Memory Checking for Parallel RAMs

Surya Mathialagan
MIT
Quick Overview
Quick Overview

- We define a new notion of memory checking in the parallel RAM model.
Quick Overview

• We define a new notion of memory checking in the parallel RAM model.

• We construct memory checkers for PRAMs matching the asymptotic efficiency of memory checkers for the RAM setting, while achieving optimal parallel depth.
Quick Overview

• We define a new notion of memory checking in the parallel RAM model.

• We construct memory checkers for PRAMs matching the asymptotic efficiency of memory checkers for the RAM setting, while achieving optimal parallel depth.

• As an application, we construct maliciously secure Oblivious Parallel RAM with polylog overhead.
Integrity of Cloud Servers
Integrity of Cloud Servers
Integrity of Cloud Servers

Database D

read(3)

D[3]
Integrity of Cloud Servers

![Diagram showing user accessing database with read(3) and read(5) operations. A devil representing an attack is also shown.]
Integrity of Cloud Servers

Database D

Integrity of Cloud Servers

• How can a client use her **small** but **trusted** local memory to ensure that server is sending back correct responses?
Integrity of Cloud Servers

- How can a client use her small but trusted local memory to ensure that server is sending back correct responses?

- Answer: Authentication
Integrity of Cloud Servers

• How can a client use her small but trusted local memory to ensure that server is sending back correct responses?

• Answer: Authentication… if the database is static
Integrity of Cloud Servers

read(3)
data := D[3]

Database D
Integrity of Cloud Servers

• What if the database contents are *dynamically* updated?
Integrity of Cloud Servers

• What if the database contents are **dynamically** updated?
Integrity of Cloud Servers

• What if the database contents are **dynamically** updated?
Integrity of Cloud Servers

- What if the database contents are **dynamically** updated?

- We want reads to correspond to most recent version! (i.e. $data_{\text{new}}$ not $data$)
Memory Checking

[Blum, Evans, Gemmel, Kannan, Naor ’94]
Memory Checking

[Blum, Evans, Gemmel, Kannan, Naor ’94]
Memory Checking

[Blum, Evans, Gemmel, Kannan, Naor '94]
Memory Checking

[Blum, Evans, Gemmel, Kannan, Naor ’94]
**Memory Checking**

[Blum, Evans, Gemmel, Kannan, Naor ’94]

- **Correctness**: For any PPT malicious server, MC either aborts or gives correct (i.e. most recent version of address) responses.
• **Correctness**: For any PPT malicious server, MC either aborts or gives correct (i.e. most recent version of address) responses.
• **Correctness:** For any PPT malicious server, MC either aborts or gives correct (i.e. most recent version of address) responses.
Memory Checking

[Blum, Evans, Gemmel, Kannan, Naor '94]

- **Correctness**: For any PPT malicious server, MC either aborts or gives correct (i.e. most recent version of address) responses.

- **Completeness**: If the server behaves honestly, MC doesn’t abort.
Multiple Users Sharing Memory
Multiple Users Sharing Memory

- Examples
Multiple Users Sharing Memory

- Examples
  - Shared database across many clients
Multiple Users Sharing Memory

- Examples
  - Shared database across many clients
  - CPUs with shared memory
Multiple Users Sharing Memory

- Examples
  - Shared database across many clients
  - CPUs with shared memory
  - Distributed computing
Multiple Users Sharing Memory

- Examples
  - Shared database across many clients
  - CPUs with shared memory
  - Distributed computing
  - Integrity verification is very useful here too!
More Modes of Failure!

Consider two concurrent writes:

write(addr, data₁)

write(addr, data₂)
More Modes of Failure!

Consider two concurrent writes:

1. write(addr, data₁)
2. write(addr, data₂)

Let's denote the address as \( \emptyset \) and the data as variables \( data₁ \) and \( data₂ \).
More Modes of Failure!

Consider two concurrent writes:

1. Write at address `addr` with data `data_1`.
2. Write at address `addr` with data `data_2`.

The diagram shows a scenario where the two writes result in a conflict at address `addr`. The first writer sees `data_1`, and the second writer sees `data_2`.

This illustrates the potential for data corruption when multiple writers access the same location synchronously.
More Modes of Failure!

Consider two concurrent writes:

1. \text{write}(\text{addr}, \text{data}_1)
2. \text{write}(\text{addr}, \text{data}_2)
More Modes of Failure!

