Flat Combining Synchronized Global Data Structures

Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin

contention $\rightarrow$ cooperation

simple, distributed, batched synchronization

sequential consistency at cluster scale

© Disney, Inc. Fantasia (The Pastoral Symphony)
synchronized shared data structures

Standard library aids *productivity*

*Generality* costs performance/scalability
synchronized shared data structures

Standard library aids **productivity**

**Generality** costs performance/scalability

Must maintain **consistency**
synchronized shared data structures

Standard library aids **productivity**

**Generality** costs performance/scalability

Must maintain **consistency**
synchronized shared data structures

Standard library aids **productivity**

**Generality** costs performance/scalability

Must maintain **consistency**

more **concurrency** → more **contention**
contention → cooperation
contention $\rightarrow$ cooperation
contention → cooperation
contention $\rightarrow$ cooperation
contention: global lock

Contention causes **failed lock acquires** (typically compare-and-swaps)

**Retries** consume bandwidth

Sharing causes cache traffic/thrashing
contention: fine-grained sync

Complicated schemes are error-prone
Still failed compare-and-swaps and **retries**
Same result: **serialized** access
cooperation: flat combining

[1] “Flat Combining and the Synchronization-Parallelism Tradeoff”
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir
(SPAA ’10)
cooperation: flat combining

Cooperation via publication list
One combiner does all the work

[1] “Flat Combining and the Synchronization-Parallelism Tradeoff”
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir
(SPAA ’10)
cooperation: flat combining

Cooperation via publication list

One combiner does all the work

Publication List

Master

Stack

Thread 1 push( )
Thread 2 pop( )
Thread 3 push( )
Thread 4 push( )

Publication Record
4
Publication Record
_ (null)
Publication Record
8
Publication Record
7

[1] “Flat Combining and the Synchronization-Parallelism Tradeoff” Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir (SPAA '10)
cooperation: flat combining

Cooperation via publication list
One combiner does all the work

[1] “Flat Combining and the Synchronization-Parallelism Tradeoff”
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir
(SPAA ’10)
cooperation: flat combining

Cooperation via publication list

One combiner does all the work

[1] “Flat Combining and the Synchronization-Parallelism Tradeoff” Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir (SPAA ’10)
cooperation: flat combining

Cooperation via publication list
One combiner does all the work

[1] “Flat Combining and the Synchronization-Parallelism Tradeoff”
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir
(SPAA ’10)
cooperation: flat combining

Cooperation via publication list
One combiner does all the work

[1] “Flat Combining and the Synchronization-Parallelism Tradeoff”
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir
(SPAA ’10)
cooperation: flat combining

Cooperation via publication list
One combiner does all the work

[1] “Flat Combining and the Synchronization-Parallelism Tradeoff”
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir
(SPAA ’10)
cooperation: flat combining

Cooperation via publication list
One combiner does all the work

[1] “Flat Combining and the Synchronization-Parallelism Tradeoff”
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir
(SPAA ’10)
Flat combining in multicore

Simple locking scheme, but maximum of 1 failed CAS per thread
- beats combining trees and funnels
- beats fine-grained synchronization

Applicable if combined ops are faster than individually, due to:
- cache locality
- shared traversal (e.g. some linked list)
- better sequential algorithm
  (priority queue: pairing heap vs. skiplist)

Flat combining in PGAS

**Distributed** synchronization
- reduce serialization on global lock
- avoid making operations globally visible if possible

<table>
<thead>
<tr>
<th>Core</th>
<th>Worker</th>
<th>s-&gt;push(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Worker</td>
<td>s-&gt;push(8)</td>
</tr>
<tr>
<td>Core</td>
<td>Worker</td>
<td>s-&gt;pop()</td>
</tr>
<tr>
<td>Core</td>
<td>Worker</td>
<td>s-&gt;push(4)</td>
</tr>
</tbody>
</table>

Node 0

Global Stack

Node N

Global heap

Master
top

Network

storage

56 13 7
Flat combining in PGAS

Distributed synchronization
- reduce serialization on global lock
- avoid making operations globally visible if possible

Combining structure: local proxy
- calls operate on this instead
- resolve locally if possible
Flat combining in PGAS

**Distributed** synchronization
- reduce serialization on global lock
- avoid making operations globally visible if possible

Combining structure: local **proxy**
- calls operate on this instead
- resolve locally if possible

One worker commits combined op
- progress guarantee: always one in flight per core
Flat combining in PGAS
Flat combining in PGAS

Workers operate on local proxy – resolve locally where possible

Core
Worker s->push(7)
Worker s->push(8)
Worker s->pop()
Worker s->push(4)

Core

Proxy
buffer push_count

Network

Global Stack

Core
Master
top

Global heap

Workers operate on local proxy – resolve locally where possible.
Flat combining in PGAS

Workers operate on local proxy – resolve locally where possible
Flat combining in PGAS

Workers operate on local proxy – resolve locally where possible
Flat combining in PGAS

Workers operate on local proxy – resolve locally where possible

```
Core
Worker s->push()
Worker s->push()
Worker s->pop()
Worker s->push(4)
...```

