Note: we added the new Table 4 after the artifact submission, so everything after it has been shifted over by 1 (so some of the script names no longer match the position of the current table in the paper, but they still work).

Setup instructions (Getting Started)

Machine specs

Please use a machine with preinstalled g++ (at least version 11) and git. We have tested the artifact on an Amazon c6i.metal instance running Ubuntu 20.04 with 128 threads and 256 GB of memory) and g++ 11.4.

To run PAM/CPAM, you will also need jemalloc (instructions below how to install it)

To make the plots, you will need python with matplotlib (installation instructions: https://matplotlib.org/stable/users/installing/index.html).

The test machine should have multiple threads but does not necessarily need 128 threads. In terms of memory, the known minimum necessary to run the graph evaluation is 118 GB. This amount of memory is needed to run on the largest graph we tested (Friendster), but you could probably do with less to run the smaller graphs and microbenchmarks. If there are any concerns about the hardware configuration you have, please reach out.

The code should compile and run on non x86 machines, but the performance was only tested on the machine above.

Make sure you have the right version of g++:

Check that you have a sufficiently recent version of g++ (we have verified this artifact for g++ 11):
`g++ -version`

If your g++ version is not at least 11, check to see if you already have a more up-to-date version installed by looking in `/usr/bin/g++*`

If you find an appropriately up-to-date binary there (eg g++-11), you can point to it like so:
`export CXX=g++-11`

If you don't have g++>=11, install it (example instructions here: https://stackoverflow.com/questions/67298443/when-gcc-11-will-appear-in-ubuntu-repositories):

```
sudo apt install build-essential manpages-dev software-properties-common
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
```

```
sudo apt update && sudo apt install gcc-11 g++-11
```

Then point to it:
```
export CXX=g++-11
```

Get the code:

From zip:

Download the zip file (`Packed-Memory-Array.zip`) and unzip it. This will be the top level Packed-Memory-Array/ directory in later instructions.

With git:

Clone the repo:

```
git clone https://github.com/wheatman/Packed-Memory-Array.git
```

Go to the repo and switch to the branch `for_artifact`:

```
cd Packed-Memory-Array
git checkout for_artifact
```

Set up the submodules:
```
git submodule init
git submodule update
```

Compile PMA/CPMA:

Make sure you are in the top-level `Packed-Memory-Array` directory.

Build the binaries:
```
make PARLAY=1 build/basic_uint64_t_uncompressed_Eytzinger
make PARLAY=1 build/basic_uint64_t_delta_compressed_Eytzinger
```

After compiling, make sure that you got the binaries by looking in
`Packed-Memory-Array/build`

Compile PAM/CPAM:

You need jemalloc on top of the other dependencies above.

If you have sudo:

```
sudo apt update
sudo apt install libjemalloc-dev
```

Otherwise, you need to build it from source (if you don't have sudo):

Directions from: https://github.com/jemalloc/jemalloc/blob/dev/INSTALL.md and https://github.com/jemalloc/jemalloc/wiki/Getting-Started

Assuming you are in some directory dir/

```
wget https://github.com/jemalloc/jemalloc/releases/download/5.3.0/jemalloc-5.3.0.tar.bz2

tar xjf jemalloc-5.3.0.tar.bz2
cd jemalloc-5.3.0

./configure --prefix path/to/install/jemalloc (for example, I used <dir>/jemalloc-5.3.0/build
make
make install
```

Then set the path to point to where you installed it:
`PATH=/path/to/install/jemalloc:$PATH`

The makefiles have been edited to accommodate jemalloc installed from source with
`-L`jemalloc-config --libdir` -Wl,-rpath,`jemalloc-config --libdir` -ljemalloc `jemalloc-config --libs``

Setting up RMA:

Requirements

The serial batch insert experiments for RMA (https://ieeexplore.ieee.org/document/8731468) require sudo access. If you do not have sudo, you can skip this part.