Consider two concurrent writes:

1. Write to addr, set to data₁
2. Write to addr, set to data₂

Then read(addr), see data₁

Then read(addr), see data₂
More Modes of Failure!

Consider two concurrent writes:

- User writes `data_1` to address `addr`.
- User writes `data_2` to address `addr`.

As a result, the system may end up with:

- The user reads `data_1` from address `addr`.
- The system stores `data_1` as the final result.

However, the system should ideally store both `data_1` and `data_2` as the final results, which is not the case here.
More Modes of Failure!

Consider two concurrent writes:

This “branching” might happen even when the server makes an “honest mistake”!!
More Modes of Failure!

Consider two concurrent writes:

This “branching” might happen even when the server makes an “honest mistake”!!
Branching Timelines

*From Marvel TV Series: Loki*
Branching Timelines

From Marvel TV Series: Loki
We need to maintain a single consistent version of the database (or *sacred timeline*) across all the clients!
Our Definition
Memory Checking for Parallel RAMs

Concurrent accesses

$m$ Clients

$M_1$

$M_2$

$\vdots$

$M_m$

Database of size $N$
Memory Checking for Parallel RAMs

We now define a notion of memory checking for parallel RAMs.

$m$ Clients  
Database of size $N$
We now define a notion of memory checking for parallel RAMs.

Set-up phase:

$m$ Clients

Database of size $N$
We now define a notion of memory checking for parallel RAMs.

Set-up phase:

• $\{M_i\}$ initialize their states together.
We now define a notion of memory checking for parallel RAMs.

Set-up phase:

- \( \{M_i\} \) initialize their states together.
- No *direct* communication after (except through the server).
We now define a notion of memory checking for parallel RAMs.

Set-up phase:

• \( \{M_i\} \) initialize their states together.

• No direct communication after (except through the server).

Note: It is possible that the clients have secure channels, but we want to make no assumptions.
Memory Checking for Parallel RAMs

Concurrent accesses

$M_1$

$M_2$

\ldots

$M_m$
Memory Checking for Parallel RAMs

- **Correctness:** All $\{M_i\}_i$ send back correct responses, or some $M_j$ aborts.
Memory Checking for Parallel RAMs

- **Correctness:** All \( \{M_i\}_i \) send back correct responses, or some \( M_j \) aborts.

- All reads are **correct**
Memory Checking for Parallel RAMs

• **Correctness:** All $\{M_i\}_i$ send back correct responses, or some $M_j$ aborts.

  • All reads are correct
  
  • Concurrent writes are tie-broken (arbitrarily chosen by server)
Memory Checking for Parallel RAMs

- **Correctness:** All $\{M_i\}_i$ send back correct responses, or some $M_j$ aborts.

  - All reads are correct
  - Concurrent writes are tie-broken (arbitrarily chosen by server)
Memory Checking for Parallel RAMs

- **Correctness:** All $\{M_i\}_i$ send back correct responses, or some $M_j$ aborts.
  - All reads are correct
  - Concurrent writes are tie-broken (arbitrarily chosen by server)
- **Completeness:** No $M_j$ aborts if server is not malicious.
Memory Checking for Parallel RAMs

Concurrent accesses

\[ M_1 \]

\[ M_2 \]

\[ \vdots \]

\[ M_m \]
Memory Checking for Parallel RAMs

Concurrent accesses

$M_1$

$M_2$

\vdots

$M_m$

Efficiency metrics
Memory Checking for Parallel RAMs

Efficiency metrics

- **Local Space:** Space per checker. This talk: $O(1)$ words/$O(\lambda)$ bits.
Memory Checking for Parallel RAMs

Efficiency metrics

• **Local Space:** Space per checker. This talk: $O(1)$ words/$O(\lambda)$ bits.

Concurrent accesses

Local Space
Memory Checking for Parallel RAMs

Efficiency metrics

- **Local Space**: Space per checker. This talk: $O(1)$ words/$O(\lambda)$ bits.
- **Server Space**: Server storage size. This talk: $O(N)$ words.
Memory Checking for Parallel RAMs

Efficiency metrics

- **Local Space**: Space per checker. This talk: $O(1)$ words/$O(\lambda)$ bits.

- **Server Space**: Server storage size. This talk: $O(N)$ words.
Memory Checking for Parallel RAMs

Efficiency metrics

- **Local Space:** Space per checker. This talk: \(O(1)\) words/\(O(\lambda)\) bits.