```
Node 0
Core ... Core
Proxy
Master top
42 13 7
...```

```
Node N
Core ... Core
Proxy
Global Stack
storage```

```
Network
Proxy
Proxy
Proxy
Proxy
Global heap```

Brandon Holt – PGAS ’13 – 3-4 Oct 2013
Flat combining in PGAS

Workers operate on local proxy – resolve locally where possible
Flat combining in PGAS

Workers operate on local proxy
– resolve locally where possible

One worker becomes **combiner**:
– freeze current Proxy, create fresh one for next round
– globally commit
– wake blocked workers when finished
– trigger next Proxy to go
Flat combining in PGAS

Workers operate on local proxy
– resolve locally where possible

One worker becomes combiner:
– freeze current Proxy, create fresh one for next round
– globally commit
– wake blocked workers when finished
– trigger next Proxy to go
Flat combining in PGAS

Workers operate on local proxy
– resolve locally where possible

One worker becomes combiner:
– freeze current Proxy, create fresh one for next round
– globally commit
– wake blocked workers when finished
– trigger next Proxy to go
Flat combining in PGAS

Workers operate on local proxy
– resolve locally where possible

One worker becomes **combiner:**
– freeze current Proxy, create fresh one for next round
– globally commit
– wake blocked workers when finished
– trigger next Proxy to go
Flat combining in PGAS

Sequential Consistency

C++ model: SC for Data-Race-Free

Enforcing linearizability:
– ensure program order by blocking thread until globally committed
– globally- and locally-observable order must coincide
Flat combining in PGAS

Sequential Consistency

C++ model: SC for Data-Race-Free

Enforcing linearizability:
- ensure program order by blocking thread until globally committed
- globally- and locally-observable order must coincide

GlobalStack

push/pop annihilate each other, can be anywhere in global order
Flat combining in PGAS

**Sequential Consistency**

C++ model: SC for Data-Race-Free

Enforcing linearizability:
- ensure program order by blocking thread until globally committed
- globally- and locally-observable order must coincide
Flat combining in PGAS

**Sequential Consistency**

C++ model: SC for Data-Race-Free

**Enforcing linearizability:**
- ensure program order by blocking thread until globally committed
- globally- and locally-observable order must coincide

**GlobalSet/GlobalMap**
- insert/lookup must preserve order
- cheaper to disallow local lookups
Grappa: a latency-tolerant PGAS runtime
Grappa: a latency-tolerant PGAS runtime

Global-view, single system image

Core 0
- Worker main()
- Worker
- Worker
- Worker

Core C
- Worker read()
- Worker write()
- Worker calc()
- Worker push()

Node 0
- Core
- Core
- Core
- Core

Node N
- Core
- Core
- Core
- Core

Tasks
- work()

Memory

Global Heap

Aggregation buffer

Network
Grappa: a latency-tolerant PGAS runtime

Global-view, single system image

C++11 library interface

using namespace Grappa;

void grappa_main() {
    auto array = global_alloc<int>(N);

    forall_global(0, N, [=](int i) {
        auto val = delegate::read(array+i);
        if (val == 0) {
            delegate::call((array+i).core(), [=] {
                // ...
            });
        }
    });
}
Grappa: a latency-tolerant PGAS runtime

Global-view, single system image

C++11 library interface

Aggregated communication
Grappa: a latency-tolerant PGAS runtime

Global-view, single system image

C++11 library interface

Aggregated communication

Lightweight user-level threads for latency tolerance
Grappa: a latency-tolerant PGAS runtime

Global-view, single system image

C++11 library interface

*Aggregated* communication

Lightweight user-level threads for *latency tolerance*

Access other cores’ data only via *delegate operations*
**Grappa**: a latency-tolerant PGAS runtime

Global-view, single system image

C++11 library interface

*Aggregated* communication

Lightweight user-level threads for *latency tolerance*

Access other cores’ data only via *delegate operations*
Grappa: a latency-tolerant PGAS runtime

Global-view, single system image

C++11 library interface

Aggregated communication

Lightweight user-level threads for latency tolerance

Access other cores’ data only via delegate operations

Atomicity due to cooperative scheduling & delegates
Flat combining in Grappa
Flat combining in Grappa

Massive multithreading
- many workers, lots of combining
- lightweight suspend/wake

Synchronizing with Proxy is free
- **cooperative multithreading** within core
- only access other cores’ memory via delegate ops
Flat combining framework
Flat combining framework

Reusable logic:

- manage freezing and creating fresh Proxys
- ensure always one combiner
- handle delivering results to blocked workers and waking them
Flat combining framework

Reusable logic:

- manage freezing and creating fresh Proxys
- ensure always one combiner
- handle delivering results to blocked workers and waking them

Each data structure defines:

- Proxy structure
- ops that combine into local proxy
- sync op that globally commits proxy’s state

```cpp
template< class T >
class GlobalStack {

  class Proxy: FCProxy< T > { 
    // Local state for tracking requests
    T  pushed_values[1024];
    T* popped_results[1024];
    int npush, npop;

    // combining ops
    void push(T val);
    T pop();

    void sync() override; // commit globally
  };

};
```
Flat combining framework

