Advanced OpenMP

Charles Bouillaguet
charles.bouillaguet@lip6.fr

October 18, 2022

atomic is not a Panacea

Example: sum of an array

```
int sum = 0;
for (int i = 0; i < n; i++)
    sum += A[i];
```

$$T = 5.95\text{s} \qquad (n = 10^{10})$$

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < n; i++)
    #pragma omp atomic
    sum += A[i];
```

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++)

    sum += A[i];
```

`atomic` is not a Panacea

Example: sum of an array

```c
int sum = 0;
for (int i = 0; i < n; i++)
    sum += A[i];
```

$T = 5.95\text{s} \qquad (n = 10^{10})$

```c
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < n; i++)
    #pragma omp atomic
    sum += A[i];
```

$T \geq 200\text{s}$!!!

```c
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++)

    sum += A[i];
```

$T = 0.46\text{s}\ (\times 12.9)$

Histogram (E.g. numpy.histogram)

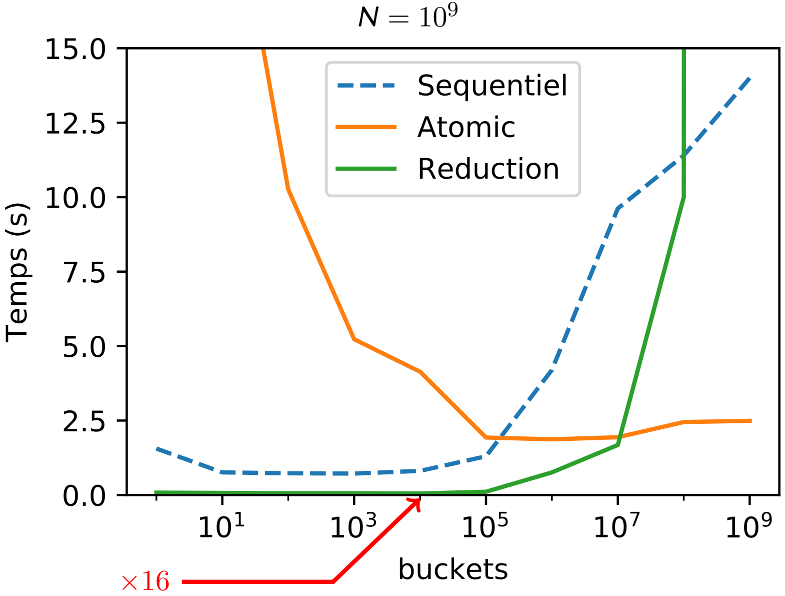
Plan A

```
void histogram(u32 A[], u64 n, u64 buckets, u64 H[])
{
    #pragma omp parallel for
    for (u64 i = 0; i < n; i++) {
        u64 x = (A[i] * buckets) >> 32;
        #pragma omp atomic
        H[x]++;
    }
}
```

Plan B

```
void histogram(u32 A[], u64 n, u64 buckets, u64 H[])
{
    #pragma omp parallel for reduction(+:H[0:buckets])
    for (u64 i = 0; i < n; i++) {
        u64 x = (A[i] * buckets) >> 32;
        H[x]++;
    }
}
```

Histogram



$N = 10^9$
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Temps (s)
Sequentiel
Atomic
Reduction
10^1
10^3
10^5
10^7
10^9
buckets
$\times 16$

The Sad Truth

- Barrier $\rightarrow$ waiting
- Critical $\rightarrow$ serialization
- Atomic $\rightarrow$ slower than normal memory accesses

Synchronization $\rightarrow$ limits scalability

$\Longrightarrow$ important role of data locality

You **must** follow the rule

You **must** follow the rule

Or else…

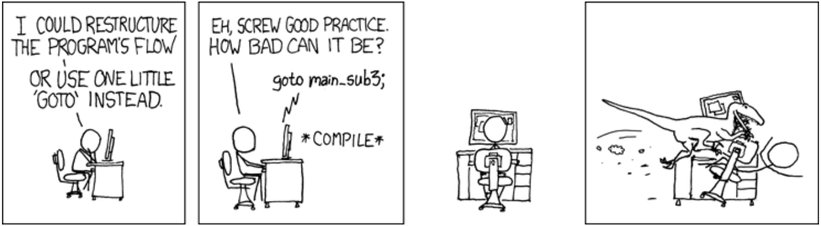
You **must** follow the rule

Or else…

You will be living

In a **world of PAIN**

Don't follow the rules?



https://xkcd.com/292/

Everyone inevitably goes through this

- Okay, but then if I avoid `i++`, it's okay, right?
- I read in my CPU doc that aligned reads/writes are atomic; if we stick to that, we'll be fine, right?

Everyone inevitably goes through this

- Okay, but then if I avoid `i++`, it's okay, right?
- I read in my CPU doc that aligned reads/writes are atomic; if we stick to that, we'll be fine, right?

Teenage crisis

Let's drop the golden rule

Everyone inevitably goes through this

- Okay, but then if I avoid `i++`, it's okay, right?
- I read in my CPU doc that aligned reads/writes are atomic; if we stick to that, we'll be fine, right?

Teenage crisis

Let's drop the golden rule

Guru switch ⟶

Safety

World of PAIN

Peterson Lock (1981)

```
bool flag[2];
int turn;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = 1;                            // I'm interested
    turn = 1 - i;                           // you go first
    while (turn != i && flag[1-i]) {};      // wait 'til he's not interested
}                                           //     or its my turn

void unlock()
{
    int i = omp_get_thread_num();
    flag[i] = 0;                            // I'm not interested
}
```

CLAIMS

- Mutual exclusion
- *Deadlock*-free
- *Starvation*-free

Peterson Lock

LIVE DEMO

Guru problem #1: **Compiler treachery**

- We had:

```
    turn = 1 - i;                             // you go first
    while (turn != i && flag[1-i]) {}; // wait
```

- The optimizer "knows" that `turn != i`
- This then becomes:

```
void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;                        // I'm interested
    while (flag[1-i]) {};                  // wait
}
```

- Both threads call `lock()` at the same time...
- ... `flag[0] == flag[1] == true` ...
- Deadlock.

Guru problem #2: **???**

- Mutual exclusion failure
- Both threads present in the critical section simultaneously
- How is this possible?

