Brief Announcement: The Problem Based Benchmark Suite

Julian Shun[†] Guy E. Blelloch[†] Jeremy T. Fineman[*] Phillip B. Gibbons[‡]
Aapo Kyrola[†] Harsha Vardhan Simhadri[†] Kanat Tangwongsan[†]

[†]Carnegie Mellon University [*]Georgetown University [‡]Intel Labs, Pittsburgh

{jshun,guyb,akyrola,harshas,ktangwon}@cs.cmu.edu
jfineman@cs.georgetown.edu, phillip.b.gibbons@intel.com

ABSTRACT

This announcement describes the problem based benchmark suite (PBBS). PBBS is a set of benchmarks designed for comparing parallel algorithmic approaches, parallel programming language styles, and machine architectures across a broad set of problems. Each benchmark is defined concretely in terms of a problem specification and a set of input distributions. No requirements are made in terms of algorithmic approach, programming language, or machine architecture. The goal of the benchmarks is not only to compare runtimes, but also to be able to compare code and other aspects of an implementation (e.g., portability, robustness, determinism, and generality). As such the code for an implementation of a benchmark is as important as its runtime, and the public PBBS repository will include both code and performance results.

The benchmarks are designed to make it easy for others to try their own implementations, or to add new benchmark problems. Each benchmark problem includes the problem specification, the specification of input and output file formats, default input generators, test codes that check the correctness of the output for a given input, driver code that can be linked with implementations, a baseline sequential implementation, a baseline multicore implementation, and scripts for running timings (and checks) and outputting the results in a standard format. The current suite includes the following problems: integer sort, comparison sort, remove duplicates, dictionary, breadth first search, spanning forest, minimum spanning forest, maximal independent set, maximal matching, K-nearest neighbors, Delaunay triangulation, convex hull, suffix arrays, n-body, and ray casting. For each problem, we report the performance of our baseline multicore implementation on a 40-core machine.

Categories and Subject Descriptors: F.2 [Analysis of Algorithms and Problem Complexity]: General

Keywords: Parallel Algorithms, Benchmarking, Algorithm Performance

1 Introduction

When writing parallel code for a particular problem, should one use transactions, race-free algorithms, nested parallelism, bulk synchronization, speculative parallelism, futures, data parallelism, threads, message passing, or other options? Should one use a GPU, a multicore, or a cluster? Despite decades of experience with parallelism, there is still little guidance on what approach to use when writing parallel codes, especially for irregular, non-numeric applications.

To help address this challenge, we are developing a **problem-based benchmark suite** (PBBS) for a broad set of non-numeric problems (see http://www.cs.cmu.edu/~pbbs). Unlike most existing benchmarks, which are based on specific code, the benchmarks are defined in terms of the problem specifications—a concrete description of valid inputs and corresponding valid outputs, along with some specific inputs. Any algorithms, programming methodologies, specific programming languages, or machines can be used to solve the problems. The benchmark suite is designed to compare the benefits and shortcomings of different algorithmic and programming approaches, and to serve as a dynamically improving set of educational examples of how to parallelize applications. The nature of PBBS will encourage the community to submit open-source solutions that will be judged by not only its performance but also the quality of the code: its elegance, readability, extensibility, modularity, scalability, correctness guarantees, and the ability to formally analyze performance. We realize many of these measures are hard to quantify and ultimately the judgment will be in the eye of the reader. Thus, the main outcome should be the code itself (and its performance numbers).

Our benchmark problems are selected to have reasonably simple efficient solutions (our base implementations all use fewer than 500 lines of code), but represent realistic real-world problems covering a wide class of domains and potential solution approaches. These consist of many well-known problems that are already de facto standards for benchmarking, such as sorting, nearest-neighbor searching, breadth first search, Delaunay triangulation and ray tracing, as well as many others. In the suite, each benchmark consists of (1) the problem specification including specific input and output file formats, (2) input generators and specific input instances, (3) code for checking the correctness of output for the given input, (4) scripts for running tests, (5) a reasonably efficient sequential base implementation for the problem, and (6) a reasonably efficient parallel (multicore) base implementation for the problem.

2 Related Work

Many benchmark suites have been designed and are currently being used for many different purposes, but none match our goals for a problem-based suite. There are several broad-based performance-based suites such as SPEC, WorldBench, V8, and Da Capo [6]; and domain-specific benchmarks such as BioBench [2], the San Diego vision benchmarks [18], MediaBench [12] (multimedia), SATLIB [10] (satisfiability), MineBench [15] (data mining), and the TPC benchmarks (databases). Except for SATLIB and the TPC benchmarks,

SPAA'12, June 25–27, 2012, Pittsburgh, Pennsylvania, USA.
ACM 978-1-4503-1213-4/12/06.

