Performance Characterization and Benchmarking for High Performance Systems and Applications

Erich Strohmaier
NERSC/LBNL
Estrohmaier@lbl.gov
To evaluate and compare application and system performances we need a frame of reference in the performance space.

Right now only peak performance and Linpack are widely used.

A reference can be established by a set of benchmarks.

Users should be able to relate the performance of these benchmarks to their codes.

To develop such benchmarks we first need a better understanding what the critical performance aspects of algorithms are.
General Approach

- Develop a new quantitative characterization of algorithms and codes focusing on performance aspects.
- Avoid using any specific hardware models or concepts for this characterization.
- Develop synthetic performance probes and benchmarks testing these characteristics.
- Relate benchmark performance with code performance.
- Our focus is initially the performance influence of global data-access.
Design Ideas

Performance Characterization:

- Hardware independent.
- Global data access as main focus.
- Random data access as starting point.

Benchmark probe:

- Reference implementations together with a pencil and paper description.
- Runtimes not tied to computational complexities of specific algorithms.
- System and generation scalable.
- Focus on sustainable rates using substantial fractions of available resources.
Characterize performance behavior of applications and algorithms independent from hardware.

- Use most general architecture model possible.

Based on von Neumann model we assume that the effects of data access and instruction stream are independent (first order approximation)

“Time to solution =
  f(Algorithmic Complexity) ‘*’
  ( f(Data Access Characteristics),
    ‘+’f(Structure of Operations) )”
Concepts for Performance Ch.

Code complexities:
- Computational complexity.
- Data access complexity.

Instruction stream:
- Computational granularity.
  - Ratio of instructions to data accesses.
- Length of basic instruction blocks.
  - Between branches.
- Number of “global” operations.
  - Coupling parallel instructions streams.
- Length of local instruction blocks.
  - Between global operations.
Data Access Characteristics

Data access pattern: What do we want to capture?

- **Re-use** of data by modern algorithm for improving locality – *Temporal locality*.
  - Hierarchical block-structured or recursive algorithms.
  - Hard to define hardware independent.
- **Limitations of “vector”-length** – *Granularity*.
  - Due to data-dependencies, communication, etc.
  - Becomes particularly important in parallel context.
- **Regular contiguous memory access** – *Regularity*.
  - stride 1.
  - Data-structures etc.
How can we *quantitatively* describe data re-use?

Look at temporal distribution function:

The probability distribution of how long ago I last used a data item.

At every access I have a $f(t)\%$ probability to hit a location I have visited within the last $t$ cycles.

Cumulative temporal Distribution

Temporal distance is similar to reuse distance, stack distribution, stack distance).
Re-use Number

Define a “re-use” number:

- $M$ be the used memory in words.
- The re-use of a specific word is the number $k$ of accesses to it during a window of $M$ successive data accesses.
- The average re-use for the code is the average $k$ during this window for all accessed words.

(This assumes that all windows give me the same answer)

- The probability at a temporal distance of $M$ is then:

\[ P(M) = \frac{k-1}{k} \]
Temporal Distribution

Approximate the temporal distribution function of codes by a simple generic function.

We try to capture the ‘main’ re-use effect by using a generic function with only a few numeric parameters.

For recursive algorithms the cumulative temporal distribution function should be self-similar and scale-invariant. (A recursive algorithm is self-similar.)

Power Function Distribution
Power Distribution

- Characterized by one number.
- Slope in log-log related to the ‘Re-use’ factor.
- Concept does not use hardware concepts such as ‘cache’
- Distribution function is problem size and scale invariant.

Cumulative temporal Distribution
All we need now is a synthetic pseudo-random algorithm which has a power distribution as temporal distribution function.

Many algorithms generate the same temporal distribution, so we have some choices.

The details of the chosen algorithm could produce artifacts if not selected carefully.

In particular the temporal distribution function is independent of the selected data mapping!

Still (almost) any regularity possible!
Granularity

Limitation of “vector”-length due to data-dependencies.

- The amount of “pre-computable” addresses.
  - Access can be irregular (‘indirect’) or
  - Regular (‘strided’).
  - Limits the amount of dynamic reordering such as
    gather-scatter or message assembly.

- We focus on indirect as it becomes more important
  and represent more of a lower-bound for achievable
  performance.

- Granularity becomes very important for parallel
  version with explicit communication.
  - It (severely) limits message sizes.
A mapping of the data structure to the address space which permits stride 1 access exposes regularity.

Re-mapping during execution might be necessary for many algorithms to expose regularity.

This form of ‘dynamic’ regularity has associated re-mapping costs (gather-scatter operations).

This type of (“irregular”) data access becomes more and more important and is usually not avoidable.

If irregular data access is present in a code it is likely to become the performance bottleneck (Amdahl’s Law).

Irregular data access is “our focus”.
Synthetic Benchmark Probe

- Measures sustainable rates.
  - Warm caches etc.
- Non-uniform random memory access for re-use.
  - Power-function as temporal distribution function.
  - Use indexed (“irregular”) data access to measure a lower bound for performance.
- Granularity
  - Vector length for pre-computed addresses and organization of communication.
- Regularity for simulating data structures.
- We have (only) 3 parameters so far (Small enough?).
Went through a few iterations with the concept.

Still have not figured out the details of the non-uniform random distribution necessary to generate a power function as temporal distribution (math problem).

Are 3 parameters too many already?

Extending the concept to parallel systems.

Details of the random process – homogeneous or inhomogeneous memory-access?
(Do we access all words the same number or do we allow different access numbers?)

Detail of data-mapping – organized or pseudo-random?
(Do we group frequent accessed words together?)
Implemented several (sequential) test-codes.
- Which kernel – DAXPY (again)?
- How many different index vectors?
  - Impacts also data structures and regularity.
for (i = 0+off; i < IdxSize+off+0; i+=8) {
    tmp  += data[ind[i]];
    tmp1 *= data[ind[i+1]];
    ...
    ...
}
Test Results – IBM Power3

R=1; no re-use (k=1)

![Graph showing time (cycles) vs. M=G (4B words)]
Current Kernel

Distribution: \(\text{power}(\text{random}(), 1/A) \times (N/R - 1)\);

if \((R == 1)\) {
  for \((j = 0; j < G; j++)\) {
    res\[j\] += weight\[j\] * data[ind\[j\]];
  }
}

else {
  for \((j = 0; j < G/R; j++)\) {
    pos = ind\[j\] \times R;
    for \((k = 0; k < R; k++)\) {
      res\[j\] += weight\[j \times R + k\] \times data[pos + k];
    }
  }
}
Test Results – IBM Power3

R=1; 64 MWord (8B)
Test Results – IBM Power3

G=1024; 64 MWord (8B)

(time [ns])

R

Legend:
- 1
- 0.3
- 0.1
- 0.03
- 0.01
Future

Finish concept and benchmarking probe (parallel).
Determine the re-use factors and granularities for actual codes (with paper and pencil) for making some meaningful choices.
‘Fix’ some values for parameters to be used as “The Benchmark”.
Need to test the correlation between benchmark probe performance and code performance for the same re-use factors, granularities, and regularities.