Order Relations Between Concurrent Memory Accesses

- Memory Accesses

$$W_i(x)a : T_i \text{ writes value } a \text{ in variable } x$$
$$R_i(x)b : T_i \text{ reads variable } x \text{ and gets value } b$$

- *Program Order*:

$$x \xrightarrow{po} y : \text{code states that } x \text{ is done first and } y \text{ after}$$

- *Extended Communication Order*:

$$\begin{array}{rcll} W(x)a & \xrightarrow{\texttt{rf}} & R(x)a & : \text{the read gets the written value} \\ W(x)a & \xrightarrow{\texttt{mo}} & W(x)b & : a \text{ is written before } b \\ R(x)a & \xrightarrow{\texttt{rb}} & W(x)b & : \text{The read takes place before } b \text{ is written in } x \\ & & & \quad \text{by definition, } \xrightarrow{\texttt{rb}} = (\xrightarrow{\texttt{rf}})^{-1}; \xrightarrow{\texttt{mo}} \end{array}$$

Order Relations Between Concurrent Memory Accesses

- Memory Accesses

$$W_i(x)a : T_i \text{ writes value } a \text{ in variable } x$$
$$R_i(x)b : T_i \text{ reads variable } x \text{ and gets value } b$$

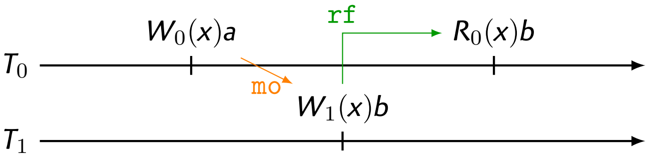
- *Program Order*:

$$x \xrightarrow{po} y : \text{code states that } x \text{ is done first and } y \text{ after}$$

- *Extended Communication Order*:

$$\begin{array}{rcll} W(x)a & \xrightarrow{\texttt{rf}} & R(x)a & : \text{the read gets the written value} \\ W(x)a & \xrightarrow{\texttt{mo}} & W(x)b & : a \text{ is written before } b \\ R(x)a & \xrightarrow{\texttt{rb}} & W(x)b & : \text{The read takes place before } b \text{ is written in } x \end{array}$$

by definition, $\xrightarrow{\texttt{rb}} = (\xrightarrow{\texttt{rf}})^{-1}; \xrightarrow{\texttt{mo}}$



Order Relations Between Concurrent Memory Accesses

$W(x)a \xrightarrow{\texttt{rf}} R(x)a$: the read gets the written value

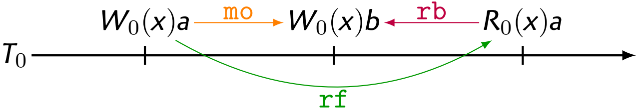
$W(x)a \xrightarrow{\texttt{mo}} W(x)b$: a is written before b

$R(x)a \xrightarrow{\texttt{rb}} W(x)b$: The read takes place before b is written in x

by definition, $\xrightarrow{\texttt{rb}} = (\xrightarrow{\texttt{rf}})^{-1} ; \xrightarrow{\texttt{mo}}$

Intuition

- In principle, $\xrightarrow{\texttt{rf}}$, $\xrightarrow{\texttt{mo}}$, $\xrightarrow{\texttt{rb}}$ cannot "contradict" $\xrightarrow{po}$
- Cannot read "outdated" values ($\xrightarrow{\texttt{rb}}$ backwards)



Order Relations Between Concurrent Memory Accesses

$W(x)a \xrightarrow{\texttt{rf}} R(x)a$: the read gets the written value

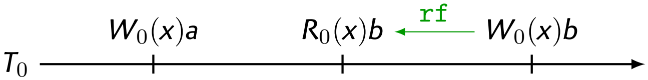
$W(x)a \xrightarrow{\texttt{mo}} W(x)b$: a is written before b

$R(x)a \xrightarrow{\texttt{rb}} W(x)b$: The read takes place before b is written in x

by definition, $\xrightarrow{\texttt{rb}} = (\xrightarrow{\texttt{rf}})^{-1}; \xrightarrow{\texttt{mo}}$

Intuition

- In principle, $\xrightarrow{\texttt{rf}}$, $\xrightarrow{\texttt{mo}}$, $\xrightarrow{\texttt{rb}}$ cannot “contradict” $\xrightarrow{po}$
- Cannot read “outdated” values ($\xrightarrow{\texttt{rb}}$ backwards)
- Cannot read “from the future” ($\xrightarrow{\texttt{rf}}$ backwards)

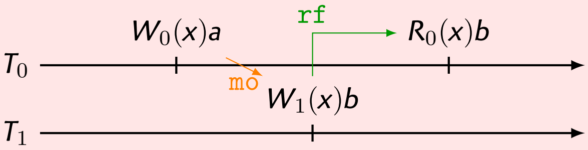


Order Relations Between Concurrent Memory Accesses

$$
\begin{array}{rcll}
W(x)a & \xrightarrow{\texttt{rf}} & R(x)a & : \text{the read gets the written value} \\
W(x)a & \xrightarrow{\texttt{mo}} & W(x)b & : a \text{ is written before } b \\
R(x)a & \xrightarrow{\texttt{rb}} & W(x)b & : \text{The read takes place before } b \text{ is written in } x \\
 & & & \quad \text{by definition, } \xrightarrow{\texttt{rb}} = (\xrightarrow{\texttt{rf}})^{-1}; \xrightarrow{\texttt{mo}}
\end{array}
$$

Sequentially : $\xrightarrow{\texttt{rf}} \subseteq \xrightarrow{po}$, $\xrightarrow{\texttt{mo}} \subseteq \xrightarrow{po}$ and $\xrightarrow{\texttt{rb}} \subseteq \xrightarrow{po}$

This is not true in parallel



Sequential Consistency

Intuition

Everything happens as if memory accesses were executed *sequentially* (in some order that we do not necessarily know).

Sequential Consistency

Intuition

Everything happens as if memory accesses were executed *sequentially* (in some order that we do not necessarily know).

Definition (Sequential Consistency)

A parallel system us sequentially consistent if, for each of the possible executions of the threads to which it can lead, we can build a history H:

- Totally ordered sequence
- Contains each memory access (only once)
- Compatible with the code of each thread (respects $\xrightarrow{po}$)
- Reading x gives the last value written in x

Theorem

Sequential Consistency $\iff$ *no cycles with* $\xrightarrow{po} \cup \xrightarrow{rf} \cup \xrightarrow{mo} \cup \xrightarrow{rb}$.

Peterson Lock (1981)

```
bool flag[2];
int turn;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = 1;                           // I'm interested
    turn = 1 - i;                          // you go first
    while (turn != i && flag[1-i]) {};     // wait 'til he's not interested
}                                          //     or its my turn

void unlock()
{
    int i = omp_get_thread_num();
    flag[i] = 0;                           // I'm not interested
}
```

CLAIMS

- Mutual exclusion
- *Deadlock*-free
- *Starvation*-free

Correctness of the *Peterson Lock*

Sequential Consistency

Everything happens as if memory accesses were executed *sequentially* (in some order that we do not necessarily know).

Theorem

If the memory is sequentially consistent, then the Peterson lock guarantees mutual exclusion.

Proof: ...