Basic Building Blocks	Scan, Integer Sort, Comparison Sort, Remove Duplicates, Dictionary, Sparse matrix-vector multiply
Graph Algorithms	Breadth First Search, Spanning Forest, Minimum Spanning Forest, Maximal Independent Set, Maximal Matching, Graph Separators
Computational Geometry	Quad/Oct Tree, Delaunay Triangulation, Convex Hull, k-Nearest Neighbors
Text Processing	Tokenize, Suffix Array
Computational Biology	Multiple sequence alignment, Phylogenetic tree, N-body
Data Mining	Build Index, Edit Distance Graph
Graphics	Ray Casting, Micropolygon Rendering
Machine Learning	Sparse SVM, K-means, Gibbs Sampling in Graphical Models

Table 1. A (preliminary) set of 28 problem-based benchmarks covering a reasonably broad set of non-numerical applications.

these are code-based benchmarks. The TPC and SATLIB are problem based, but for specific domains.

For parallel machines, there have also been many benchmarks developed. Broad-based performance benchmarks include Splash-2 [19], PARSEC [5], and STAMP [8], which are designed for shared memory machines. Other benchmarks cover a more general class of machines but are meant to measure particular machine characteristics, such as the HPC Challenge Benchmarks [14] that put an emphasis on measuring communication throughput. There are benchmarks aimed at particular languages, such as the Java Grande Benchmark Suite [17]. There are also some domain-specific parallel benchmarks such as ALPBench [13] (multimedia) and BioParallel [11]. All these benchmarks are code-based. The Berkeley "dwarfs" define a set of 13 parallel computational patterns [3]. While sharing some of the same high-level goals as ours (e.g., evaluate parallel programming models), their benchmarks are in terms of patterns, not problems. The Galois benchmarks [16] are defined in terms of particular algorithmic approaches but are not problem based.

In terms of being defined with regards to a problem specification, perhaps the closest benchmarks to PBBS are the NAS benchmarks [4]. In the original form (NPB 1), these consisted of a set of eight problem-based benchmarks where one of the main goals was architecture neutrality. Indeed, several different programming styles (vector code, message passing, data parallel) were used to code the benchmarks on different machines. These benchmarks, however, did not focus on code quality and because vendors were not required to release their codes, some of the solutions were extremely messy. Also, the NAS benchmarks were focused on numerical computing.

Finally, there have also been various attempts to compare programming languages by defining a set of benchmarks. Probably the one that captures the broadest set of languages is the Computer Language Benchmarks Game [9], which compares over 25 programming languages on a set of 12 micro benchmarks. Benchmarks results are reported in terms of performance and size of the `gzip`-compressed source file (comments and redundant whitespace removed). The benchmarks, however, only consider small inputs—for example, their "n-body" benchmark consists of 5 bodies. Also, the benchmarks require that the program use the "same algorithm" as specified—returning the same result is not sufficient.

In summary, we know of no benchmark suite that matches our goals—i.e., defined in input/output terms, covers a broad set of non-numeric problems, scales to large problem sizes, and emphasizes code quality.

3 Benchmark Problems and Current Status

We selected benchmark problems with the following goals in mind. First, the set of problems should have a wide coverage from state-of-the-art real-world applications. Second, the problem must have a well-defined way to validate output correctness or quality. Third, the problem should have efficient solutions that can be implemented in a reasonably small program. Finally, the inputs to these problem should be scalable. Table 1 summarizes a set of problem-based benchmarks categorized by application domain or type of data. These 28 benchmarks represent our current list of what we believe would make a good mix of problems, though the list is flexible.

An important challenge with defining a benchmark in terms of a problem's input-output behaviors is picking a good and scalable set of test inputs. A good set of test inputs should withstand "tricks" that fail to work in the general case, should represent a realistic input, and should have varying sizes. We leverage the growing body of work on generating scalable synthetic data that models real data. There are many standard distributions used in computational geometry that are much more realistic than evenly distributed random data. Similarly, there has been considerable work in generating graphs that have characteristics similar to real-world graphs [1] and DNA data that represents a population.

We require the program to output the result to a file in a particular format. We provide test code that checks correctness and outputs any quality criteria (e.g. the size of a graph cut). The time for input and output is not included in the running time or code length—for some benchmarks it could dominate the cost.

It is important to have at least one base implementation of each benchmark so that results can be compared and as a proof of concept that the benchmarks fit within our parameters (e.g., have reasonably simple and efficient solutions). We are currently developing two base implementations for each benchmark, one serial and one parallel. Our parallel implementations are designed for multicores and use only parallel loops, nested fork-join, and compare-and-swap operations and are currently implemented in Intel Cilk Plus. We have implemented an initial set of base implementations for some of the benchmarks and have made initial timings. We ran our experiments on a 40-core (with hyper-threading) machine with 4×2.4GHZ Intel 10-core E7-8870 Xeon Processors, a 1066MHz bus, and 256GB of main memory. All programs were compiled with Intel's `icpc` compiler (version 12.1.0 with Cilk Plus support) with the `-O3` flag.