- **Server Space:** Server storage size. This talk: \(O(N)\) words.

- **Work blowup:** Ratio of server accesses per underlying PRAM access.
Memory Checking for Parallel RAMs

Efficiency metrics

- **Local Space**: Space per checker. This talk: $O(1)$ words/$O(\lambda)$ bits.
- **Server Space**: Server storage size. This talk: $O(N)$ words.
- **Work blowup**: Ratio of server accesses per underlying PRAM access.
Memory Checking for Parallel RAMs

Efficiency metrics

- **Local Space**: Space per checker. This talk: $O(1)$ words/$O(\lambda)$ bits.
- **Server Space**: Server storage size. This talk: $O(N)$ words.
- **Work blowup**: Ratio of server accesses per underlying PRAM access.
- **Depth blowup**: Number of parallel steps to support a single batch of requests.
Memory Checking for Parallel RAMs

**Efficiency metrics**

- **Local Space**: Space per checker. This talk: $O(1)$ words/$O(\lambda)$ bits.
- **Server Space**: Server storage size. This talk: $O(N)$ words.
- **Work blowup**: Ratio of server accesses per underlying PRAM access.
- **Depth blowup**: Number of parallel steps to support a single batch of requests.
Memory Checking for Parallel RAMs

Efficiency metrics

- **Local Space**: Space per checker. This talk: $O(1)$ words/$O(\lambda)$ bits.
- **Server Space**: Server storage size. This talk: $O(N)$ words.
- **Work blowup**: Ratio of server accesses per underlying PRAM access.
- **Depth blowup**: Number of parallel steps to support a single batch of requests.
Our Results
Our Results
Our Results

- For RAM setting, the best constructions have $O(\log N)$ work blow-up. Lower bound of $\Omega(\log N / \log \log N)$ known for special cases.

[Blum-Evans-Gemmel-Kannan-Naor '91]
Our Results

• For RAM setting, the best constructions have $O(\log N)$ work blow-up. Lower bound of $\Omega(\log N/\log \log N)$ known for special cases.
Our Results

- For RAM setting, the best constructions have $O(\log N)$ work blow-up. Lower bound of $\Omega(\log N/\log \log N)$ known for special cases.
Our Results

• For RAM setting, the best constructions have $O(\log N)$ work blow-up. Lower bound of $\Omega(\log N/\log \log N)$ known for special cases.

• Immediately gives: $m$-CPU PRAM memory checker with $O(\log N)$ work blow-up by serialising the algorithm. But $O(m \log N)$ depth blowup.
Our Results

- For RAM setting, the best constructions have $O(\log N)$ work blow-up. Lower bound of $\Omega(\log N/\log \log N)$ known for special cases.

- Immediately gives: $m$-CPU PRAM memory checker with $O(\log N)$ work blow-up by serialising the algorithm. But $O(m \log N)$ depth blowup.

- We show how to also obtain $O(\log N)$ depth.
Our Results

- For RAM setting, the best constructions have $O(\log N)$ work blow-up. Lower bound of $\Omega(\log N/\log \log N)$ known for special cases.

- Immediately gives: $m$-CPU PRAM memory checker with $O(\log N)$ work blow-up by serialising the algorithm. **But** $O(m \log N)$ depth blowup.

- We show how to also obtain $O(\log N)$ depth.

**Theorem 1.** Assuming OWFs, there exists a memory checking protocol for PRAM programs with $O(\log N)$ **worst-case work** and **depth** blowup.
Our Results

- For RAM setting, the best constructions have $O(\log N)$ work blow-up. Lower bound of $\Omega(\log N / \log \log N)$ known for special cases.

  - Immediately gives: $m$-CPU PRAM memory checker with $O(\log N)$ work blow-up by serialising the algorithm. **But** $O(m \log N)$ depth blowup.

- We show how to also obtain $O(\log N)$ **depth**.