Build/run RMA

First, from the top-level `Packed-Memory-Array` directory, go to the RMA directory:
`cd other_systems/rma`

To build RMA, run
`sh compile_rma.sh`

Running the RMA requires changing around some settings about huge pages which can be done with the following commands run with sudo:

```
sudo echo 4294967296 > /proc/sys/vm/nr_overcommit_hugepages
sudo echo 1 > /proc/sys/vm/overcommit_memory
```

Availability

The permanent artifact is available at https://zenodo.org/records/10222939. For ease of access the code is also available on github at https://github.com/wheatman/Packed-Memory-Array

Reproducing the paper:

Microbenchmarks:

Figure 1 (and Table 7)

To run PMA/CPMA:

After you have built the parallel binaries with PARLAY=1 (see Compile PMA/CPMA)

From `Packed-Memory-Array`, run:

```
cd scripts
sh run-fig-1.sh
```

This should take a few hours. After it is done, make sure you have two output files in `scripts/outputs/`: `cpma_parallel_batch.out` and `pma_parallel_batch.out`.

To run PAM / CPAM:

IMPORTANT: make sure you have installed jemalloc (either through apt or through source) before you start this part

From the top level `Packed-Memory-Array` directory, do:

```
cd other_systems/CPAM/examples/microbenchmarks
make testParallel-CPAM-NA testParallel-CPAM-NA-Diff testParallel-CPAM-NA-Seq
testParallel-CPAM-NA-Diff-Seq testParallel-PAM-NA -j
```

Next, run the batch insert tests for PAM / CPAM:

```
sh run_fig_1.sh
```

The entire script will take 1-2 days, but the small batch sizes take almost all of the time. After it is done, make sure you have output files in outputs/: `cpac_parallel_batch.out`, `pam_parallel_batch.out`, and `upac_parallel_batch.out`

Make the plots:

Go back to `Packed-Memory-Array/scripts/`

```
python3 make-fig-1.py
```

Which will make batch_insert_micro.pdf (Fig 1) in plots/ and batch_insert_micro.csv (Table 7) in csvs/ (both under scripts/).

Example of what the output will look like, which can be compared visually to Figure 1 in the paper:

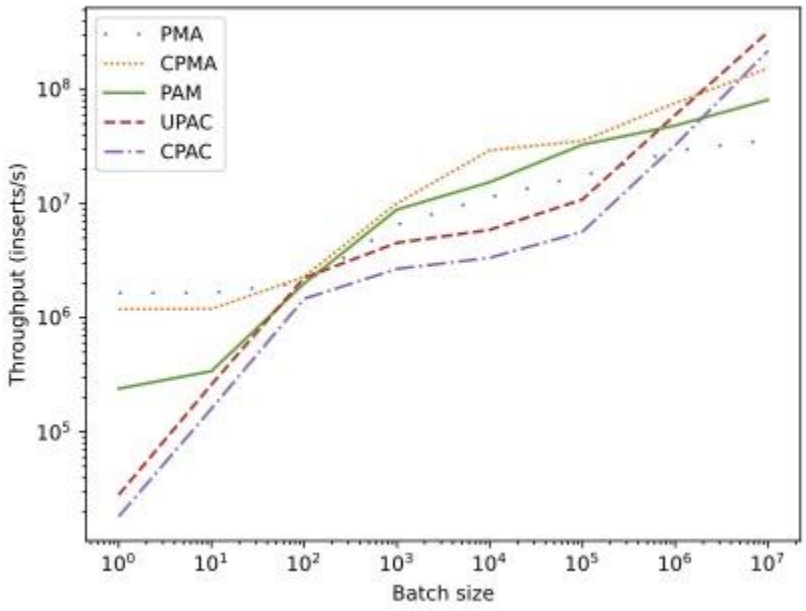


Figure 2 (and Table 8)

Make sure you have built the parallel binaries with PARLAY=1 (see above instructions under Compile PMA/CPMA)

To run PMA/CPMA:

From `Packed-Memory-Array/scripts`, run:

```
sh run-fig-2.sh
```

Make sure you have two output files in scripts/outputs/: `cpma_parallel_map_range.out` and `pma_parallel_map_range.out`

To run PAM / CPAM:

IMPORTANT: make sure you have installed jemalloc (either through apt or through source) before you start this part

If you have not built the binaries yet, look under

From the `Packed-Memory-Array/other_systems/CPAM/examples/microbenchmarks` directory, build the binaries if you haven't yet (see Figure 1, run PAM/CPAM). Then run:

```
sh run_fig_2.sh
```

