Parallel Scaling Properties from a Basic Block View

Melanie Kambadur, Kui Tang, Joshua Lopez, Martha A. Kim

The goal:

Narrow down the source of problematic parallel program scaling.

Micro-scaling metrics expose a broad range of performance flaws by reporting the quantitative and qualitative scaling properties of a program's basic blocks. The metrics

capture the effects of hardware, operating systems, thread models, and inputs,

and

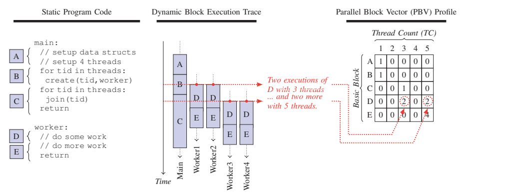
reveal performance issues about arbitrary regions of interest, including individual basic blocks, functions, algorithms, or critical sections.

These software-centric metrics

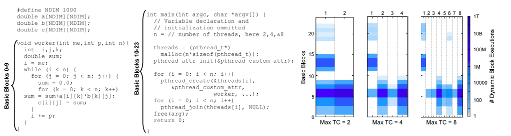
can be collected at runtime with minimal programmer effort, require no hardware modifications, and are not language or thread library specific.

From Parallel Block Vectors to Micro-Scaling Metrics

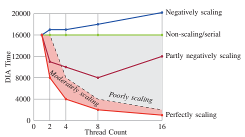
1 **Start with the existing parallel block vector (PBV) profile.** PBVs [1] measure application-level parallelism each time a basic block is executed. We use these profiles as a first step in collecting micro-scaling metrics. The figure below shows a PBV for a pseudo program with five basic blocks, BB:A-E. In the center, the master thread and four worker threads execute the blocks over time. At right, a PBV profile counts the dynamic executions of each static block at each thread count.



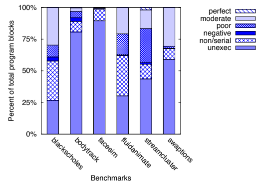
2 **Collect PBV snapshots at different thread counts.** The second step is to collect PBV profiles while running the application at different maximum thread counts. This can be done at runtime using a tool such as the open source Harmony [2]. Profiles at multiple thread counts reveal per-block scaling effects, as for the matrix multiply program below.



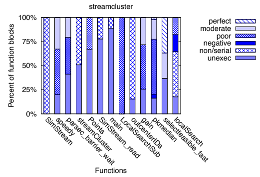
3 **Rank blocks by scaling properties.** The final step is to use a numeric scaling metric to enable a full ranking of blocks from least to most scalable. This metric is simply the change in estimated runtime (see final panel) of a block over its increasing number of maximum available thread counts, or the slope if a block's DIA-time is plotted against maximum thread count as in the figure at right. Along this spectrum, blocks can also be divided into named scaling categories as depicted.



Isolating scaling deficiencies in benchmarks



Micro-scaling metrics at the application level show coverage of scaling deficiencies. These code coverage breakdowns hint at the prevalence of different kinds of performance inefficiencies throughout a code base. For example, if there are a small number of poorly scaling blocks in a program, then only a few lines of code contain most of the scalability issues.



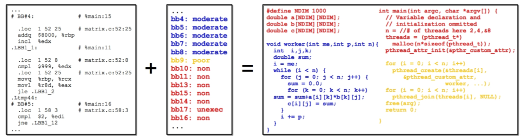
Micro-scaling metrics show that badly scaling program segments do not align well with functions. Many functions in `streamcluster` (shown left) and other applications contain basic blocks with a variety of scaling properties. For example the `pkmedian` function contains blocks that exhibit five different qualitative types of scaling behavior.

Estimated time coverage of top 10 hottest blocks

Application	DIA-time @ 2 threads	DIA-time @ 16 threads
blackscholes	99.6%	99.4%
bodytrack	97.6%	95.4%
facesim	49.2%	56.2%
fluidanimate	96.1%	92.6%
streamcluster	95.0%	99.2%
swaptions	91.7%	91.7%

They also show that blocks are responsible for a large percentage of runtime even as parallelism scales up. The table at left lists the relative time consumed by the top ten hottest blocks out of the whole program time. While the ten hottest blocks are not always the same at 2 and 16 threads, it is true that at both degrees of parallelism only a few blocks take up most of the program runtime.

Micro-Scaling Metrics in source code. It can be tricky to reason about basic block boundaries in software. To make micro-scaling metrics understandable at a high level, debug-annotated assembly code can be used to propagate the qualitative metric categorizations into source code colorings. For example, we might color perfectly scaling blocks green, moderately scaling blocks blue, poorly scaling blocks yellow, non-scaling red, etc.



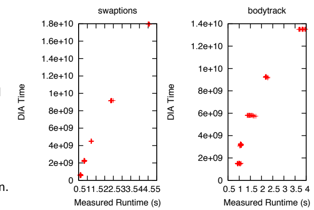
Debug-Annotated Assembly File | Per Block Scaling Categories | Line-by-line code coloring to indicate scaling properties

Methodology Details

Machine Specifications. The hardware platform for all experiments was a Dell Precision T5500 Workstation with two quad core processors (Intel X5550, 2.66 GHz, 8M last level cache) and 24 GB of RAM.

Benchmarks. We used a selection of applications in the Parsec parallel benchmark suite [2]. These applications are non-scientific, written in C and C++, parallelized using pthreads, and have a variety of scaling behaviors. We ran each application twenty times with 1, 2, 4, 8, and 16 threads.

DIA-Time Approximation. Block execution counts are not alone sufficient to determine runtime---differing block lengths and instructions cause blocks' actual execution time to vary significantly. So, we use a model called *Dynamic Instruction Approximate*, or *DIA-time:* blocks' dynamic instruction counts are linearly weighted by measured average Intel X86 instruction latencies and by the degree of parallelism exhibited by the program as a whole during a block's execution.



The Future of Micro-Scaling Metrics

- **Integration into existing performance analysis tools.** Existing parallel performance analyses either look for specific kinds of inefficiencies in software or find a broader range of inefficiencies by monitoring ineffective use of hardware and thread resources over time. In the second case, the thread or time-based resource usage patterns have to be mapped back to the source code. Instead, micro-scaling metrics monitor parallelism from the program's point of view, so that any scaling deficiencies found are already tied to the source code. We think this new, flexible, fine-grained analysis complements existing techniques, and it would be beneficial to add micro-scaling metrics to existing performance analysis tools for software engineers.
- **Quantifying the ability of micro-scaling to narrow down scaling deficiencies.** In future work, we plan to concretely measure the degree to which micro-scaling metrics narrow down scalability problems in source code. We plan to examine this both in benchmarks and real-world applications.

Acknowledgements

This work has been made possible through the generous support of the National Science Foundation (CNS-1117135).

References

[1] M. Kambadur, K. Tang, and M. A. Kim. *Harmony: Collection and analysis of parallel block vectors.* In Proceedings of the International Symposium on Computer Architecture (ISCA), *2012.*
[2] Harmony source code download http://arcade.cs.columbia.edu/harmony-2012-08-17.tar.gz
[3] C. Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, 2011.

ARCADE Lab, Columbia University, New York, NY   http://arcade.cs.columbia.edu