Poster: The Problem-Based Benchmark Suite (PBBS), V2

Daniel Anderson
Carnegie Mellon University
Pittsburgh, PA, USA
dlanders@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, PA, USA
guyb@cs.cmu.edu

Laxman Dhulipala
University of Maryland, College Park
College Park, MD, USA
laxman@umd.edu

Magdalen Dobson
Carnegie Mellon University
Pittsburgh, PA, USA
mrdobson@cs.cmu.edu

Yihan Sun
UC Riverside
Riverside, CA, USA
yihans@cs.ucr.edu

Abstract

The Problem-Based Benchmark Suite (PBBS) is a set of benchmark problems designed for comparing algorithms, implementations and platforms. For each problem, the suite defines the problem in terms of the input-output relationship, and supplies a set of input instances along with input generators, a default implementation, code for checking correctness or accuracy, and a timing harness. The suite makes it possible to compare different algorithms, platforms (e.g. GPU vs CPU), and implementations using different programming languages or libraries. The purpose is to better understand how well a wide variety of problems parallelize, and what techniques/algorithms are most effective.

The suite was first announced in 2012 with 14 benchmark problems. Here we describe some significant updates. In particular, we have added nine new benchmarks from a mix of problems in text processing, computational geometry and machine learning. We have further optimized the default implementations; several are the fastest available for multicore CPUs, often achieving near perfect speedup on the 72 core machine we test them on. The suite now also supplies significantly larger default test instances, as well as a broader variety, with many derived from real-world data.

CCS Concepts: • **Computing methodologies → Parallel algorithms**.

Keywords: benchmarking, parallel algorithms, performance

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508422

1 Introduction

The Problem Based Benchmark Suite (PBBS) [8] is a collection of benchmark problems aimed at helping better understand the most effective algorithms and implementations for a variety of common and widely used problems. Unlike most benchmarks which are based on specific code and achieving the best performance out of that code [2, 3, 10] (e.g., using new hardware, runtimes, or compiler techniques), the benchmarks in PBBS are defined in terms of an I/O specification. The design of the suite was motivated by the need to better understand the tradeoffs among the many different hardware and software platforms and algorithmic techniques that can be used to get good parallel performance. They are hence designed to be agnostic to the programming language used, programming style used (e.g., nested parallel vs. bulk synchronous vs. map-reduce), and hardware platform (e.g., GPU vs CPU vs distributed memory). The most similar other benchmarks are the Parboil benchmarks [9]. However they focus mostly on regular scientific tasks, and other than BFS and histogram there is no overlap in the problem sets.

The original suite was released a decade ago and has been used in several projects. In this poster we describe several important updates that have been made to the suite. In PBBS, each benchmark supplies the following, using comparison-based sorting as an example:

- The definition of the input problem, e.g., takes an input sequence and a comparator, and returns an output sequence with the same elements sorted based on the comparator.
- Specification of default input instances along with generators for them. For sorting the instances are sequences of doubles generated in three distributions (uniform, exponentional, near-sorted), as well as a sequence of pairs of doubles, and a sequence of strings.
- A testing program that takes an input and the output generated by an implementation and checks it for correctness, e.g., checks the output has the same keys and is sorted.
- A timing harness for timing the code. This requires that the code can be linked with C++, otherwise the user can print out times in a specific format.

- Scripts for generating the specified inputs, running the timing harness, and running the testing code. By default the implementation is run three times and the geometric mean of times is reported.
- At least one parallel implementation designed for CPUs.

The updates we have made to the benchmark suite include the following. Firstly, we have added nine new benchmarks, marked in Table 1, bringing the total to twenty-two (we dropped a dictionary benchmark). We added four in the category of text processing since the previous benchmarks only had one, and it is an important class of applications. We also added a feature-based classification benchmark since this is a broadly used application.

Secondly, we have added many more implementations of some of the benchmarks. Our original suite only had at most one parallel and one sequential implementation. We now have several implementations of some of the benchmarks. Sorting now has six implementations–sample sort, stable sample sort, merge sort, quicksort, serial sort, and ips4o sort [1]. In some cases one algorithm dominates while in other cases it depends on the particular input. Some of the additional implementations are our own, and some are contributed by others.

Thirdly, we have improved the performance of many of the default implementations. The nearest-neighbor default parallel implementation, for example, is 3x faster than the previous version. The integer sort is also considerably faster, and the BFS now uses the backward-forward optimization from Ligra [7] and matches the GBBS [5] performance.

Fourthly, we have broadened the set of input sets and distributions. Originally, for example, we only had some synthetic graphs, we now use a variety of graphs from the SNAP graph dataset [6]. Similarly for the text, we have accumulated a collection of real text from wikipedia, DNA sources, and elsewhere. We also now include two sets of input instances for each benchmark, one small and one large. The small one is similar in size to the original, and the large one is an order of magnitude larger.

Fifthly, the framework is now built on ParlayLib [4], a library of parallel tools for C++. Importantly, ParlayLib has been ported to and tested on a wide variety of compilers and platforms. In addition to the testing and timing harnesses, and data generators, for each benchmark we have at least one parallel implementation based on ParlayLib.

Community Contributions. We would like to encourage researchers to contribute to PBBS. This can include contributing new implementations either on multicores or on other platforms (currently all our implementations are for multicores). It can also include new benchmarks, or improvements to the framework. Currently contributions can be made by making a git push request. Some information on the expectations of benchmark are given below, and more in the online documentation.