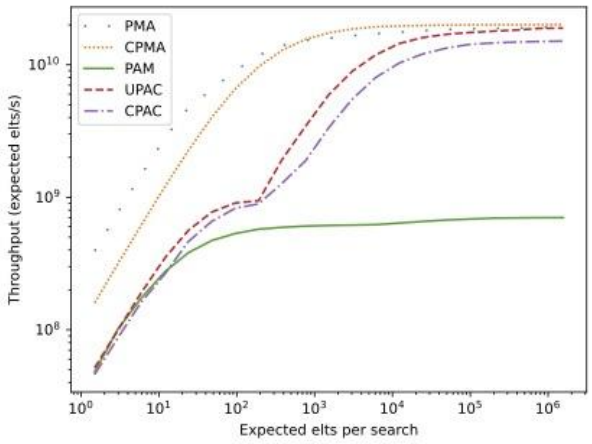
Make the plots:

Go back to `Packed-Memory-Array/scripts/`

```
python3 make-fig-2.py
```

Which will make map_range_micro.pdf (Fig 2) in plots/ and map_range_micro.csv (Table 8) in csvs/ (both under scripts/).

Example of output corresponding to fig 2 in the paper:



map_range_micro.csv will have much more data than in the table in the paper because we truncated the data for space in the actual paper.

Table 2

Assuming you have run the parallel batch inserts for Figure 1, all that is left is to do the serial batch inserts.

Build/run serial batch inserts:

Go back to the top-level `Packed-Memory-Array` directory and rebuild the PMA in serial (without PARLAY=1 or CILK=1):

```
make build/basic_uint64_t_uncompressed_Eytzinger -B
```

Then go to scripts/ and run the serial batch inserts:

```
cd scripts/
sh run-table-2.sh
```

Make the table:

After running, check that `scripts/outputs/` now has `pma_serial_batch.out` in addition to `pma_parallel_batch.out` that we made during figure 1. Then make the table:

```
python3 make-table-2.py
```

You should end up with `pma_serial_parallel_batch.csv` in `scripts/csvs/` matching the format of Table 2 in the paper.

Figure 7 (and Table 10)

Build/run serial data point for PMA/CPMA:

From the top level `Packed-Memory-Array` directory, build and run the serial batch inserts (100M base with batch size 1M)

```
make build/basic_uint64_t_uncompressed_Eytzinger -B
make build/basic_uint64_t_delta_compressed_Eytzinger -B
cd scripts
sh run-serial-fig-7.sh
```

Check in `scripts/outputs` for two files: `pma_batch_scaling.out` and `cpma_batch_scaling.out`

Build/run parallel scalability points for PMA/CPMA:

From the top level `Packed-Memory-Array` directory, build the parallel versions and run the scalability experiments (100M base with batch size 1M):

```
make PARLAY=1 build/basic_uint64_t_uncompressed_Eytzinger
make PARLAY=1 build/basic_uint64_t_delta_compressed_Eytzinger
```

Note: you might have to modify the instructions in running the parallel scalability points and plotting/writing the csv if you have fewer than 128 cores. If you have fewer cores, edit the line in `scripts/run-parallel-fig-7.sh`:

```
for THREADS in 2 4 8 16 32 64 128
```

To match whatever thread count you have. For example, if you have 32 threads, the line would become:

```
for THREADS in 2 4 8 16 32
```

Once you have edited the script if necessary, run:

```
cd scripts
sh run-parallel-fig-7.sh
```

Make the plot and table:

Note: you might have to modify the instructions in running the parallel scalability points and plotting/writing the csv if you have fewer than 128 cores. If you have fewer cores, edit the line in `scripts/make-fig-7.py` that tells you how many data points you have:

```
num_entries = 8
```

To log(number of cores you have) + 1. For example, with 128 cores, lg(128) = 7, but +1 for the serial point. If you had 32 cores, that would be `num_entries = 6` because lg(32) = 5.

