# POSTER: ParGeo: A Library for Parallel Computational Geometry

Yiqiu Wang[1], Shangdi Yu[1], Laxman Dhulipala[1], Yan Gu[2], Julian Shun[1]
{yiqiuw, shangdiy, laxman}@mit.edu ygu@cs.ucr.edu jshun@mit.edu
[1]MIT CSAIL [2]UC Riverside

**Abstract:** We present ParGeo, a multicore library for computational geometry algorithms. We describe two of the algorithms from ParGeo, convex hull and the smallest enclosing ball, and present a short evaluation of all implementations currently in ParGeo.

## 1 Introduction

Computational geometry algorithms have important applications in areas such as graphics, robotics, computer vision, and geographic information systems. There exist numerous libraries for computational geometry, but most of them are not designed for parallel processing. For example, CGAL [1] is a famous library of computational geometry algorithms that includes a wide range of packages, but most implementations are not parallel. Batista et al. [2] parallelize the spatial sorting, box intersection, and Delaunay triangulation algorithms in CGAL. Libigl specializes in the construction of discrete differential geometry operators and finite-element matrices. Other libraries include PMP, Cinolib, and Tetwild, which are designed for polygonal and polyhedron meshes. These libraries are partially parallelized and tackle different problems from CGAL.

This paper presents the ParGeo library, which targets similar classes of problems as CGAL, but contains many more parallel multicore implementations of algorithms than prior work. A subset of the algorithms in ParGeo are from the Problem Based Benchmark Suite [8]. The code for ParGeo is publicly available at https://github.com/ParAlg/ParGeo.

## 2 The ParGeo Library

We describe here the current (preliminary) implementations in ParGeo, and we plan to continue adding more implementations to ParGeo. Many of the implementations have been ported into ParGeo from our prior work [10–12, 14].

ParGeo contains efficient multicore implementations of $k$d-trees. The code supports $k$d-tree based spatial search, including $k$-nearest neighbor and range search. The code can also compute the bichromatic closest pair by traversing two $k$d-trees. Our code is optimized for fast $k$d-tree construction by performing the split in parallel, and the queries themselves are data-parallel. We also include a parallel batch-dynamic $k$d-tree that supports batch insertions and deletions. Our $k$d-tree can be used to generate a well-separated pair decomposition (WSPD), which can be used to compute the Euclidean minimum spanning tree (EMST) and spanners.

In addition, ParGeo contains parallel implementations for classic algorithms in computational geometry, including Morton sorting, closest pair, convex hull, and smallest enclosing ball. ParGeo also contains a collection of geometric graph generators for point data sets. It includes routines for constructing $k$-nearest neighbor graphs using $k$d-trees, and also supports common spatial network graphs, including the Delaunay graph and the $\beta$-skeleton graph.

Below, we briefly describe our parallel implementations for convex hull and smallest enclosing ball, which we have not presented in prior work.

**Convex Hull.** The convex hull of a set of points in $\mathbb{R}^d$ is the smallest convex polyhedron containing all of the points. ParGeo includes a practical parallel incremental algorithm for $\mathbb{R}^2$ and $\mathbb{R}^3$ that is able to express both the randomized incremental algorithm and the quickhull algorithm. The high level idea is similar to a sequential incremental algorithm, where a convex hull is iteratively updated by adding points in a round-based manner. However, unlike a sequential incremental algorithm that adds one point per round, we add multiple points in parallel in a round. The key challenge is that some of the points cannot be processed in parallel due to concurrent modifications on the shared convex polyhedron. We use a reservation algorithm [3] to resolve these conflicts, such that we only process the points that modify disjoint facets of the polyhedron. We give each point a unique ID, and have it perform priority concurrent writes with its ID to reserve all of its visible facets. Only a point that has its ID written to all of its visible facets will proceed to modify the polyhedron. The algorithm terminates when there are no more points outside of the polyhedron. Our ideas are based on a recent theoretical algorithm from Blelloch et al. [5].

We have also implemented existing algorithms which use our incremental algorithm as a subroutine. They include a divide-conquer algorithm that divides the input based on the number of processors and combines the partial results in the end, as well as the pseudo-hull algorithm by Tang et al. [9].