Reusable logic:
- manage freezing and creating fresh Proxys
- ensure always one combiner
- handle delivering results to blocked workers

Each data structure:
- Proxy structure
- ops that combine into local proxy
- sync op that globally commits proxy’s state

Implemented (so far):
- GlobalStack, GlobalQueue
- GlobalHashSet, GlobalHashMap

```cpp
template< class T >
class GlobalStack {
    class Proxy: FCProxy< T > {
        // Local state for tracking requests
        T pushed_values[1024];
        T* popped_results[1024];
        int npush, npop;
        
        void sync() override;
        // commit globally
    };

    // Local state for tracking requests
    T pushed_values[1024];
    T* popped_results[1024];
    int npush, npop;

    void push(T val);
    T pop();

    void sync();
    // commit globally
};
```
Flat combining performance evaluation

**Experimental setup**

- Run on the PIC cluster at Pacific Northwest National Lab (PNNL)
- AMD Interlagos 2.1 GHz, 40 Gb Infiniband (Mellanox Connect-X 2, with QLogic switch)
- 16 cores per node, 2048 workers per core
Flat combining performance evaluation

**Methodology**

Random throughput workload
- With/without flat combining
- Varied operation mix (push/pop, lookup/insert)

**Experimental setup**
- Run on the PIC cluster at Pacific Northwest National Lab (PNNL)
- AMD Interlagos 2.1 GHz, 40 Gb Infiniband (Mellanox Connect-X 2, with QLogic switch)
- 16 cores per node, 2048 workers per core

```c
void test(GlobalAddress<GlobalStack<long>> stack) {
    forall_global(0, 1<<28, [=](long i){
        if (choose_random(push_mix)) {
            stack->push(next_random<long>());
        } else {
            stack->pop();
        }
    });
}
```
Flat combining
performance evaluation
Flat combining
performance evaluation

Throughput (millions of ops/sec)

GlobalQueue

GlobalStack

Nodes

8 16 32 48 64

8 16 32 48 64

Flat Combining
- distributed
- none

Operation Mix
- 100% push
- 50% push, 50% pop
Flat combining performance evaluation

![Graph showing performance evaluation of GlobalHashMap and GlobalHashSet].

- **GlobalHashMap** and **GlobalHashSet** are compared across different numbers of nodes (8, 16, 32, 48, 64).
- **Throughput** (millions of ops/sec) is measured for **Flat Combining**.
- **Distributed** and **none** are represented by different lines.
- **Operation Mix** includes 100% insert, 50% insert, 50% lookup.
- **Keys** range from $0$ to $2^{10}$ and $2^{14}$.

---

Brandon Holt – PGAS ‘13 – 3-4 Oct 2013
Flat combining performance evaluation

**Experimental setup** (same)

- Run on the PIC cluster at Pacific Northwest National Lab (PNNL) (AMD Interlagos 2.1 GHz, 40 Gb Mellanox InfiniBand)
- 16 cores per node, 2048 workers per core
Flat combining performance evaluation

Methodology: Apps
- Scale 26 Graph500-spec graph (64 M vertices, 1 B edges)
- Breadth First Search benchmark (find parent tree from random root)
- Connected Components (using 3-phase algorithm)

Experimental setup (same)
- Run on the PIC cluster at Pacific Northwest National Lab (PNNL) (AMD Interlagos 2.1 GHz, 40 Gb Mellanox InfiniBand)
- 16 cores per node, 2048 workers per core
Flat combining
performance evaluation
Flat combining
performance evaluation

Breadth First Search

- MTEPS vs Nodes

Connected Components

- MTEPS vs Nodes

Flat Combining
- custom
- distributed
- none
Flat Combining Synchronized Global Data Structures

take-aways

Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin
Flat Combining Synchronized Global Data Structures take-aways

massive concurrency enables sequential consistency at scale thanks to Grappa’s latency tolerance

lesson learned from multicore: cooperation beats contention

add new data structures easily with flat-combining framework

Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin
Flat Combining
Synchronized
Global Data Structures

take-aways

massive concurrency enables
sequential consistency at scale
thanks to Grappa’s latency
tolerance

add new data structures easily
with flat-combining framework

questions?

Brandon Holt, Jacob Nelson, Brandon Myers,
Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin
© Disney, Inc. Fantasia (The Pastoral Symphony)
Thank you!

© Disney, Inc. Fantasia (The Pastoral Symphony)
Flat combining performance

![Graph showing throughput (millions of ops/sec) for GlobalQueue and GlobalStack with different nodes and operation mixes.]
Flat combining performance

- random throughput workload
- 2048 workers per core
- 16 cores per node

<table>
<thead>
<tr>
<th>Nodes</th>
<th>GlobalQueue</th>
<th>GlobalStack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput (millions of ops/sec)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>32</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>48</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>64</td>
<td>10000</td>
<td>10000</td>
</tr>
</tbody>
</table>

Flat Combining
- centralized
- distributed
- none

Operation Mix
- 100% push
- 50% push, 50% pop