Once you have edited the script if necessary,

Make sure you are in scripts/, then run

```
python3 make-fig-7.py
```

Will output the plot in `batch_insert_scaling.pdf` (in `scripts/plots/`) and the csv in `batch_insert_scaling.csv` (in `scripts/csvs/`)

Example of output plot corresponding to Figure 7:

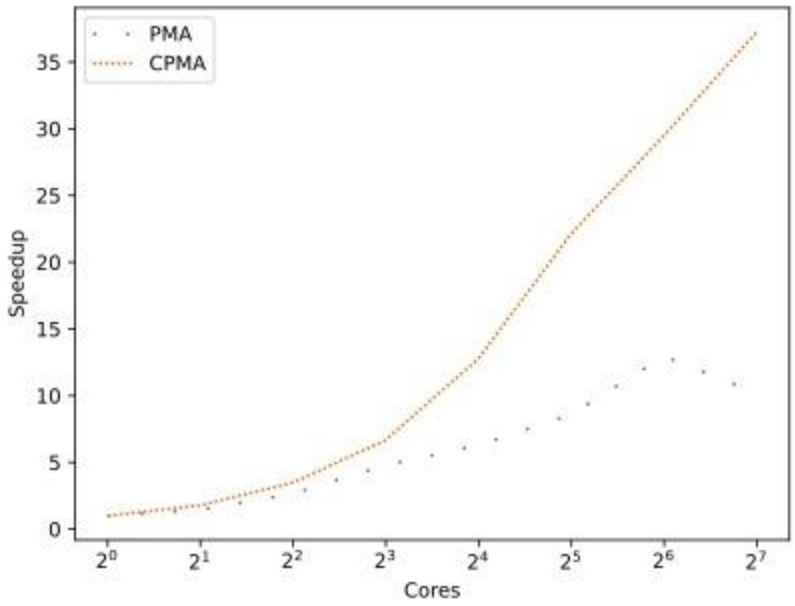


Figure 8 (and Table 11)

Build/run serial data point for PMA/CPMA:

From the top level `Packed-Memory-Array` directory, build and run the serial map range (100M base with range size 2^{34}):

```
make build/basic_uint64_t_uncompressed_Eytzinger -B
make build/basic_uint64_t_delta_compressed_Eytzinger -B
cd scripts
sh run-serial-fig-8.sh
```

Check in `scripts/outputs` for two files: `pma_map_range_scaling.out` and `cpma_map_range_scaling.out`

Build/run parallel scalability points for PMA/CPMA:

From the top level `Packed-Memory-Array` directory, build the parallel versions and run the scalability experiments:

```
make PARLAY=1 build/basic_uint64_t_uncompressed_Eytzinger
make PARLAY=1 build/basic_uint64_t_delta_compressed_Eytzinger
```

Note: you might have to modify the instructions in running the parallel scalability points and plotting/writing the csv if you have fewer than 128 cores. If you have fewer cores, edit the line in `scripts/run-parallel-fig-8.sh`:

```
for THREADS in 2 4 8 16 32 64 128
```

To match whatever thread count you have. For example, if you have 32 threads, the line would become:
```
for THREADS in 2 4 8 16 32
```

Once you have edited the script if necessary, run:

```
cd scripts
sh run-parallel-fig-8.sh
```

Make the plot and table:

Note: you might have to modify the instructions in running the parallel scalability points and plotting/writing the csv if you have fewer than 128 cores. If you have fewer cores, edit the line in `scripts/make-fig-8.py` that tells you how many data points you have:

```
num_entries = 8
```

To log(number of cores you have) + 1. For example, with 128 cores, lg(128) = 7, but +1 for the serial point. If you had 32 cores, that would be `num_entries = 6` because lg(32) = 5.

Once you have edited the script if necessary,

Make sure you are in scripts/, then run

```
python3 make-fig-8.py
```

Will output the plot (fig 8) in `map_range_scaling.pdf` (in `scripts/plots/`) and the csv (table 11) in `map_range_scaling.csv` (in `scripts/csvs/`)

Example of output plot corresponding to Figure 8:

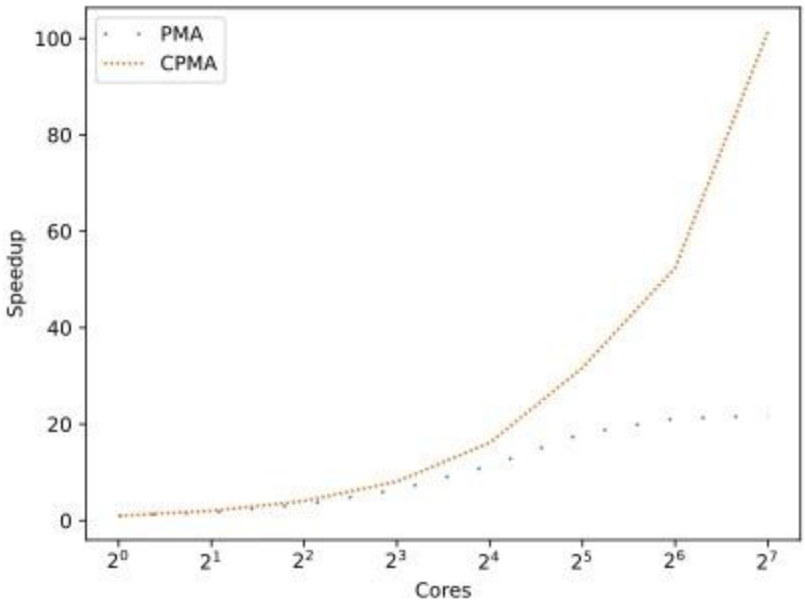


Table 3

The RMA experiments will take several hours since they are run in serial.

Requirements

The serial batch insert experiments for RMA (https://ieeexplore.ieee.org/document/8731468) require sudo access. If you do not have sudo, you can skip this part.

Build/run RMA

First, from the top-level `Packed-Memory-Array` directory, go to the RMA directory:
`cd other_systems/rma`

To build RMA, run
`sh compile_rma.sh`

Running the RMA requires changing around some settings about huge pages which can be done with the following commands run with sudo:

```
sudo echo 4294967296 > /proc/sys/vm/nr_overcommit_hugepages
sudo echo 1 > /proc/sys/vm/overcommit_memory
```

Then run the experiments with:
`sh batch_insert.sh`

Check that you got an output file at `rma_batch_insert.out` in `Packed-Memory-Array/other_systems/rma`

It will be in the form:
batch_size, num_batches, time
10000000, 10,18.496 seconds
10000000, 10,18.513 seconds
10000000, 10,18.468 seconds
10000000, 10,18.569 seconds
10000000, 10,18.317 seconds
10000000, 10,18.525 seconds
10000000, 10,18.548 seconds
10000000, 10,18.371 seconds
10000000, 10,18.278 seconds
10000000, 10,18.418 seconds
…

Make the table:

If you have not run the uncompressed PMA serial batch experiments, go to the instructions for making Table 2 and run them.

Once you have run the uncompressed serial batch PMA, go to `Packed-Memory-Array/scripts` and run:

```
sh make-table-3.sh
```

You should see a file called `batch_rma_pma.csv` in `scripts/csvs` corresponding to Table 3.

Table 5

Run PMA/CPMA

Make sure you have the parallel binaries for PMA and CPMA (see instructions for “Compile PMA/CPMA” at the top of this microbenchmarks section if you have not done so).

Then go to `Packed-Memory-Array/scripts` and run:
```
sh run-table-4.sh
```

In `scripts/outputs`, make sure you have `pma_micro_sizes.out` and `cpma_micro_sizes.out`

Run CPAM

Make sure you have the parallel binaries for UPAC and CPAC (see “Compile PAM/CPAM” under “Setup instructions” if you have not made them yet).

From the top-level `Packed-Memory-Array` directory, go to the microbenchmarks for CPAM:

```
cd other_systems/CPAM/examples/microbenchmarks
```

Run the size experiments:

```
sh run_table_4.sh
```

Make sure you have `upac_micro_sizes.out` and `upac_micro_sizes.out` in `other_systems/CPAM/examples/microbenchmarks/outputs`

Make the table:

After running the size experiments, from the scripts directory (`Packed-Memory-Array/scripts`), make the table:

```
python3 make-table-4.py
```

In `scripts/csvs`, you should see `size_micro.csv` to match Table 4.