**Theorem 1.** Assuming OWFs, there exists a memory checking protocol for PRAM programs with $O(\log N)$ worst-case work and depth blowup.
Application to Oblivious Parallel RAM
Application to Oblivious Parallel RAM

- Oblivious Parallel RAMs (OPRAMs) are access-pattern hiding PRAM compilers. [Boyle-Chung-Pass ’16]
Application to Oblivious Parallel RAM

- Oblivious Parallel RAMs (OPRAMs) are access-pattern hiding PRAM compilers. [Boyle-Chung-Pass ’16]

- A recent work constructed an honest-but-curious OPRAM constructions with $O(\log N)$ blowup in both work and depth. Optimal! [Asharov-Komargodski-Lin-Peserico-Shi ’22]
Application to Oblivious Parallel RAM

- Oblivious Parallel RAMs (OPRAMs) are access-pattern hiding PRAM compilers. [Boyle-Chung-Pass '16]

- A recent work constructed an honest-but-curious OPRAM constructions with $O(\log N)$ blowup in both work and depth. Optimal! [Asharov-Komargodski-Lin-Peserico-Shi '22]

- We obtain the first construction of maliciously secure OPRAM with polylogarithmic overhead.
Application to Oblivious Parallel RAM

- Oblivious Parallel RAMs (OPRAMs) are access-pattern hiding PRAM compilers.  
  [Boyle-Chung-Pass ’16]

- A recent work constructed an honest-but-curious OPRAM constructions with \( O(\log N) \) blowup in both work and depth. Optimal!  
  [Asharov-Komargodski-Lin-Peserico-Shi ’22]

- We obtain the first construction of maliciously secure OPRAM with polylogarithmic overhead.

**Theorem 2.** Assuming OWFs, there exists an maliciously secure OPRAM compiler with \( O(\log^2 N) \) work and depth blowup*. 


Our Results

**Theorem 3.** Assuming OWFs, there exists an *offline* memory checking protocol for PRAM programs with $O(1)$ amortised *work* and *depth* blowup.
Our Results

**Theorem 3.** Assuming OWFs, there exists an offline memory checking protocol for PRAM programs with $O(1)$ amortised work and depth blowup.

*Query phase:* Answers can be wrong! Repeat until clients say done.
Theorem 3. Assuming OWFs, there exists an offline memory checking protocol for PRAM programs with $O(1)$ amortised work and depth blowup.

**Query phase:** Answers can be wrong! Repeat until clients say done.
Our Results

**Theorem 3.** Assuming OWFs, there exists an *offline* memory checking protocol for PRAM programs with $O(1)$ amortised *work* and *depth* blowup.

*Query phase:* Answers can be wrong! Repeat until clients say done.

*Verification phase:* Reports if **all correct** or **some mistake.**
Our Results

**Theorem 3.** Assuming OWFs, there exists an offline memory checking protocol for PRAM programs with $O(1)$ amortised work and depth blowup.
Our Results

**Theorem 3.** Assuming OWFs, there exists an *offline* memory checking protocol for PRAM programs with $O(1)$ amortised *work* and *depth* blowup.

Checks if any mistake happened after a large batch of concurrent requests.
Our Results

**Theorem 3.** Assuming OWFs, there exists an *offline* memory checking protocol for PRAM programs with $O(1)$ amortised work and depth blowup.
Our Results

**Theorem 3.** Assuming OWFs, there exists an **offline** memory checking protocol for PRAM programs with $O(1)$ amortised *work* and *depth* blowup.

- **Idea:** Adapt the counting technique from a previous work [M.-Vafa ’23]
Theorem 3. Assuming OWFs, there exists an offline memory checking protocol for PRAM programs with $O(1)$ amortised work and depth blowup.

- **Idea:** Adapt the counting technique from a previous work [M.-Vafa ’23]
- Memory checkers maintain local counters $T_i$ of the number of updates.
Our Results

**Theorem 3.** Assuming OWFs, there exists an *offline* memory checking protocol for PRAM programs with $O(1)$ amortised *work* and *depth* blowup.

- **Idea:** Adapt the counting technique from a previous work [M.-Vafa ’23]
- Memory checkers maintain local counters $T_i$ of the number of updates.
- Every database entry is tagged with counters $\text{ctr}_i$
Our Results

**Theorem 3.** Assuming OWFs, there exists an *offline* memory checking protocol for PRAM programs with $O(1)$ amortised *work* and *depth* blowup.