Basic Building Blocks		
	SORT	Comparison based sort.
*	HIST	Histogram keys in given integer range.
	ISORT	Sorts integer keys in given range along with data.
	DDUP	Remove duplicates (integers, pairs, and strings).
Graph Algorithms		
	BFS	Return a breadth-first-search tree from a given vertex.
	MIS	Return a maximal independent set.
	MM	Return a maximal matching.
	SF	Return a spanning tree.
	MSF	Return a minimum spaning tree with float weights.
Text Processing		
	SA	Suffix array of a string of bytes.
*	WC	Break string into words, and report count for each word.
*	IIDX	Parse documents and generate inverted index.
*	LRS	Find the longest repeated substring.
*	BWD	Inverse Burrows-Wheeler transform on string of bytes.
Geometry and Graphics		
	CH	2D Convex Hull of ponts in clockwise order.
	DT	2D Delaunay triangulation of set of points.
*	DR	2D Delaunay refinement of set of triangles.
	KNN	k Nearest neighbors of points in 2D and 3D.
	RAY	In 3D determine first triangle each ray intersects.
*	RQ	In 2D for each rectangle count points it contains.
Other		
*	CLASS	Predict labels for feature vectors given training vectors.
*	NBODY	Determine gravitational forces among n bodies in 3D.

Table 1. Benchmark descriptions. * indicates it is new in V2.

References

[1] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2017. In-place parallel super scalar samplesort (IPS4o). In *European Symposium on Algorithms (ESA).*

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *ACM Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT).*

[3] Stephen M. Blackburn and et. al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA).*

[4] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).*

[5] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *Intl. Workshop on Graph Data Management Experiences and Systems (GRADES).*

[6] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[7] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP).*

[8] Julian Shun, Guy E. Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the Problem-Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).*

[9] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing.* Technical Report IMPACT-12-01. UIUC.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ACM Int. Symposium on Computer Architecture (ISCA).*

A Artifact Description

A.1 Overview

The artifact is the full contents of the problem-based benchmark suite (PBBS) version 2 as described above. To evaluate performance of the benchmarks, we provide a script for running the benchmarks. We also outline here the structure of the directories.

A.2 Requirements

Running experiments requires the following.

- **Operating System**: Linux or macOS.
- **Hardware (small instances)**: Multicore machine with at least 12GB of memory and 1GB of disk space.
- **Hardware (large instances)**: Multicore machine with at least 64GB of memory and 10GB of disk space.
- **Software**: A C++-17 compiler. jemalloc (or other efficient allocator) is not required, but improves performance. numactl is also not required but improves performance on multichip systems. CGAL is required to be installed for one of the optional implementations but not needed for the default ones.

Timing results are reported to standard output (stdout).

A.3 How delivered

The benchmark suite is available on GitHub:

https://github.com/cmuparlay/pbbsbench

Documentation can be found at:

https://cmuparlay.github.io/pbbsbench/

A.4 Setup

The suite can be cloned via github using:

```
$ git clone https://github.com/cmuparlay/pbbsbench.git
$ cd pbbsbench
$ git submodule update --init
```

The repository includes three sub-repositories: ParlayLib, PAM and ips4o which will be loaded as submodules. The ParlayLib library is used throughout, but the other two are just used for one benchmark each.

A.5 Running the benchmarks

The top level directory includes a script `runall` for running the benchmarks in various ways. Using

```
$ ./runall
```

will compile, run, verify results and report times for all the default benchmark implementations (at least one for each problem) on the large instances. This runs each implementation on all available cores (if parallel) and can take an hour or more. The following optional arguments can be used:

`-par` : just run one parallel implementation of each problem.
`-ext` : run all implementations, not just the defaults.
`-only <prob>/<impl>` : only run the implementation `<impl>` of the benchmark `<prob>`.
`-small` : use the small problem instances instead of the large ones.
`-scale` : run scaling experiments using increasing number of cores.
`-nonuma` : do not use `numactl`. This option must be passed if `numactl` is not installed.
`-nocheck` : do not check correctness of result.

For a quick run try:

```
$ ./runall -par -small -nocheck
```

A.6 Benchmark Directories

Within the `benchmarks` directory at toplevel is a subdirectory for each benchmark. The documentation for the benchmarks, including specification and description of default input instances, is available in the general documentation.

Within each benchmarks is a subdirectory for each of the implementations. Each benchmark also has some directories shared across implementations. In particular each has a directory called `bench` containing the driver and testing code. Each benchmark also has a xxxData page containing data generators for the benchmark (xxx varies by benchmark).

Within each implementation directory, you can run `make` to make the executable, and then run `./testInputs` to run the benchmarks. These are run automatically by the `./runall` script. On a machine with multiple chips, using

```
$ numactl -i all ./testInputs
```

will give better results. `./testInputs_small` will use the smaller inputs. See the documentation for optional arguments. The inputs are specified in the script and can be changed if desired.

To add an implementation—we would love contributions—create a new directory within the benchmark. It is probably best to start by copying an existing one.

A.7 Timing

Users can use the benchmark suite as they please, but there are certain expectations for timing an implementation so that it can be properly compared with other implementations, including the default ones. We do not expect a full end-to-end timing of the executable because often the time is dominated by reading the input file and possibly writing an output file, which have little to do with the benchmark. Instead we expect a benchmark to wrap a timer (real time) around the benchmark itself. More details and the particular format for outputting the result and times so they can be used by the scripts is given in the documentation.

Acknowledgments

Funding provided by the National Science Foundation grants CCF-1901381, CCF-1910030, CCF-1919223, and CCF-2103483.