Graph evaluation:

Download datasets:

The space requirements for the graphs are here:

graph	size	Source vertex for algorithms
lj.adj	640MB	0
co.adj	1.68GB	1000
er.adj	7.44GB	0
tw.adj	20.56GB	12
fs.adj	31.4GB	100000

Before downloading the graphs, make sure you have enough space to store them. If not, you can download a subset of the graphs and then run the graph evaluation, but need to edit the scripts (ie, download and run scripts) to comment out the graphs you don't have.

Starting from the top-level `Packed-Memory-Array/` directory:

```
cd graphs
sh download-graphs.sh
```

After it is done, make sure you have the 5 graphs: LJ, CO, ER, TW, and FS under `Packed-Memory-Array/graphs`

File format:

The graph file format is .adj described at https://www.cs.cmu.edu/~pbbs/benchmarks/graphIO.html. For utilities including converters from other file formats see https://github.com/jshun/ligra/tree/master.

Compile data structures:

To run the graph evaluation with CPMA, just build the parallel binary as before from the `Packed-Memory-Array` directory:

```
make PARLAY=1 build/basic_uint64_t_delta_compressed_Eytzinger
```

Figure 9 (and Table 12), Figure 10 (and Table 13), Table 6:

Run F-Graph (CPMA):

Once you have downloaded the graphs, start from the `Packed-Memory-Array` directory:

```
cd scripts
sh run-graph-eval.sh
```

This will run the algorithms, sizes, and batch inserts (inserts are only on the FS graph).

Check that you have 5 output files (one for each of the graphs) in `Packed-Memory-Array/scripts/outputs/graph-eval/`: `lj.out`, `co.out`, etc

Run C-PaC (CPAM):

To run the graph algorithms and sizes for C-PaC (after you have downloaded the graphs with the instructions above), start from the `Packed-Memory-Array` directory:

```
cd other_systems/CPAM/examples/graphs/algorithms
make
bash run_all.sh
```

Check for 5 output files with
`ls outputs`

To run the graph batch updates, starting from the top-level Packed-Memory-Array directory,

```
cd other_systems/CPAM/examples/graphs/run_batch_updates
```

Alternatively, if you are already in
`Packed-Memory-Array/other_systems/CPAM/examples/graphs/algorithms`,
`cd ../run_batch_updates`

Then build/run the test with
`make`
`sh run_updates.sh`

Make sure there is a file `batch_inserts.out` in
`other_systems/CPAM/examples/graphs/run_batch_updates/outputs`

Run Aspen:

Start from the `Packed-Memory-Array` directory:

```
cd other_systems/aspen/code/
make run_static_algorithm run_batch_updates memory_footprint
bash run-all.sh
```

Which will run algorithms, batch inserts, and sizes. Check that there are outputs in `other_systems/aspen/code/outputs`.

Make the plots/table:

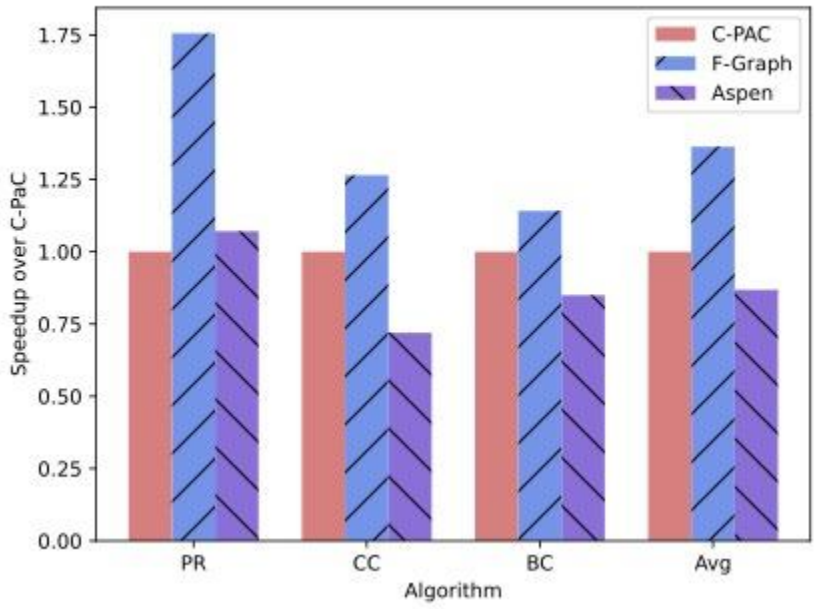
Starting from the top-level Packed-Memory-Array directory,

```
cd scripts
python3 make-graph-figs.py
```