- Suppose not: both T_0 and T_1 call `lock()`, enter the critical section
- Initial state: `flag[0] = false; flag[1] = false;`
- Code requires that:

$$T_0 : W_0(\texttt{flag[0]})\,\texttt{true} \xrightarrow{po} W_0(\texttt{turn})1 \xrightarrow{po} R_0(\texttt{turn})? \xrightarrow{po} R_0(\texttt{flag[1]})? \xrightarrow{po} CS_0$$

$$T_1 : W_1(\texttt{flag[1]})\,\texttt{true} \xrightarrow{po} W_1(\texttt{turn})0 \xrightarrow{po} R_1(\texttt{turn})? \xrightarrow{po} R_1(\texttt{flag[0]})? \xrightarrow{po} CS_1$$

- Suppose that T_0 writes `turn` last:

$$W_1(\texttt{turn})0 \xrightarrow{\texttt{mo}} W_0(\texttt{turn})1.$$

- T_0 exits the loop, and `turn == 1`, then necessarily:

$$W_0(\texttt{turn})1 \xrightarrow{po} R_0(\texttt{turn})1 \xrightarrow{po} R_0(\texttt{flag[1]})\,\texttt{false}.$$

- Put all this together:

$$\begin{tikzcd}
W_{init}(\texttt{flag[1]})\,\texttt{false} \arrow[d, "\texttt{mo}"] \arrow[rrrd, "\texttt{rf}"] & & & \\
W_1(\texttt{flag[1]})\,\texttt{true} \arrow[r, "\texttt{po}"] & W_1(\texttt{turn})0 \arrow[r, "\texttt{mo}"] & W_0(\texttt{turn})1 \arrow[r, "\texttt{po}"] & R_0(\texttt{flag[1]})\,\texttt{false} \arrow[lll, bend left, "\texttt{rb}"]
\end{tikzcd}$$

- Cycle $\Rightarrow$ non-SC $\Rightarrow$ Contradiction!

Guru problem #2: **lack of *sequential consistency***

- Proof: sequential consistency $\Rightarrow$ mutual exclusion
- Observation : ~~mutual exclusion~~
- *Ergo*:

  - My laptop is not sequentially consistent...

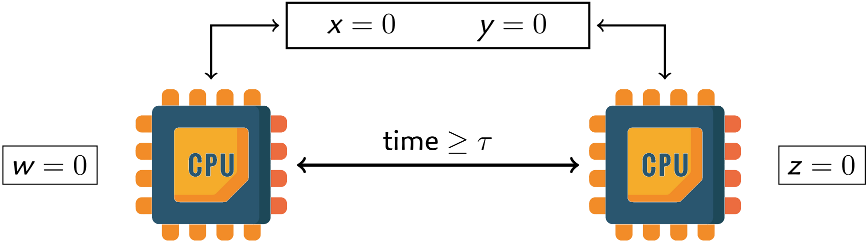
Guru problem #2: **lack of *sequential consistency***

- Proof: sequential consistency $\Rightarrow$ mutual exclusion
- Observation : ~~mutual exclusion~~
- *Ergo*:

  - My laptop is not sequentially consistent...
  - Yours is not either!

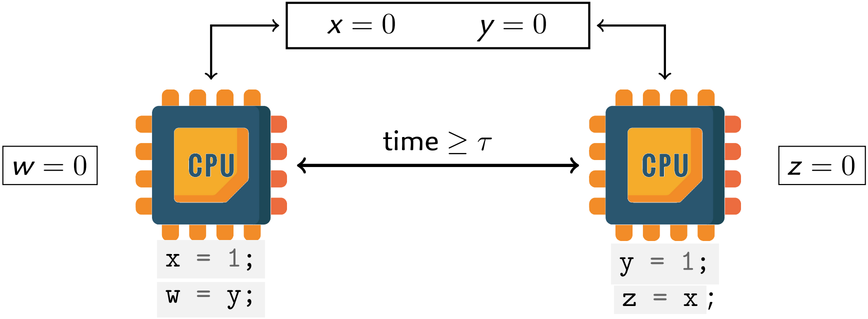
CPUs are not sequentially consistent!

Sequential Consistency is costly



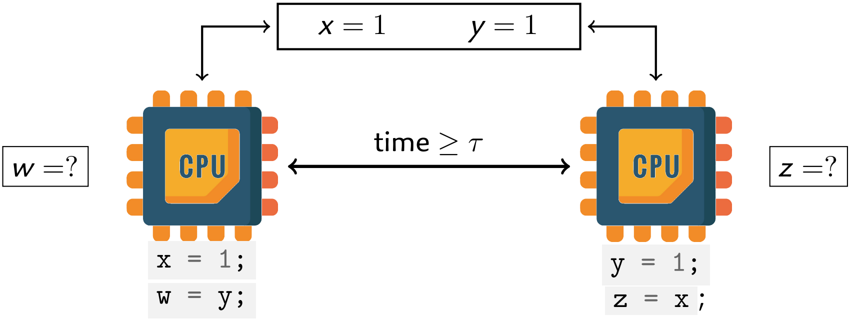
CPUs are not sequentially consistent!

Sequential Consistency is costly



CPUs are not sequentially consistent!

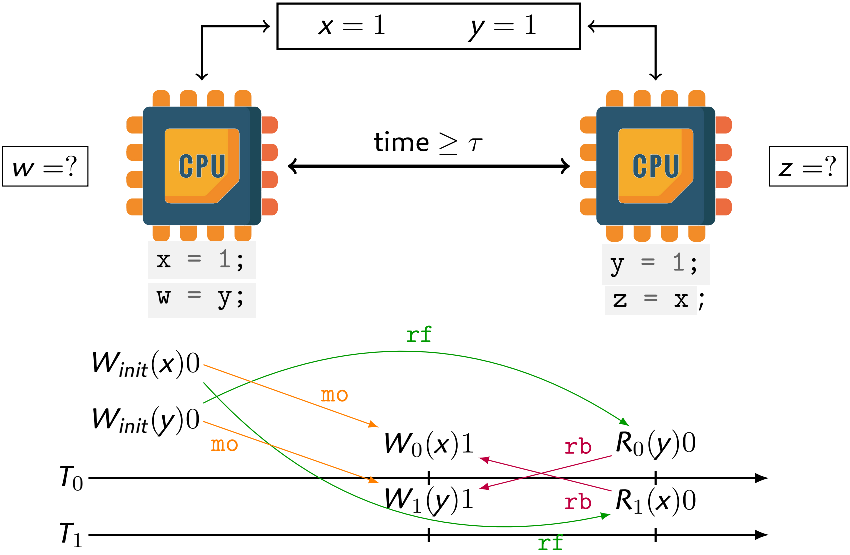
Sequential Consistency is costly



Sequential Consistency $\Rightarrow (w, z) \neq (0, 0)$

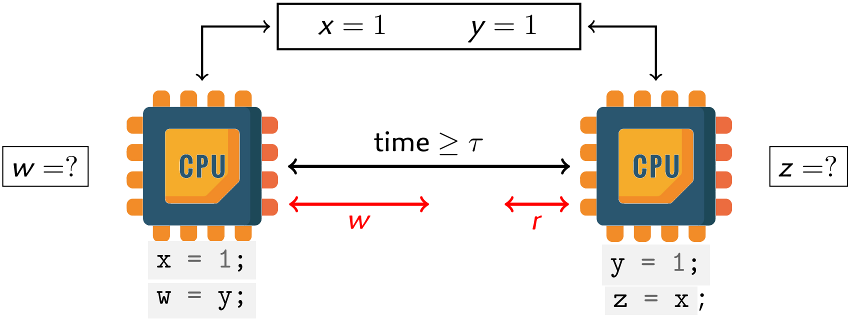
CPUs are not sequentially consistent!

Sequential Consistency is costly



CPUs are not sequentially consistent!

