heaviness / energy

- billion weighted operations per Joule

Combined Score = sqrt(perf_score*eff_score)
Example Result file:

Benchspec: example.benchspec
Operations per iteration:
- single precision floating point operations: 1610612736
- double precision floating point operations: 5737807872
- Bytes read from memory/cache: 33822867456
- Bytes written to memory/cache: 18522046464
- Bytes read from files: 805306368

Workload heaviness: 106.300 billion weighted operations


[...] (runtime and score of iterations)

Benchmark finished: Fri Jun 24 10:44:00 2011
average runtime: 2.188 s
average energy: 492.363 J
total runtime: 10.941 s
total energy: 2461.815 J

Results:
- performance score: 48.58
- efficiency score: 0.22
- combined score: 3.24
eeMark – Energy Efficiency Benchmark

- Requirements
- Benchmark Design
  - Process groups and kernel sequences
  - Power measurement and reported result
- Kernel Design
  - compute kernels
  - MPI kernels
  - I/O kernels
- Initial results
- Summary and Outlook
Kernel Design - Compute Kernels

- Perform arithmetic operations on vectors
  - Double and single precision floating point
  - 32 and 64 Bit integer

- Written in C for easy portability
  - No architecture specific code (e.g. SSE or AVX intrinsics)
  - Usage of SIMD units depends on autovectorization by compiler

- Adjustable ratio between arithmetic operations and data transfers
  - Compute bound and memory bound versions of same kernel
Source Code Generation

Source code created with python based generator

config file

- Compiler options
- Source code optimizations
  - Block size used by kernels to optimize L1 reuse
  - Alignment of buffers
  - Usage of `restrict` keyword
  - Additional pragmas
- Lists of available functions and respective templates
  - Few templates for numerous functions
int work_mul_double_1 (void * input, void * output, void * data, uint64_t size) {
    int i, j;
    uint64_t count = (size / sizeof(double)) / 2048;
    double * RSTR src1_0 = (double *)input + 0;
    double * RSTR src2_0 = (double *)data + 0;
    double * RSTR dest_0 = (double *)output + 0;
    double * RSTR src1_1 = (double *)input + 512;
    double * RSTR src2_1 = (double *)data + 512;
    double * RSTR dest_1 = (double *)output + 512;
    double * RSTR src1_2 = (double *)input + 1024;
    double * RSTR src2_2 = (double *)data + 1024;
    double * RSTR dest_2 = (double *)output + 1024;
    double * RSTR src1_3 = (double *)input + 1536;
    double * RSTR src2_3 = (double *)data + 1536;
    double * RSTR dest_3 = (double *)output + 1536;

    for(i=0; i<count; i++){
        for(j=0; j<512; j++){
            dest_0[j] = src1_0[j] * src2_0[j];
            dest_1[j] = src1_1[j] * src2_1[j];
            dest_2[j] = src1_2[j] * src2_2[j];
            dest_3[j] = src1_3[j] * src2_3[j];
        }
        src1_0+=2048;
        src2_0+=2048;
        dest_0+=2048;
        src1_1+=2048;
        src2_1+=2048;
        dest_1+=2048;
        src1_2+=2048;
        src2_2+=2048;
        dest_2+=2048;
        src1_3+=2048;
        src2_3+=2048;
        dest_3+=2048;
    }
    return 0;
}

Source Code Example

Simple loop form
(i=0;i<n;i++)

No calculation within array index

Coarse grained loop unrolling to provide independent operations
Kernel Design - Communication and I/O Kernels

• MPI kernels
  – bcast/reduce involving all ranks
  – bcast/reduce involving one rank per group
  – bcast/reduce within a group
  – send/receive between groups
  – rotate within a group

• I/O kernels
  – POSIX I/O with one file per process
  – MPI I/O in with one file per group of processes
Producer Consumer Example

- Unbalanced workload
  - Consumers wait in MPI_Barrier
  - Higher power consumption during MPI_Barrier than in active periods of consumers
Process 0 collects data from workers and writes to file

- Usually overlapping I/O and calculation
- Stalls if file system buffer needs to be flushed to disk
Frequency Scaling with pe-Governor

- Process 0-5 compute bound: highest frequency
- Process 6-11 memory bound: lowest frequency
  - High frequency during MPI functions
Compute bound and memory bound phases in all processes

Frequency dynamically adjusted by pe-Governor
# Frequency Scaling Governor Comparison

<table>
<thead>
<tr>
<th>Workload</th>
<th>ondemand governor</th>
<th>pe-Governor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>runtime [ms]</td>
<td>energy [J]</td>
</tr>
<tr>
<td>All ranks compute bound</td>
<td>4911</td>
<td>1195</td>
</tr>
<tr>
<td>All ranks memory bound</td>
<td>4896</td>
<td>1299</td>
</tr>
<tr>
<td>Compute bound and memory bound group</td>
<td>4939</td>
<td>1267</td>
</tr>
<tr>
<td>Each rank with compute and memory bound kernels</td>
<td>4856</td>
<td>1273</td>
</tr>
</tbody>
</table>

- **pe-Governor** decides based on performance counters
  - Significant savings possible for memory bound applications
  - Overhead can increase runtime and energy requirements
Summary

- Flexible workload
  - Stresses different components in HPC systems
  - Scales with system size

- Architecture independent
  - Implemented in C
  - Uses only standard interfaces (MPI, POSIX)
  - Simple code that enables vectorization by compilers

- Report with performance and efficiency rating
  - Evaluate effectiveness of power management
  - Compare different systems
Thank you

Further Information at eeClust homepage

- www.eeClust.de