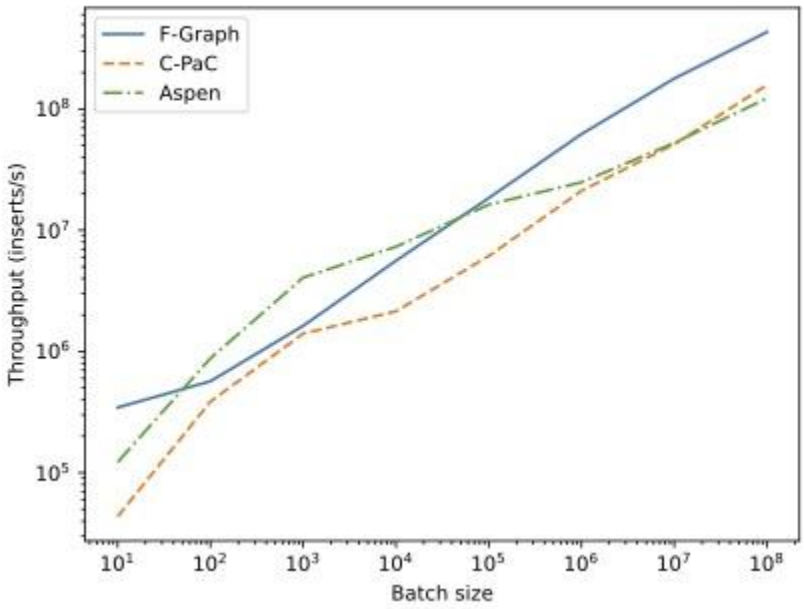
In `scripts/plots`, you should now have `graph_algs.pdf` (Figure 9) and `graph_batch_insert.pdf` (Figure 10). In `scripts/csvs`, you will find the corresponding tables `graph_algs.csvs` (Table 14) and `graph_batch_insert.csv` (Table 12).

In `scripts/csvs`, you should also have `graph_sizes.csv` (Table 6).

Example of graph_algs.pdf:



Example of graph_batch_insert.pdf:



PMA API for reusability

To use the PMA/CPMA for other purposes, include it as a header-only library

The library itself is header only, so you can just #include "CPMA.h".
You will need to download the submodules with
`git submodule init`
`git submodule update`

The PMA/CPMA supports the following API:

- `uint64_t size()` : The number of elements being stored in the PMA
- `CPMA()`: construct an empty PMA
- `CPMA(key_type *start, key_type *end)`: construct a PMA with the elements in the given range
- `bool has(key_type e)`: return true if the key e is in the PMA
- `bool insert(element_type e)`: inserts the element e into the PMA, returns false if the key was already there.
- `uint64_t insert_batch(element_ptr_type e, uint64_t batch_size, bool sorted = false)`: inserts a batch of elements of size `batch_size`
- `uint64_t remove_batch(key_type *e, uint64_t batch_size, bool sorted = false)`: removes a batch of elements
- `bool remove(key_type e)`: removes the element with key e
- `uint64_t get_size()`: returns the amount of memory in bytes used by the PMA
- `uint64_t sum()`: Returns the sum of all elements in the PMA
- `key_type max() / min()`: returns the smallest or largest key stored in the PMA.
- `bool map(F f)`: runs function `f` on all elements in the PMA
- `parallel_map(F f)`: runs function `f` on all elements in the pma in parallel
- `bool map_range(F f, key_type start_key, key_type end_key)`: runs function `f` on all elements with keys between `start_key` and `end_key`.
- `uint64_t map_range_length(F f, key_type start, uint64_t length)`: runs function `f` on at most `length` elements starting from key at least `start`
- The PMA also supports iteration as it has begin and end functions so you can perform operations like for (auto el : pma). Note that this may be slower than using the map functions

For some examples of different uses you can look in include/PMA/internal/test.hpp
Here we have many different uses of the PMA that can easily be modified to exercise different characteristics of the data structure. After changing test.hpp rebuild with the specified parameters as described above.