Sequential Consistency is costly



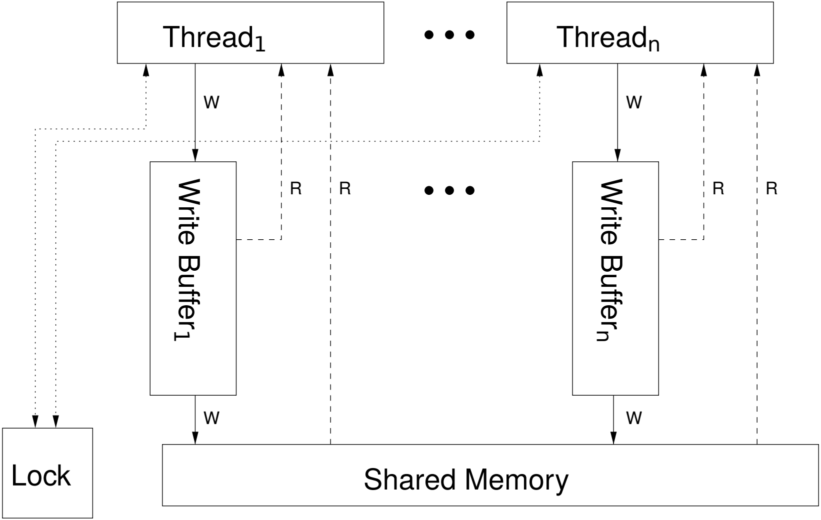
r : min. time to do a read

w : min. time to do a write

Sequential Consistency $\Longrightarrow r + w \geq \tau$

One must read a write from the other...

Store Buffering



(image : A Tutorial Introduction to the ARM and POWER Relaxed Memory Models)

Architectures with *Total Store Ordering*

x86 and SPARC

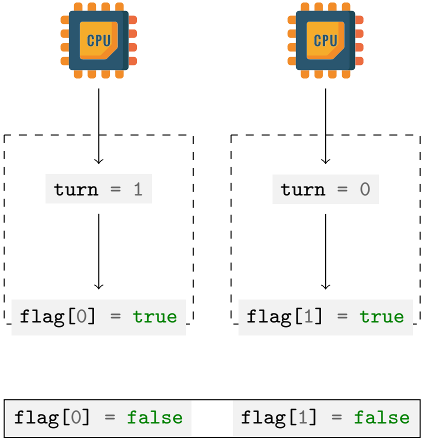
Each hardware thread has a **Store Buffer**

- My writes are enqueued in my *Store Buffer*
- FIFO queue
- I *see* the effects of my writes immediately
- My *Store Buffer* will be “flushed” to the memory... later
- At that time:
  - *All the other threads* “see” my writes
  - They see them in the same order (the order I did them)

Peterson Lock Failure in the presence of *Store Buffering*

Peterson Lock

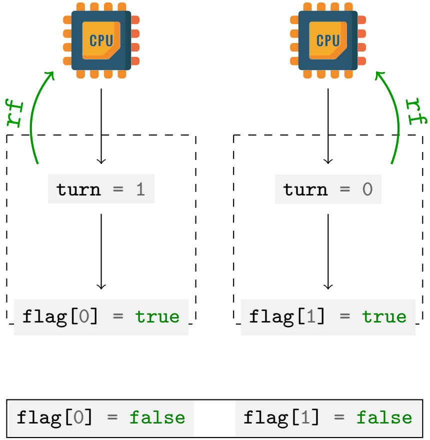
```
bool flag[2];
int victim;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;
    turn = 1 - i;
    while (turn != i && flag[1-i])
        {};
}
```



Peterson Lock Failure in the presence of *Store Buffering*

Peterson Lock

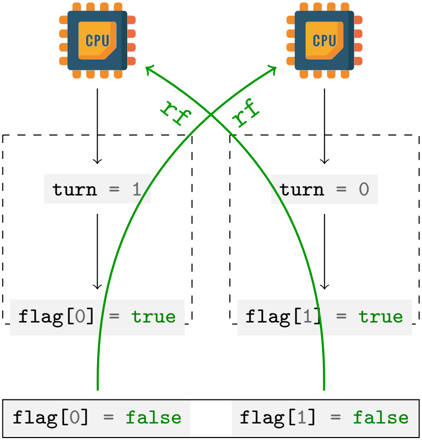
```
bool flag[2];
int victim;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;
    turn = 1 - i;
    while (turn != i && flag[1-i])
        {};
}
```



Peterson Lock Failure in the presence of *Store Buffering*

Peterson Lock

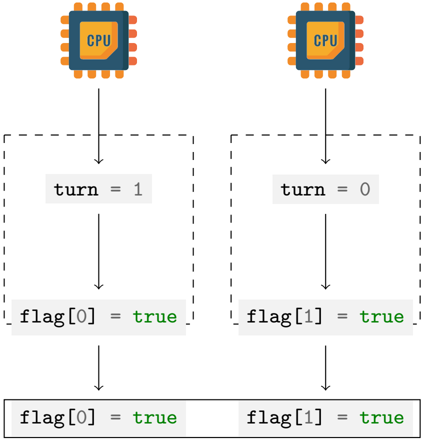
```
bool flag[2];
int victim;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;
    turn = 1 - i;
    while (turn != i && flag[1-i])
        {};
}
```



Peterson Lock Failure in the presence of *Store Buffering*

Peterson Lock

```
bool flag[2];
int victim;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;
    turn = 1 - i;
    while (turn != i && flag[1-i])
        {};
}
```



Message Passing

Producer

```
void produce(void *payload)
{
    msg = payload;
    flag = 1;
}
```

Consumer

```
void * receive()
{
    while (!flag) {}; // wait
    return msg;
}
```

Message Passing

Producer

```
void produce(void *payload)
{
    msg = payload;
    flag = 1;
}
```

Consumer

```
void * receive()
{
    while (!flag) {}; // wait
    return msg;
}
```

- Works with *Total Store Ordering*
- x86, SPARC, ...

Message Passing

Producer

```
void produce(void *payload)
{
    msg = payload;
    flag = 1;
}
```

Consumer

```
void * receive()
{
    while (!flag) {}; // wait
    return msg;
}
```

- Fails on ARM, POWER, ...
- Threads do not necessarily *see* the writes in the same order!

Message Passing

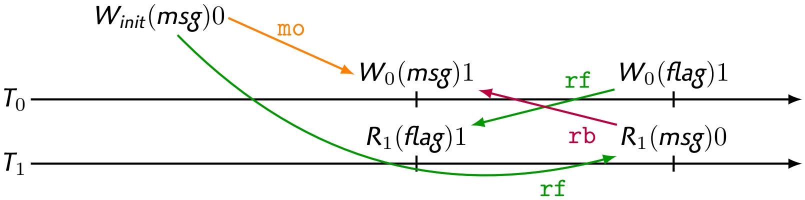
Producer

```c
void produce(void *payload)
{
    msg = payload;
    flag = 1;
}
```

Consumer

```c
void * receive()
{
    while (!flag) {}; // wait
    return msg;
}
```

- Fails on ARM, POWER, ...
- Threads do not necessarily *see* the writes in the same order!