- **Idea:** Adapt the counting technique from a previous work [M.-Vafa ’23]
- Memory checkers maintain local counters $T_i$ of the number of updates.
- Every database entry is tagged with counters $\text{ctr}_i$
- Verification phase: Check if $\sum_i \text{ctr}_i = \sum_j T_j$. 
New: Offline-Checking for PRAMs

ctr_i: #times D[i] updated
T_i: #times M_i wrote

write(4, data_1)
write(4, data_2)
write(4, data_3)
New: Offline-Checking for PRAMs

$c_{r_i}$: #times $D[i]$ updated
$T_i$: #times $M_i$ wrote
New: Offline-Checking for PRAMs

c\text{ctr}_i: \text{#times} D[i] \text{ updated}
T_i: \text{#times} M_i \text{ wrote}
New: Offline-Checking for PRAMs

\[ D[1] \]
\[ \text{ctr}_1 \]
\[ D[2] \]
\[ \text{ctr}_2 \]
\[ D[3] \]
\[ \text{ctr}_3 \]
\[ D[4] \]
\[ \text{ctr}_4 \]
\[ D[5] \]
\[ \text{ctr}_5 \]
\[ D[6] \]
\[ \text{ctr}_6 \]
\[ D[7] \]
\[ \text{ctr}_7 \]
\[ D[8] \]
\[ \text{ctr}_8 \]

\[ \text{ctr}_i : \# \text{times } D[i] \text{ updated} \]
\[ T_i : \# \text{times } M_i \text{ wrote} \]

\[ M_1 \]
\[ T_1 \]
write(4, data1)

\[ M_2 \]
\[ T_2 \]
write(4, data2)

\[ M_3 \]
\[ T_3 \]
write(4, data3)

\[ M_1 \text{ won :)} \]
\[ M_1 \text{ won :)} \]
New: Offline-Checking for PRAMs

$\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, $\mathcal{D}_4$, $\mathcal{D}_5$, $\mathcal{D}_6$, $\mathcal{D}_7$, $\mathcal{D}_8$ cause incremental counting, where $\mathcal{D}_i$ causally leads $\mathcal{D}_{i+1}$.

$c_{tr_i}$: #times $\mathcal{D}_i$ updated

$T_i$: #times $M_i$ wrote

Incremented!

$M_1$, $M_2$, $M_3$ cause relation checking, where $M_i$ causally leads $M_{i+1}$.

Incremented!

$M_1$ won :(

$M_1$ won :(

$T_1$, $T_2$, $T_3$ cause data checking, where $T_i$ causally leads $T_{i+1}$.

$\text{write}(4, \text{data}_1)$

$\text{write}(4, \text{data}_2)$

$\text{write}(4, \text{data}_3)$

Pick me!

Pick me!

Write back $\text{data}_1$ and increment $c_{tr_4}$

Pick me!
New: Offline-Checking for PRAMs

Note that this maintains:

\[ \sum_i \text{ctr}_i = \sum_j T_j \]

since both sides are incremented.
Issue 1: Spoofing Attack

\[
\begin{align*}
\text{Pick me!} & \quad \text{Pick me!} & \quad \text{Pick me!} \\
\text{write}(4, \text{data}_1) & \quad \text{write}(4, \text{data}_2) & \quad \text{write}(4, \text{data}_3)
\end{align*}
\]
Issue 1: Spoofing Attack

Pick me!  

$M_2$ won :(

Pick me!  

$M_3$ won :(

$M_1$ won :(

$T_1$ write(4, data_1)

$T_2$ write(4, data_2)

$T_3$ write(4, data_3)
Issue 1: Spoofing Attack

Idea: Use authentication, and a careful “agreement” protocol to ensure someone wins.
Issue 2: Branching attack

Pick me!
Issue 2: Branching attack

[Diagram showing branching attack with nodes labeled $D[1]$ to $D[8]$ and $M_1$, $M_2$, $M_3$.]
Issue 2: Branching attack

$M_1$ increments $T_1$ locally.

$M_2$ increments $T_2$ locally.

You won!

$M_1$ won :(

write(4, data_1)

write(4, data_2)

write(4, data_3)
Issue 2: Branching attack

$D_1$ increments $T_1$ locally.

$M_1$ increments $T_1$ locally.

$M_1$ won!

Can only incremented by 1!!

$D_2$ increments $T_2$ locally.

$M_2$ increments $T_2$ locally.

$M_2$ won :( 

$D_3$ increments $T_1$ locally.

write(4,data_1)

write(4,data_2)

write(4,data_3)

You won!
Issue 2: Branching attack

Note that this maintains:

\[ \sum_i \text{ctr}_i \leq \sum_j \text{T}_j \]

Where equality no longer holds iff exactly one CPU won.
Open Problems
Open Problems

• Can we obtain a statistically secure offline memory checker for CRCW programs with $O(1)$ amortised overhead?
Open Problems

• Can we obtain a **statistically secure** offline memory checker for CRCW programs with $O(1)$ amortised overhead?