Table 2 summarizes the results of these experiments. We report the weighted average of runtimes over various inputs. For example, for Comparison Sort, we use three sequences of doubles distributed according to uniform, exponential, and almost sorted distributions and two sequences of character strings from a trigram distribution. All sequences were of length 10^7. For the graph benchmarks, we used three types of graphs: random graphs, grid graphs, and rMat (power law) graphs. Each graph has either 10^7 or 2^{24} nodes. Descriptions of all the algorithms and further experimental results can

Application Algorithm	**1** **thread**	**40** **core**	T_1/T_{40}	T_S/T_{40}
Integer Sort				
serialRadixSort	0.48	–	–	–
parallelRadixSort	0.299	0.013	23.0	36.9
Comparison Sort				
serialSort	2.85	–	–	–
sampleSort	2.59	0.066	39.2	43.2
Remove Duplicates				
serialHash	0.689	–	–	–
parallelHash	0.867	0.027	32.1	25.5
Dictionary				
serialHash	0.574	–	–	–
parallelHash	0.748	0.025	29.9	23
Breadth First Search				
serialBFS	2.61	–	–	–
parallelBFS	5.54	0.247	22.4	10.6
Spanning Forest				
serialSF	1.733	–	–	–
parallelSF	5.12	0.254	20.1	6.81
Min Spanning Forest				
serialMSF	7.04	–	–	–
parallelKruskal	14.9	0.626	23.8	11.2
Maximal Ind. Set				
serialMIS	0.405	–	–	–
parallelMIS	0.733	0.047	14.1	8.27
Maximal Matching				
serialMatching	0.84	–	–	–
parallelMatching	2.02	0.108	18.7	7.78
K-Nearest Neighbors				
octTreeNeighbors	24.9	1.16	21.5	–
Delaunay Triangulation				
serialDelaunay	56.3	–	–	–
parallelDelaunay	76.6	2.6	29.5	21.7
Convex Hull				
serialHull	1.01	–	–	–
quickHull	1.655	0.093	17.8	10.9
Suffix Array				
serialKS	17.3	–	–	–
parallelKS	11.7	0.57	20.5	30.4
Ray Casting				
kdTree	7.32	0.334	21.9	–

Table 2. Weighted average of running times (seconds) over various inputs on a 40-core machine with hyper-threading (80 threads). Time of the parallel version on 40 cores (T_{40}) is shown relative to both (i) the time of the serial version (T_S) and (ii) the parallel version on one thread (T_1). In some cases our parallel version on one thread is faster than the baseline serial version.

be found in [7]. All code is available on the benchmark web page: http://www.cs.cmu.edu/~pbbs.

Acknowledgements. This work is partially funded by the National Science Foundation under Grant number 1019343 to the Computing Research Association for the CIFellows Project and Grant number CCF-1018188, and by Intel via the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program and the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

4 References

[1] L. Akoglu and C. Faloutsos. RTG: A recursive realistic graph generator using random typing. *Data Min. Knowl. Discov.*, 19(2), 2009.

[2] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. BioBench: A benchmark suite of bioinformatics applications. In *IEEE ISPASS*, 2005.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, UC Berkeley, 2006.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks—summary and preliminary results. In *ACM/IEEE Supercomputing*, 1991.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *ACM PACT*, 2008.

[6] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM OOPSLA*, 2006.

[7] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *ACM PPoPP*, 2012.

[8] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08*, September 2008.

[9] B. Fulgham. The computer language benchmarks game. http://shootout.alioth.debian.org/, 2009.

[10] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. P. Gent, H. v. Maaren, and T. Walsh, editors, *SAT 2000*. IOS Press, 2000.

[11] A. Jaleel, M. Mattina, and B. Jacob. Last-level cache (LLC) performance of data-mining workloads on a CMP—A case study of parallel bioinformatics workloads. In *IEEE HPCA*, 2006.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *IEEE/ACM MICRO*, 1997.

[13] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *IEEE IISWC*, 2005.

[14] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D. Takahashi. The HPC challenge (HPCC) benchmark suite. In *ACM/IEEE SC06 Conference Tutorial*, 2006.

[15] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. N. Choudhary. MineBench: A benchmark suite for data mining workloads. In *IEEE IISWC*, 2006.

[16] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *ACM PLDI*, 2011.

[17] L. A. Smith, J. M. Bull, and J. Obdrzalek. A parallel Java Grande benchmark suite. In *ACM/IEEE SC2001*, 2001.

[18] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego vision benchmark suite. In *IEEE IISWC*, 2009.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ACM ISCA*, 1995.