The OpenMP Memory Model

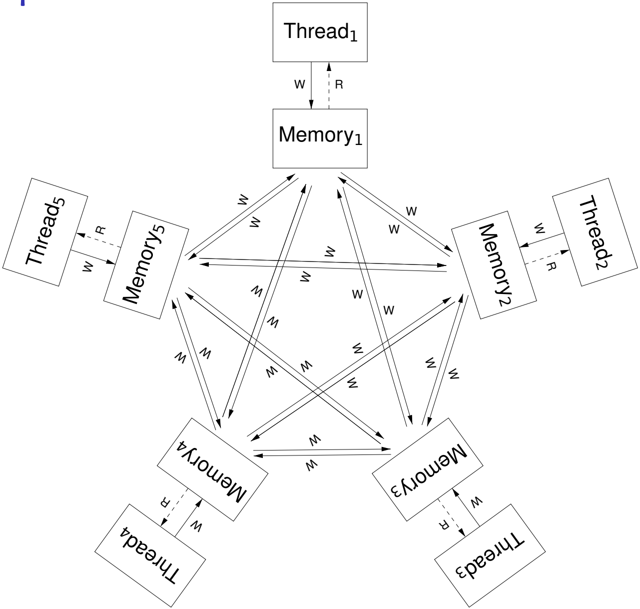
Threads have access to the same memory, but...

Each thread has a private temporary view of the memory

- Not necessarily always synchronized
- A read **may** come from the private temporary view
- A write **may** stay inside the private temporary view
- (Implicit) synchronizations on:
  - `#pragma omp barrier`
  - Exit of `#pragma omp for/sections/single`
  - Entry/exit of `#pragma omp parallel/critical/atomic`
  - *Task Scheduling Points*
- (Explicit) synchronizations with `#pragma omp flush`

- Various reasons, including compilers and the hardware itself
- Atomic operations are **sequentially consistent**

Visual Representation



(image : A Tutorial Introduction to the ARM and POWER Relaxed Memory Models)

Quizz

Initially, $x = y = 0$.

$$\begin{array}{ccc}
R_0(x)0 & W_1(x)1 & R_2(x)1 \\
\downarrow \texttt{po} & \downarrow \texttt{po} & \downarrow \texttt{po} \\
R_0(y)1 & W_1(y)1 & R_3(y)0
\end{array}$$

Possible?

1. No, it is contradictory
2. No, because it is not sequentially consistent
3. Yes, possible on ARM and POWER CPUs but not on x86
4. Yes, it is sequentially consistent

Quizz

Initially, $x = y = 0$.

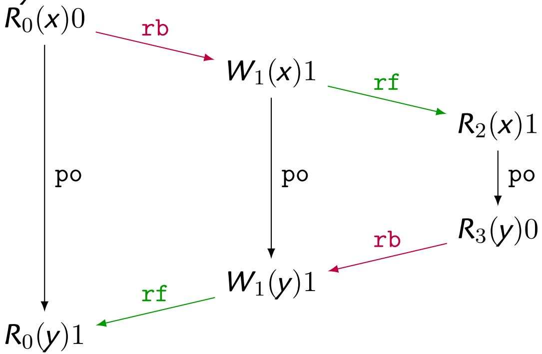
$$\begin{tikzcd}
R_0(x)0 \arrow[r, "\texttt{rb}"] \arrow[d, "\texttt{po}"] & W_1(x)1 \arrow[r, "\texttt{rf}"] \arrow[d, "\texttt{po}"] & R_2(x)1 \arrow[d, "\texttt{po}"] \\
R_0(y)1 & W_1(y)1 \arrow[l, "\texttt{rf}"'] & R_3(y)0 \arrow[l, "\texttt{rb}"']
\end{tikzcd}$$

Possible?

1. No, it is contradictory
2. No, because it is not sequentially consistent
3. Yes, possible on ARM and POWER CPUs but not on x86
4. Yes, it is sequentially consistent

Quizz

Initially, $x = y = 0$.



Possible?

1. No, it is contradictory
2. No, because it is not sequentially consistent
3. Yes, possible on ARM and POWER CPUs but not on x86
4. Yes, it is sequentially consistent

Quizz (harder)

Initially, $x = y = 0$.

$$\begin{tikzcd}[column sep=huge]
R_0(y)1 \arrow[d, "\mathtt{po}"] & W_1(x)1 \arrow[d, "\mathtt{po}"] & R_2(x)1 \arrow[d, "\mathtt{po}"] \\
R_0(x)0 & W_1(y)1 & R_3(y)0
\end{tikzcd}$$

Possible ?

1. No, it is contradictory
2. No, because it is not sequentially consistent
3. Yes, possible on ARM and POWER but not on x86
4. Yes, it is sequentially consistent

Quizz (harder)

Initially, $x = y = 0$.

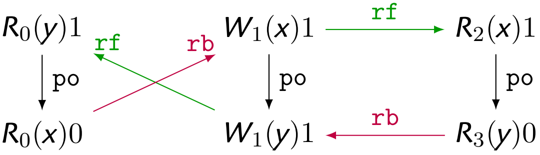
$$\begin{tikzcd}
R_0(y)1 \arrow[d, "\texttt{po}"] & W_1(x)1 \arrow[d, "\texttt{po}"] \arrow[r, "\texttt{rf}"] & R_2(x)1 \arrow[d, "\texttt{po}"] \\
R_0(x)0 \arrow[ur, "\texttt{rb}"] & W_1(y)1 \arrow[ul, "\texttt{rf}"'] & R_3(y)0 \arrow[l, "\texttt{rb}"']
\end{tikzcd}$$

Possible ?

1. No, it is contradictory
2. No, because it is not sequentially consistent
3. Yes, possible on ARM and POWER but not on x86
4. Yes, it is sequentially consistent

Quizz (harder)

Initially, $x = y = 0$.



Possible ?

1. No, it is contradictory
2. No, because it is not sequentially consistent
3. Yes, possible on ARM and POWER but not on x86
4. Yes, it is sequentially consistent

Everyone inevitably goes through this

- Okay, but then if I avoid `i++`, it's okay, right?
- I read in my CPU doc that aligned reads/writes are atomic; if we stick to that, we'll be fine, right?

Teenage crisis

Let's drop the golden rule

Guru switch →

Safety

World of PAIN

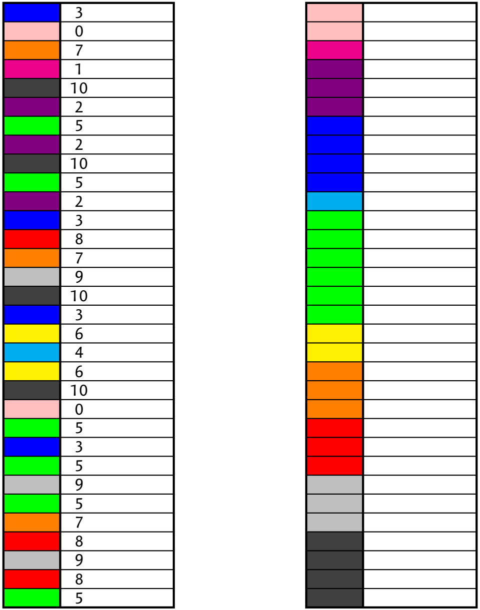
Golden rule of multi-thread programming

ALL potentially conflicting accesses* to **shared** variables **MUST** be protected (`atomic`, `critical`, ...).