• Known for single RAM setting, and we show for PRAM setting **without** concurrent read/writes.
Open Problems

• Can we obtain a **statistically secure** offline memory checker for CRCW programs with $O(1)$ amortised overhead?

  • Known for single RAM setting, and we show for PRAM setting **without** concurrent read/writes.

• Can we use these memory checkers to obtain an **optimal maliciously secure** OPRAM with $O(\log N)$ work and depth blow-up?
Open Problems

• Can we obtain a **statistically secure** offline memory checker for CRCW programs with $O(1)$ amortised overhead?

  • Known for single RAM setting, and we show for PRAM setting **without** concurrent read/writes.

• Can we use these memory checkers to obtain an **optimal maliciously secure** OPRAM with $O(\log N)$ work and depth blow-up?

  • [M-Vafa ’23] shows an $O(\log N)$ **maliciously secure** ORAM construction by interleaving **offline** and **online** memory checking.
Open Problems

• Can we obtain a **statistically secure offline** memory checker for CRCW programs with $O(1)$ amortised overhead?

  • Known for single RAM setting, and we show for PRAM setting **without** concurrent read/writes.

• Can we use these memory checkers to obtain an **optimal maliciously secure OPRAM** with $O(\log N)$ work and depth blow-up?

  • [M-Vafa ’23] shows an $O(\log N)$ **maliciously secure** ORAM construction by interleaving **offline** and **online** memory checking.

  • Can we do the same?
Bonus Slides
CRCW Parallel RAM Model

\[\text{m Clients} \quad \text{Database of size } N\]
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory

$m$ Clients

Database of size $N$
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory

$m$ Clients  Database of size $N$
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory

![Diagram showing concurrent accesses to a database](image-url)
CRCW Parallel RAM Model

- **PRAM:** Multiple CPUs accessing shared memory
- **CRCW Model:** Concurrent **Read** Concurrent **Write**

$m$ Clients

Database of size $N$
CRCW Parallel RAM Model

- **PRAM:** Multiple CPUs accessing shared memory
- **CRCW Model:** Concurrent Read Concurrent Write
  - **Concurrent reads** to any location are allowed.
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory
- **CRCW Model**: Concurrent **Read** Concurrent **Write**
- **Concurrent reads** to any location are allowed.

$m$ Clients \hspace{1cm} Database of size $N$
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory
- **CRCW Model**: Concurrent \textbf{R}ead Concurrent \textbf{W}rite
- **Concurrent reads** to any location are allowed.

$m$ Clients \hspace{3cm} Database of size $N$
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory

- **CRCW Model**: Concurrent Read Concurrent Write
  - Concurrent reads to any location are allowed.
  - Concurrent writes to any location are **arbitrarily** tie-broken.
CRCW Parallel RAM Model

- **PRAM:** Multiple CPUs accessing shared memory

- **CRCW Model:** Concurrent Read Concurrent Write
  - Concurrent reads to any location are allowed.
  - Concurrent writes to any location are *arbitrarily* tie-broken.

$m$ Clients  Database of size $N$
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory

- **CRCW Model**: Concurrent Read Concurrent Write
  - Concurrent reads to any location are allowed.
  - Concurrent writes to any location are *arbitrarily* tie-broken.

$m$ Clients       Database of size $N$
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory

- **CRCW Model**: Concurrent **Read** Concurrent **Write**
  - **Concurrent reads** to any location are allowed.
  - **Concurrent writes** to any location are **arbitrarily** tie-broken.
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory
- **CRCW Model**: Concurrent **Read** Concurrent **Write**
  - Concurrent reads to any location are allowed.
  - Concurrent writes to any location are arbitrarily tie-broken.
- Note: Not every client has to perform an operation.
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory

- **CRCW Model**: Concurrent **R**ead Concurrent **W**rite
  - **Concurrent reads** to any location are allowed.
  - **Concurrent writes** to any location are *arbitrarily* tie-broken.
  - Note: Not every client has to perform an operation.

