PaC-trees: Supporting Parallel and Compressed Purely-Functional Collections

Laxman Dhulipala¹, Guy Blelloch², Yan Gu³, Yihan Sun³

¹University of Maryland, ²Carnegie Mellon University, ³UC Riverside

Artifacts available and reusable!
Library available on GitHub: https://github.com/ParAlg/CPAM
Collection Data Types [sequences, sets, maps]

• A collection of data [e.g., sequences, ordered sets, ordered maps]
• Very commonly-used in programming!
• E.g., in C++ STL: vector, (ordered) set, (ordered) map.
  • Similar in other languages

what is the most frequently used stl class

About 312,000,000 results (0.60 seconds)

The most commonly used features of STL are:

• Iterator.
• Vector. [sequence]
• Stack.
• Queue.
• Priority Queue:
  • Map.
  • Set.
  • Pair.

Search tree
Collection for inverted index

• Collection of words, each mapping to a collection of documents

**Document 1:**
The largest blue whale ever recorded had a length from head to tail of 110 feet and 17 inches.

**Document 2:**
World’s largest blue diamond to come to auction has sold for $57.5 million.

**Document 3:**
Banging your head against a wall for one hour burns 150 calories.

<table>
<thead>
<tr>
<th>Word</th>
<th>Document list</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>blue</td>
<td>1, 2</td>
</tr>
<tr>
<td>whale</td>
<td>1</td>
</tr>
<tr>
<td>largest</td>
<td>1, 2</td>
</tr>
<tr>
<td>calories</td>
<td>3</td>
</tr>
<tr>
<td>diamond</td>
<td>2</td>
</tr>
<tr>
<td>head</td>
<td>1, 3</td>
</tr>
<tr>
<td>million</td>
<td>2</td>
</tr>
</tbody>
</table>
Collection for inverted index

• Collection of words, each mapping to a collection of documents

<table>
<thead>
<tr>
<th>Document 1:</th>
<th>The largest blue whale ever recorded had a length from head to tail of 110 feet and 17 inches.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document 2:</td>
<td>World’s largest blue diamond to come to auction has sold for $57.5 million.</td>
</tr>
<tr>
<td>Document 3:</td>
<td>Banging your head against a wall for one hour burns 150 calories.</td>
</tr>
<tr>
<td>Document 4:</td>
<td>Blue whales eat half a million calories in one mouthful.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word</th>
<th>Document list</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>blue</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td>whale</td>
<td>1, 4</td>
</tr>
<tr>
<td>largest</td>
<td>1, 2</td>
</tr>
<tr>
<td>calories</td>
<td>3, 4</td>
</tr>
<tr>
<td>diamond</td>
<td>2</td>
</tr>
<tr>
<td>head</td>
<td>1, 3</td>
</tr>
<tr>
<td>million</td>
<td>2, 4</td>
</tr>
<tr>
<td>eat</td>
<td>4</td>
</tr>
<tr>
<td>mouthful</td>
<td>4</td>
</tr>
</tbody>
</table>
Collection for graph processing

• A collection of vertices, each mapping to a collection of edges
Collection for geometric queries

• A collection of points in 1D or 2D
• Find all points in a certain range

$(a, b)$

$(c, d)$
Collection Data Types [sequences, sets, maps] In parallel?

- [Goal 1] Full interface: as needed in the applications!

<table>
<thead>
<tr>
<th>Point updates/queries</th>
<th>Bulk updates/queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>build flatten</td>
</tr>
<tr>
<td>next/previous</td>
<td>map reduce filter</td>
</tr>
<tr>
<td>rank/n-th</td>
<td>range append reverse</td>
</tr>
<tr>
<td>first/last</td>
<td>multi-insert/multi-delete</td>
</tr>
<tr>
<td>insert/delete</td>
<td>union/intersection/difference</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>
Collection Data Types [sequences, sets, maps]

In parallel?

• [Goal 1] Full interface: as needed in the applications!

• [Goal 2] Concurrency: Multiple threads can work on the same data structure safely and correctly
  • Functional data structure! [immutable]
  • Each thread works on a snapshot
  • Used in many existing parallel languages/libraries [friendly for parallelism]

• [Goal 3] Parallelism: Bulk operations in parallel

<table>
<thead>
<tr>
<th>Bulk updates/queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
</tr>
<tr>
<td>map</td>
</tr>
<tr>
<td>range</td>
</tr>
<tr>
<td>multi-insert/multi-delete</td>
</tr>
<tr>
<td>union/intersection/difference</td>
</tr>
<tr>
<td>......</td>
</tr>
</tbody>
</table>
**P-trees** [Sun et al., PPoPP’17] for parallel collections

- Parallel binary search trees P-tree in the PAM library
- Functional data structure using path-copying
  - Standard way in functional languages
- General interface for collections: appliable in many applications
P-trees for parallel collections have large space overhead!