**Smallest Enclosing Ball.** The smallest enclosing ball of set


*PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea*

ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508429

of points in $\mathbb{R}^d$ is the smallest $d$-sphere containing all of the points. It is well known that the smallest enclosing ball is unique and defined by a *support set* of $d+1$ points on the surface of the ball. For the smallest enclosing ball problem in constant dimensions, the algorithm and implementation of Gartner [6], which is a variation of Welzl's randomized incremental algorithm [13], is the state of the art in the sequential setting. In the parallel setting, Larsson et al. [7] developed practical algorithms for both CPUs and GPUs.

ParGeo includes a new sampling-based algorithm for smallest enclosing ball in $\mathbb{R}^2$ and $\mathbb{R}^3$, based on Larsson et al.'s approach to quickly reduce the size of the data set. Our sampling algorithm is based on Larsson et al.'s orthant-scan [7]. Similarly to orthant-scan, our algorithm scans the input to search for good support sets in a round-based manner. Our algorithm divides the space into orthants (4 for $\mathbb{R}^2$ and 8 for $\mathbb{R}^3$) centered at the center of an initial ball. On each iteration, we scan through a small random sample, and find the furthest visible samples in each orthant. The ball is then updated to the next intermediate solution using the existing support set and the new visible samples found. The algorithm iterates until there are no more visible points, followed by a full orthant-scan to compute the final smallest enclosing ball. Compared with Larsson's algorithm, our sampling algorithm avoids repeatedly scanning the input in the initial stage. We parallelize the orthant scan by dividing the input array to into blocks and processing each block sequentially, but in parallel across different blocks.

ParGeo also includes an optimized implementation of Blelloch et al.'s [4] parallelization of Welzl's algorithm, with the move-to-front (MTF) and pivoting heuristics [6, 13].

## 3 Experimental Evaluation

We run experiments on a 36-core machine with two-way hyper-threading, and 144 GiB of RAM. Our data sets contain 10 million uniformly distributed points in a hypercube in 2, 3, and 5 dimensions. Table 1 shows the running times and parallel speedup for the implementations in ParGeo. Our implementations achieve parallel speedups of 4.07–46.61x (22.74x on average). Figure 1 compares the performance of different implementations of convex hull and smallest enclosing ball. For convex hull, we observe that the existing sequential implementations in CGAL is significantly slower than our parallel implementations. Our parallel divide-and-conquer (DC-Inc) and pseudo-hull (PH-Inc) [9] implementations that use our incremental algorithm as a sub-routine achieve the fastest running times. For smallest enclosing ball, Larsson's orthant-scan (OrthScan) [7] and our sampling algorithm are significantly faster than the implementation in CGAL and our parallelized Welzl's algorithm with heuristics.

**Acknowledgements.** This research is supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, NSF Award #CCF-2103483, Google Faculty Research Award, Google Research Scholar Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

| Implementation | $T_1$ | $T_{36h}$ | Speedup ($T_1/T_{36h}$) |
|---|---|---|---|
| $k$d-tree Build (2d) | 5.51 | 0.43 | 12.70x |
| $k$d-tree Build (5d) | 8.39 | 0.89 | 9.40x |
| $k$d-tree $k$-NN (2d) | 31.45 | 0.68 | 46.34x |
| $k$d-tree Range Search (2d) | 17.14 | 0.37 | 46.61x |
| Dynamic $k$d-tree Insert (2d) | 2.43 | 0.60 | 4.07x |
| Dynamic $k$d-tree Delete (2d) | 1.09 | 0.14 | 7.69x |
| WSPD (2d) | 6.72 | 0.24 | 27.63x |
| EMST (2d) | 33.02 | 1.58 | 20.86x |
| Convex Hull (2d) | 0.38 | 0.0088 | 43.13x |
| Convex Hull (3d) | 2.36 | 0.097 | 24.36x |
| Smallest Enclosing Ball (2d) | 0.053 | 0.0033 | 16.30x |
| Smallest Enclosing Ball (5d) | 0.13 | 0.014 | 9.54x |
| Closest Pair (2d) | 10.35 | 0.52 | 19.90x |
| Closest Pair (3d) | 28.00 | 2.32 | 12.07x |
| $k$-NN Graph (2d) | 37.89 | 1.46 | 25.99x |
| Delaunay Graph (2d) | 55.91 | 2.03 | 27.53x |
| Gabriel Graph (2d) | 59.61 | 1.99 | 29.99x |
| $\beta$-skeleton Graph (2d) | 113.27 | 3.20 | 35.37x |
| Spanner (2d) | 27.19 | 2.15 | 12.67x |

**Table 1.** Runtimes (seconds) and parallel speedups for ParGeo implementations on hypercube data sets with 10 million points. $T_1$ and $T_{36h}$ denote the single-threaded and the 36-core hyper-threaded times, respectively.