- Two efficiency metrics for PRAM algorithms:

\[
\text{Concurrent accesses } \leq \frac{m - 1}{N} \leq \text{Concurrent writes}
\]
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory
- **CRCW Model**: Concurrent Read Concurrent Write
  - **Concurrent reads** to any location are allowed.
  - **Concurrent writes** to any location are arbitrarily tie-broken.
- Note: Not every client has to perform an operation.

- **Two efficiency metrics for PRAM algorithms**:  
  - **Work**: Number of read/write operations

$m$ Clients

Database of size $N$
CRCW Parallel RAM Model

- **PRAM**: Multiple CPUs accessing shared memory
- **CRCW Model**: Concurrent Read Concurrent Write
  - Concurrent reads to any location are allowed.
  - Concurrent writes to any location are arbitrarily tie-broken.
  - Note: Not every client has to perform an operation.

- **Two efficiency metrics for PRAM algorithms**:
  - **Work**: Number of read/write operations
  - **Depth**: Number of parallel steps

\[ \text{Clients} \quad \text{Database of size } N \]
Idea: Compose ORAM and MC
Idea: Compose ORAM and MC

$m$ Clients

\[ \mathcal{O}_1 \quad M_1 \]

\[ \mathcal{O}_2 \quad M_2 \]

\[ \vdots \]

\[ \mathcal{O}_m \quad M_m \]

OPRAM

Memory Checker

Database of size $N$
Idea: Compose ORAM and MC

$m$ Clients

Maliciously Secure OPRAM

Database of size $N$

OPRAM

Memory Checker
Idea: Compose ORAM and MC

$m$ Clients

Maliciously Secure OPRAM

\[ O_1 \rightarrow M_1 \]

\[ O_2 \rightarrow M_2 \]

\[ \vdots \]

\[ O_m \rightarrow M_m \]

OPRAM  Memory Checker

Database of size $N$
Idea: Compose ORAM and MC
Idea: Compose ORAM and MC

$m$ Clients

Maliciously Secure OPRAM

Database of size $N$

$O_1$ $M_1$

$O_2$ $M_2$

$\cdots$

$O_m$ $M_m$

OPRAM Memory Checker
Idea: Compose ORAM and MC

$m$ Clients

Maliciously Secure OPRAM

OPRAM

Memory Checker

Database of size $N$
Idea: Compose ORAM and MC

$m$ Clients

Maliciously Secure OPRAM

Database of size $N$

OPRAM

Memory Checker
Idea: Compose ORAM and MC

$m$ Clients

Maliciously Secure OPRAM

OPRAM

Memory Checker

Database of size $N$
Idea: Compose ORAM and MC

Maliciously Secure OPRAM

- MC ensures all responses to OPRAM are correct.
Idea: Compose ORAM and MC

- MC ensures all responses to OPRAM are correct.
- Can invoke semi-honest security of OPRAM!
Idea: Compose ORAM and MC

$\mathcal{O}_1$ $\mathcal{O}_2$ $\ldots$ $\mathcal{O}_m$

$M_1$ $M_2$ $M_m$

$m$ Clients

Maliciously Secure OPRAM

Database of size $N$

- MC ensures all responses to OPRAM are correct.
- Can invoke semi-honest security of OPRAM!

Blowup (work and depth):
Idea: Compose ORAM and MC

$m$ Clients

Maliciously Secure OPRAM

$\mathcal{O}_1$ $\mathcal{M}_1$

$\mathcal{O}_2$ $\mathcal{M}_2$

$\ldots$

$\mathcal{O}_m$ $\mathcal{M}_m$

OPRAM

Memory Checker

Database of size $N$

- **MC** ensures all responses to OPRAM are correct.
- Can invoke semi-honest security of OPRAM!

Blowup (work and depth):

$$\log N \times \log N = \log^2 N$$
Offline-Checking for RAMs

[Blum, Evans, Gemmel, Kannan, Naor ’94], this construction is based on [M, Vafa ’23]
Offline-Checking for RAMs

[Blum, Evans, Gemmel, Kannan, Naor '94], this construction is based on [M, Vafa '23]
Offline-Checking for RAMs

[Blum, Evans, Gemmel, Kannan, Naor ’94], this construction is based on [M, Vafa ’23]

- For entry $D[i]$, let $\text{ctr}_i =$ number of times location $i$ was read or written to. Initialise to 0.
Offline-Checking for RAMs

[Blum, Evans, Gemmel, Kannan, Naor ’94], this construction is based on [M, Vafa ‘23]

- For entry $D[i]$, let $\text{ctr}_i =$ number of times location $i$ was \textit{read or written} to. Initialise to 0.
- Memory checker local stores a counter $T$ initialised to 0.
Offline-Checking for RAMs

[Blum, Evans, Gemmel, Kannan, Naor '94], this construction is based on [M, Vafa '23]

- For entry $D[i]$, let $\text{ctr}_i = \text{number of times location } i \text{ was read or written}$ to. Initialise to 0.