∗ at least one of them is a write

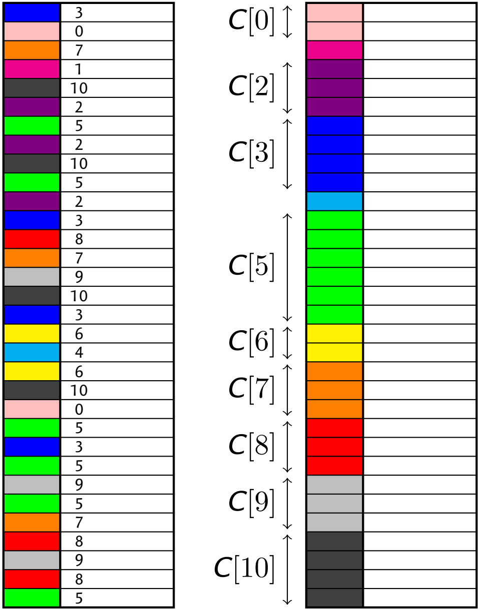
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



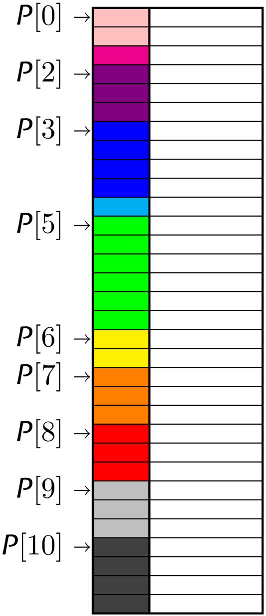
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



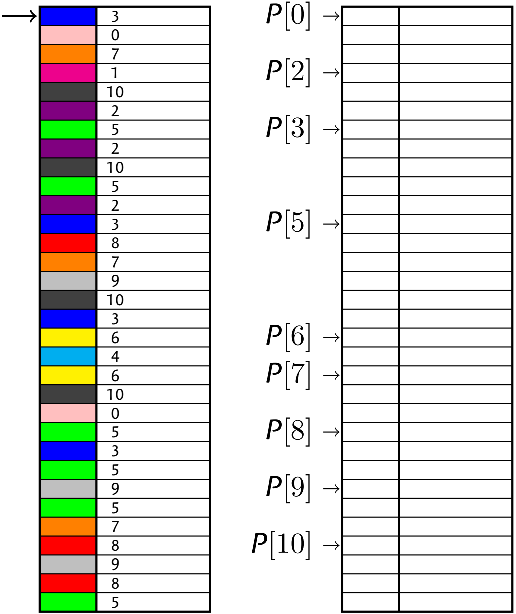
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



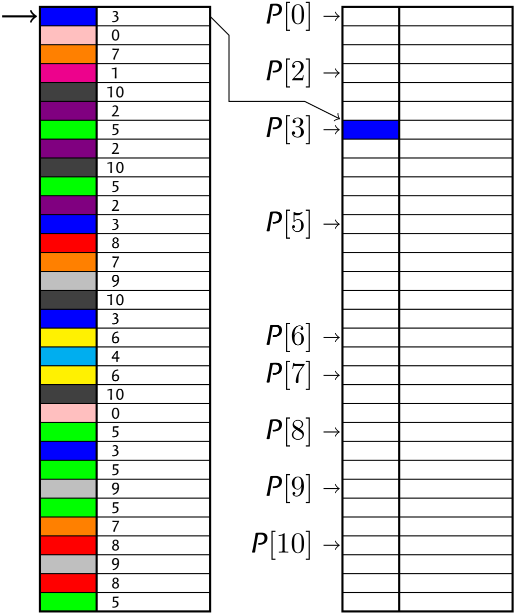
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



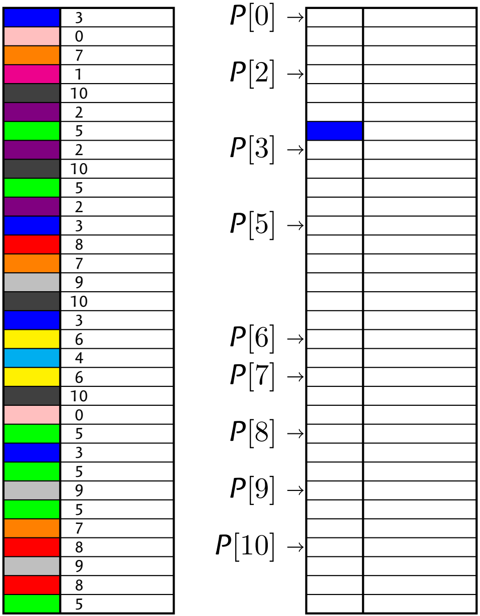
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



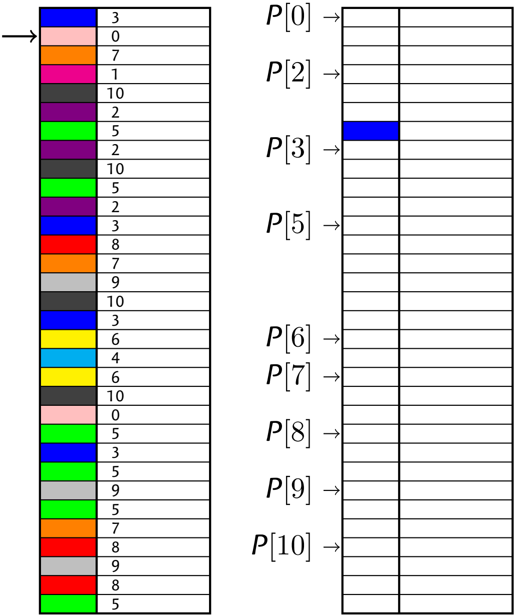
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



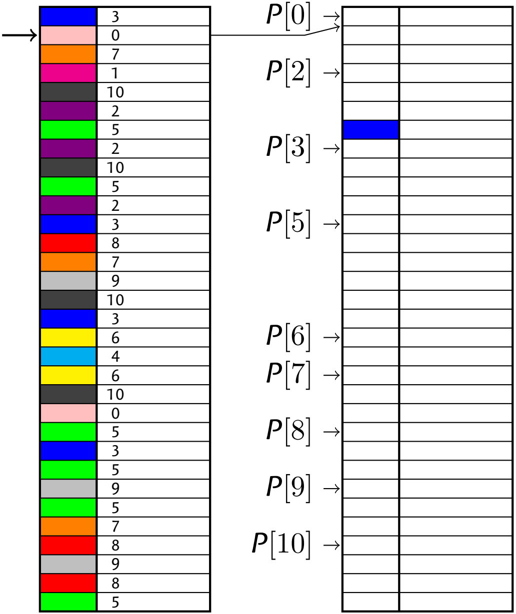
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