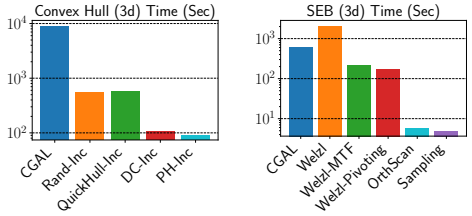


**Figure 1.** Runtimes (seconds) of convex hull (left) and smallest enclosing ball (right) on the 3d hypercube data set using all cores.

## References

[1] The computational geometry algorithms library. www.cgal.org.
[2] V. H. Batista, D. L. Millman, S. Pion, and J. Singler. Parallel geometric algorithms for multi-core computers. *Comput. Geom.*, 2010.
[3] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP*, 2012.
[4] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallelism in randomized incremental algorithms. In *SPAA*, 2016.
[5] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Randomized incremental convex hull is highly parallel. In *SPAA*, 2020.
[6] B. Gärtner. Fast and robust smallest enclosing balls. In *ESA*, 1999.
[7] T. Larsson, G. Capannini, and L. Källberg. Parallel computation of optimal enclosing balls by iterative orthant scan. *Comput. Graph.*, 2016.
[8] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: The problem based benchmark suite. In *SPAA*, 2012.
[9] M. Tang, J. Zhao, R. Tong, and D. Manocha. GPU accelerated convex hull computation. *Computers & Graphics*, 2012.
[10] Y. Wang, S. Yu, L. Dhulipala, Y. Gu, and J. Shun. Geograph: A framework for graph processing on geometric data. *SIGOPS OSR*, 2021.
[11] Y. Wang, S. Yu, Y. Gu, and J. Shun. Fast parallel algorithms for Euclidean minimum spanning tree and hierarchical spatial clustering. In *SIGMOD*, 2021.
[12] Y. Wang, S. Yu, Y. Gu, and J. Shun. A parallel batch-dynamic data structure for the closest pair problem. In *SoCG*, 2021.
[13] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, 1991.
[14] R. Yesantharao, Y. Wang, L. Dhulipala, and J. Shun. Parallel batch-dynamic $k$d-trees. *arXiv*, 2021.

## A Artifact

In this section, we present the instructions for our artifact, which is publicly available at https://zenodo.org/record/5812180.

**Setting up the environment and dependencies.** We perform all of our experiments on an Amazon EC2 `c5.18xlarge` instance with 36 cores and 144 GB of RAM, using Ubuntu 20.04.2 LTS. After setting up the instance, an Internet connection is required throughout the process for downloading dependencies for compiling, testing, and benchmarking the code.

Install the dependencies with the following commands:

```
sudo apt-get update
sudo apt-get -y install build-essential
sudo apt-get -y install cmake
```

**Compiling and running the benchmarks.** Download the artifact, unzip it, and navigate to the artifact root directory, which contains a `README.md` file.

Since all of the necessary steps are included in the `runme.sh` script, simply run the script with `sh runme.sh`. The final benchmarking results will be automatically generated as JSON files in the `build/benchmark/` directory.

**Generating Table 1.** After running `runme.sh`, create the table by running `sh tabulate.sh`. The script will first copy relevant JSON files from the `build/benchmark/` directory to the `plot/` directory, and then compute the speedups using a script. The speedups will be displayed on the console, and they should be similar to the numbers reported in Table 1.

**Generating Figure 1.** The generation of the plot requires Python3 and the `matplotlib 3.4.2` package. Assuming Python3 is already installed, install the plotting dependency with the following commands:

```
sudo apt install -y python3-pip
pip3 install matplotlib
```

Run the plotting script by running `sh plot.sh` from the artifact root directory. The plotting script first copies the JSON outputs from the `build/benchmark/` directory to the `plot/` directory, and then generates the plots. After that, the plots can be found in the `plot/` directory as PDF files, which correspond to Figure 1. The running times of the baselines whose implementations do not belong to us are not generated.