- Memory checker local stores a counter $T$ initialised to 0.

- For every access to $D[i]$, increment $T$ locally, and increment $\text{ctr}_i$. 
• For entry $D[i]$, let $\text{ctr}_i = \text{number of times location } i \text{ was read or written}$ to. Initialise to 0.

• Memory checker local stores a counter $T$ initialised to 0.

• For every access to $D[i]$, increment $T$ locally, and increment $\text{ctr}_i$.

  At the end, the memory checker iterates over the array and verifies $\sum_i \text{ctr}_i = T$. 
Offline-Checking for RAMs

[Blum, Evans, Gemmel, Kannan, Naor ’94], this construction is based on [M, Vafa ’23]

- For entry $D[i]$, let $\text{ctr}_i =$ number of times location $i$ was read or written.
- Memory checker local stores a counter $T$ initialised to 0.
- For every access to $D[i]$, increment $T$ locally, and increment $\text{ctr}_i$.
- At the end, the memory checker iterates over the array and verifies $\sum_i \text{ctr}_i = T$.

Idea: Let $t_i$ be the number of times $D[i]$ was actually accessed.

$$\sum_i \text{ctr}_i \leq \sum_i t_i = T$$

where equality holds iff there was no replay attack.
New: Offline-Checking for PRAMs
New: Offline-Checking for PRAMs
• **As before:** Initialise all entries with $\text{ctr}_i = 0$ and authenticate all entries.

• Each $M_i$ keeps a local count $T_i$. 
New: Offline-Checking for PRAMs
New: Offline-Checking for PRAMs
New: Offline-Checking for PRAMs

Idea: Want to maintain

\[ \sum_i \text{ctr}_i \leq \sum_j T_j \]

where equality holds iff server acts honestly.
New: Offline-Checking for PRAMs
**New: Offline-Checking for PRAMs**

Diagram showing a sequence of data points with corresponding control variables and memory modules. The diagram includes arrows indicating the flow of data and control signals. Each data point is labeled with a letter and a subscript, e.g., $D[1]$, $D[2]$, etc. Control variables are denoted by $ctr_1$, $ctr_2$, etc. Memory modules are represented by $M_1$, $M_2$, and $M_3$, with corresponding timestamps $T_1$, $T_2$, and $T_3$.

Arrows are labeled with the message "Pick me!" indicating the path of the signal flow. The diagram also includes operations such as `write(4, data_1)`, `write(4, data_2)`, and `write(4, data_3)`.
New: Offline-Checking for PRAMs

\[
\begin{align*}
&\quad \text{ctr}_1 \quad \text{ctr}_2 \quad \text{ctr}_3 \quad \text{ctr}_4 \quad \text{ctr}_5 \quad \text{ctr}_6 \quad \text{ctr}_7 \quad \text{ctr}_8
\end{align*}
\]

- Pick me!
- You won!
- \( M_1 \) won :
- \( M_2 \) won :
- \( M_3 \) won :

\[
\begin{align*}
&T_1 \quad T_2 \quad T_3 \\
&\quad \text{write}(4, \text{data}_1) \quad \text{write}(4, \text{data}_2) \quad \text{write}(4, \text{data}_3)
\end{align*}
\]
New: Offline-Checking for PRAMs

![Diagram of PRAMs and their counters]
New: Offline-Checking for PRAMs

- Incremented!
- Pick me!
- Pick me!
- Pick me!

- Write back data and increment \( \text{ctr}_4 \)
- \( M_1 \) won :(
- \( M_1 \) won :(

- \( T_1 \) write(4,\( \text{data}_1 \))
- \( T_2 \) write(4,\( \text{data}_2 \))
- \( T_3 \) write(4,\( \text{data}_3 \))
New: Offline-Checking for PRAMs

Note that this maintains:

\[ \sum_i \text{ctr}_i \leq \sum_j T_j \]

since both sides are incremented.