[Goal 4] Space-efficiency: avoid high space overhead!
Our PaC-tree and CPAM library

- full interface of sequences, ordered sets, ordered maps applicable to a wide range of applications
- functional/immutable
- highly-parallel
- fast both in theory and in practice
- space-efficient!
How to Be Space-Efficient? Put More Data in One Node?

• Multi-way search trees, such as functional B-tree?
• 😞 path-copying is expensive
Put More Data in One Node But Keep the Tree Binary!

C-tree in Aspen [Dhulipala et al., PLDI’19]

• Aspen: a graph processing library
• Binary trees with multiple entries in a tree node
• Separate the first entry (called head) for copying
• Design for maintaining edges in graphs, not for general collections

C-tree in Aspen (Compressing nodes in BST)
Keep the Tree Binary
But Put More Data Only in **LEAVES**!
Our new **PaC-tree**

- **[Balance invariant]** Weight-balanced: left/right subtree sizes differ within a constant factor
- **[Blocking invariant]** Any subtree of size $B$ to $2B$ will be blocked
- The blocks can be further compressed
- We use **delta encoding**: store the difference relative to the previous

<table>
<thead>
<tr>
<th>Data:</th>
<th>17</th>
<th>19</th>
<th>24</th>
<th>24</th>
<th>29</th>
<th>33</th>
<th>42</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>↓</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
</tr>
<tr>
<td>Encoded data:</td>
<td>17</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

Pac-tree of size 14, $B=2$
Keep the Tree Binary But Put More Data Only in **LEAVES**! Our new PaC-tree

- **[Balance invariant]** Weight-balanced: left/right subtree sizes differ within a constant factor
- **[Blocking invariant]** Any subtree of size $B$ to $2B$ will be blocked

---

**PaC-trees:**
- Low space usage
- Parallel and efficient algorithms

(Our new) PaC-tree (Compressing leaves in BST)
PaC-tree – Space Bounds

**PaC-trees:**

✓ Low space usage

? Parallel and efficient algorithms

---

**Theorem.** The total space of a PaC-tree with block size $B + \delta$ delta encoding, on a set $E$ of $n$ integer keys is:

$$s(E) + O(n/B + B)$$

$s(E) = $ the space to store $E$ in an array using delta encoding [lower bound]
Extended Join-based framework in PAM

- The function “join” is a **black box** – all other algorithms are based on “join”
- Path-copying: just copy a few nodes in join
- \( T = \text{Join}(T_L, e, T_R) \) : \( T_L \) and \( T_R \) are two trees, \( e \) is an entry.
- \( T_L < e < T_R \)
- Returns a valid tree \( T = T_L \cup \{e\} \cup T_R \)

\[ T = \begin{array}{c}
T_L \\
\vdots \\
T_R
\end{array} \]

(Rebalance if necessary)
Extended Join-based framework in PAM

- How to extend the algorithms to PaC-trees?
- Deal with the blocks?
- Add a primitive expose(T), returns a “left child”, a “root” and a “right child”

![Diagram showing an example of expose operation on a PaC-tree]

- We carefully redesigned “join” and “expose” abstractions, and keep the high-level algorithmic ideas in PAM unchanged!
- Keep blocking invariant true all the time!
Example: combining two trees

union($T_1, T_2$)

if $T_1 = \emptyset$ then return $T_2$
if $T_2 = \emptyset$ then return $T_1$

$(L_2, k_2, R_2) = \text{expose}(T_2)$
$(L_1, b, R_1) = \text{split}(T_1, k_2)$

*In parallel:*

$T_L = \text{Union}(L_1, L_2)$
$T_R = \text{Union}(R_1, R_2)$

return $\text{Join}(T_L, k_2, T_R)$

(example for in-place updates. Functional updates can be performed by copying corresponding nodes in the join algorithm.)
Example: combining two trees

union($T_1, T_2$)

if $T_1 = \emptyset$ then return $T_2$

if $T_2 = \emptyset$ then return $T_1$

$(L_2, k_2, R_2) = \text{expose}(T_2)$

$(L_1, b, R_1) = \text{split}(T_1, k_2)$

\textbf{In parallel:}

$T_L = \text{Union}(L_1, L_2)$

$T_R = \text{Union}(R_1, R_2)$

return $\text{Join}(T_L, k_2, T_R)$

(example for \textit{in-place} updates. Functional updates can be performed by copying corresponding nodes in the join algorithm.)
Example: combining two trees

union($T_1, T_2$)

if $T_1 = \emptyset$ then return $T_2$
if $T_2 = \emptyset$ then return $T_1$

$(L_2, k_2, R_2) = \text{expose}(T_2)$
$(L_1, b, R_1) = \text{split}(T_1, k_2)$

In parallel:

$T_L = \text{Union}(L_1, L_2)$
$T_R = \text{Union}(R_1, R_2)$

return $\text{Join}(T_L, k_2, T_R)$

(example for in-place updates. Functional updates can be performed by copying corresponding nodes in the join algorithm.)
Example: combining two trees

union($T_1, T_2$)

if $T_1 = \emptyset$ then return $T_2$
if $T_2 = \emptyset$ then return $T_1$

($L_2, k_2, R_2$) = expose($T_2$)
($L_1, b, R_1$) = split($T_1, k_2$)

In parallel:
$T_L$ = Union($L_1, L_2$)
$T_R$ = Union($R_1, R_2$)

return Join($T_L, k_2, T_R$)

(Example for in-place updates. Functional updates can be performed by copying corresponding nodes in the join algorithm.)
Example: combining two trees

union($T_1, T_2$)

if $T_1 = \emptyset$ then return $T_2$
if $T_2 = \emptyset$ then return $T_1$

$(L_2, k_2, R_2) = \text{expose}(T_2)$
$(L_1, b, R_1) = \text{split}(T_1, k_2)$

**In parallel:**

$T_L = \text{Union}(L_1, L_2)$
$T_R = \text{Union}(R_1, R_2)$

return $\text{Join}(T_L, k_2, T_R)$

(example for in-place updates. Functional updates can be performed by copying corresponding nodes in the join algorithm.)
**Example: combining two trees**

union($T_1, T_2$)
- if $T_1 = \emptyset$ then return $T_2$
- if $T_2 = \emptyset$ then return $T_1$
- $(L_2, k_2, R_2) = \text{expose}(T_2)$
- $(L_1, b, R_1) = \text{split}(T_1, k_2)$

*In parallel:*
- $T_L = \text{Union}(L_1, L_2)$
- $T_R = \text{Union}(R_1, R_2)$
- **return** $\text{Join}(T_L, k_2, T_R)$

(example for *in-place* updates. Functional updates can be performed by copying corresponding nodes in the join algorithm.)

(Theoretical guarantees are provided in the paper)
Lots of Functions and Applications Supported

• **Functions supported**
  - Sequences: Build, map, filter, reduce, take, n-th, findFirst, append, reverse
  - Ordered set and map: (most functions for sequences), next, previous, rank, range, insert, union, intersection, difference, …
  - All of them have theoretical bounds

• **Applications:**
  - 1D interval queries
  - 2D range queries
  - Inverted indexes
  - Graph processing
Experiments

- 72-core Dell PowerEdge R930 (with two-way hyper-threading)
- 1TB of main memory
- Using C++ and the work-stealing scheduler from Parlaylib
Microbenchmarks, compared to P-trees (PAM)
(Functional tree, no blocking leaves or compression)

- PaC-tree (no encoding)
  [1.61GB] 2.5x saving

- PaC-tree (encoded)
  [0.93GB] 4.3x saving

- P-tree (PAM)
  [4.00GB]

Input size $n = 10^8$, block size $B = 128$, 64bit-64bit key-values
Microbenchmarks, compared to P-trees (PAM)

Tradeoff of blocking + encoding
- may improve performance because of smaller memory footprint + I/O friendliness
- can also cause overhead due to encoding/decoding

PaC-trees achieve similar or better time on most tested functions, while being 2–4 times more space-efficient than P-trees in PAM

Input size $n = 10^8$,
block size $B = 128$
64bit-64bit key-values (Lower is better)
PaC-trees applied to graphs, compared to C-trees (Aspen) (Functional tree, blocking all tree nodes, specifically for edges in graphs)

Running time/memory relative to the best

PaC-tree is almost always faster than Aspen on all benchmarks and graphs
PaC-tree is also 1.2-1.5x more space efficient than Aspen

Both PaC-tree and Aspen use delta encoding
(Lower is better)
More experiments

- Performance vs. block size
- Space vs. block size
- Inverted indices
- interval tree
- 2D range tree
- graph streaming

- Some of them also requires augmentation, see more details in the paper.
Summary

• PaC-Tree
  • Blocked leaves, can be further encoded
  • Provable guarantee in both space and time
  • Safe and efficient for parallelism

• CPAM library
  • Full interface for collection for a wide range of applications
  • Outperforms previous non-compressed data structure for collections (P-trees), and more space-efficient!
  • Outperforms previous compressed data structure for certain applications (C-trees for graph processing) and more space-efficient!

Artifacts available and reusable!
Library available on GitHub: https://github.com/ParAlg/CPAM