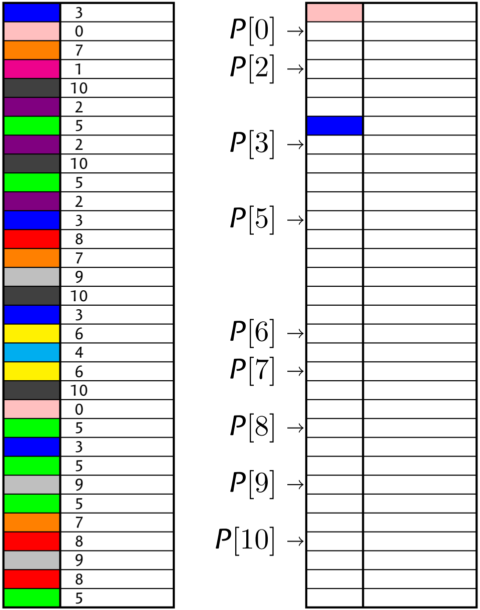
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



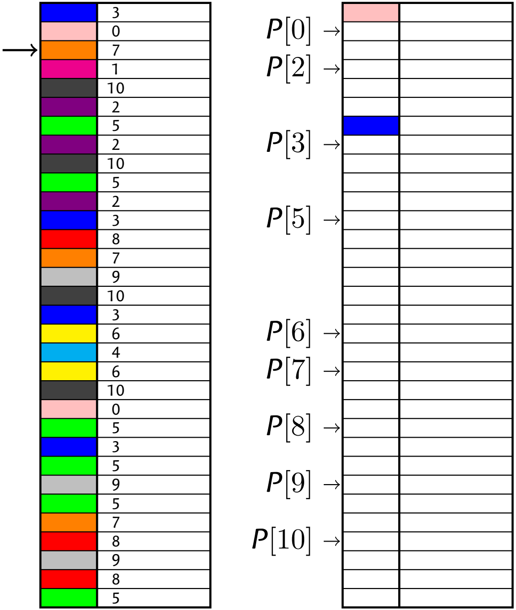
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



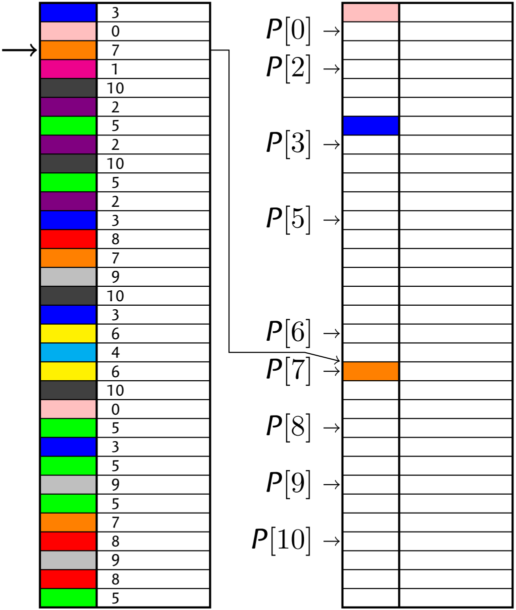
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



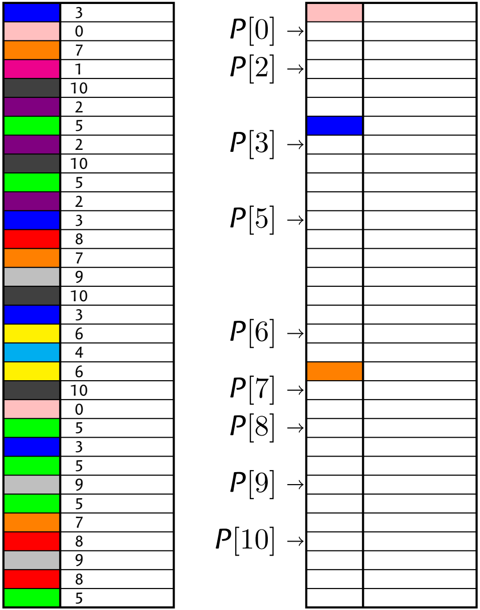
Case Study: Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



Case Study: Bucket Sort

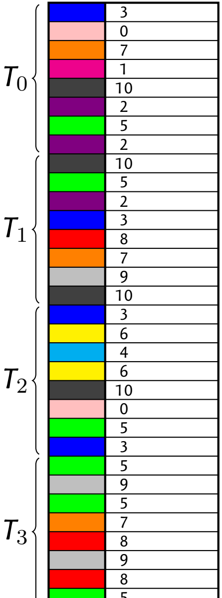
```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



Case Study: Bucket Sort

Direct Naïve Parallelization

```
// Counting

for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch

for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    int ptr;

    ptr = P[bucket]++;
    B[ptr] = A[i];
}
```



Case Study: Bucket Sort

Direct Naïve Parallelization

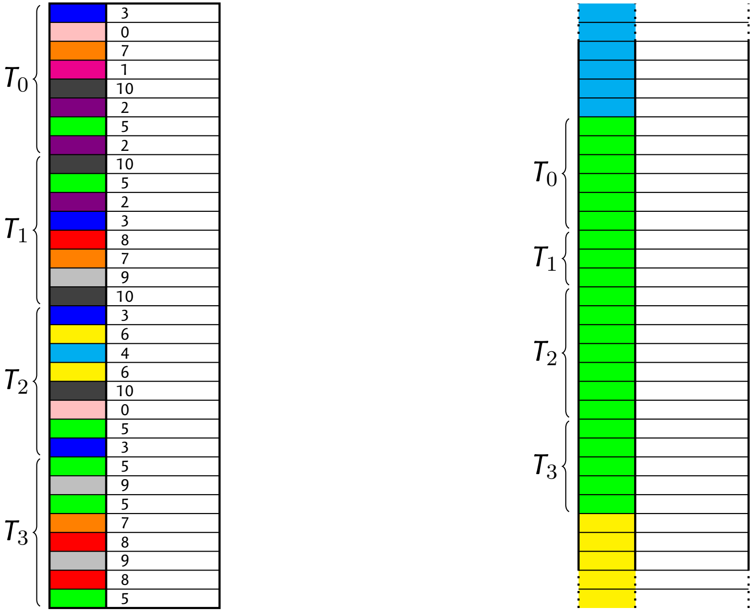
```c
// Counting
#pragma omp parallel for reduction(+:C[0:M])
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum (sequential)
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    int ptr;
    #pragma omp atomic capture
    ptr = P[bucket]++;
    B[ptr] = A[i];
}
```

General Principle n°1: **reorganize**

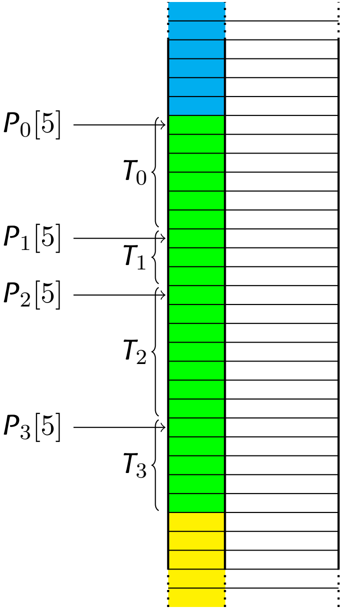
- Do a (tiny) bit of extra computation...

- ... to completely eliminate conflits

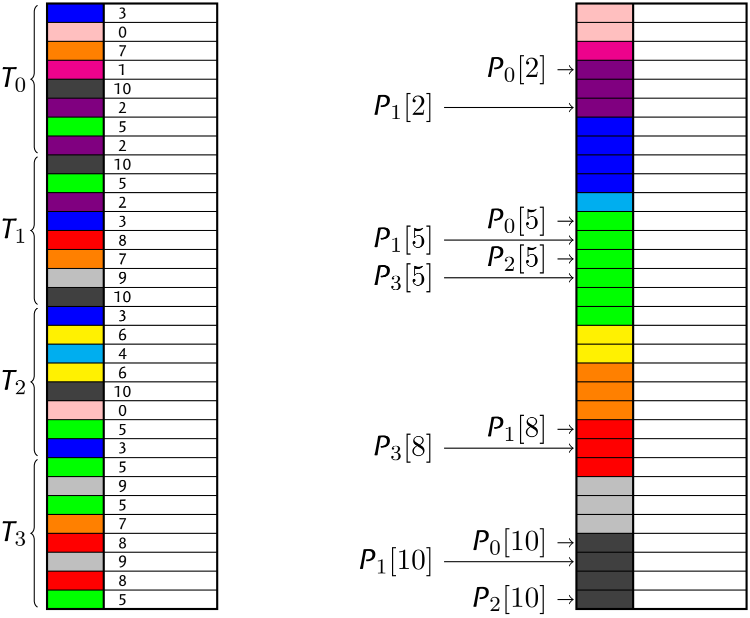
Case Study: Bucket Sort



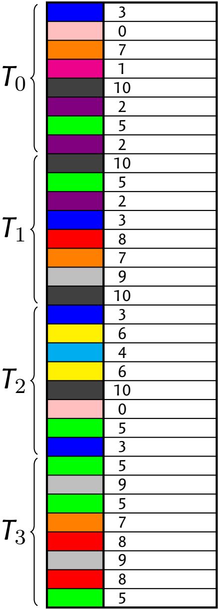
Case Study: Bucket Sort



Case Study: Bucket Sort



Case Study: Bucket Sort

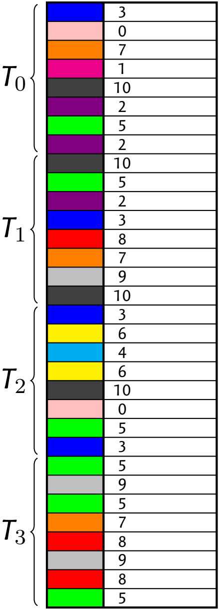


$C_i[x] =$ #items of kind x seen by thread i

Buckets (columns) / Threads (rows)

C	0	1	2	3	4	5	6	7	8	9	10
0	1	1	2	1		1		1			1
1			1	1		1		1	1	1	2
2	1			2	1	1	2				1
3						3		1	2	2	

Case Study: Bucket Sort



$C_i[x] =$ #items of kind x seen by thread i

Buckets

Threads

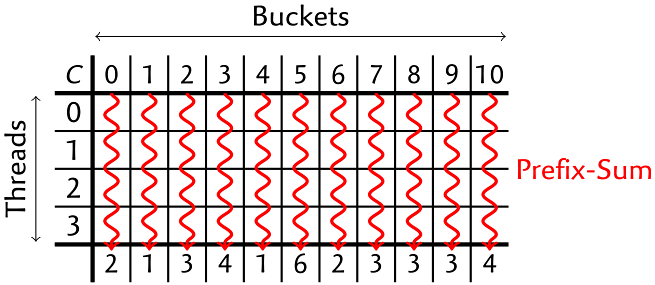
C	0	1	2	3	4	5	6	7	8	9	10
0	1	1	2	1		1		1			1
1			1	1		1		1	1	1	2
2	1			2	1	1	2				1
3						3		1	2	2	
	2	1	3	4	1	6	2	3	3	3	4

Sum

Size of each bucket

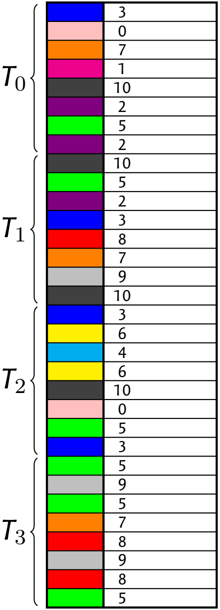
Case Study: Bucket Sort

$C_i[x] =$ #items of kind x seen by thread i



Size of each bucket

Case Study: Bucket Sort



$C_i[x] =$ #items of kind x seen by threads $< i$

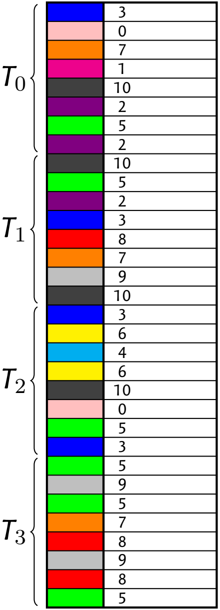
Buckets

Threads

C	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	2	1	0	1	0	1	0	0	1
2	1	1	3	2	0	1	0	2	1	1	3
3	2	1	3	4	1	3	2	2	1	1	4
	2	1	3	4	1	6	2	3	3	3	4

Size of each bucket

Case Study: Bucket Sort



$C_i[x] =$ #items of kind x seen by threads $< i$

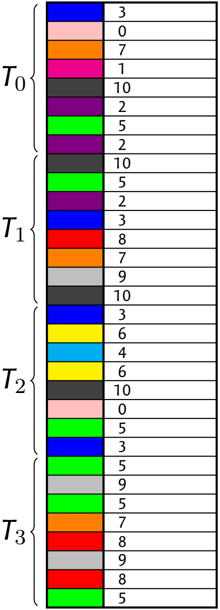
Buckets

Threads

C	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	2	1	0	1	0	1	0	0	1
2	1	1	3	2	0	1	0	2	1	1	3
3	2	1	3	4	1	3	2	2	1	1	4
	2	1	3	4	1	6	2	3	3	3	4

Prefix-Sum

Case Study: Bucket Sort



$C_i[x] =$ #items of kind x seen by threads $< i$

Buckets

Threads

C	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	2	1	0	1	0	1	0	0	1
2	1	1	3	2	0	1	0	2	1	1	3
3	2	1	3	4	1	3	2	2	1	1	4
	0	2	3	6	10	11	17	19	22	25	28

Start indice of each bucket
(#items in previous buckets)

Case Study: Bucket Sort

$C_i[x] =$ #items of kind x seen by threads $< i$

Buckets

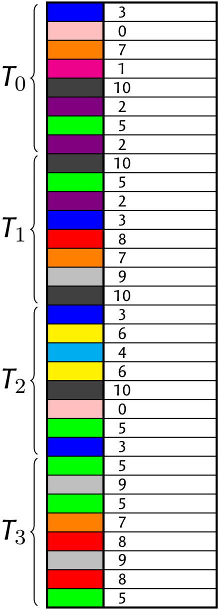
Threads

C	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	2	1	0	1	0	1	0	0	1
2	1	1	3	2	0	1	0	2	1	1	3
3	2	1	3	4	1	3	2	2	1	1	4
	0	2	3	6	10	11	17	19	22	25	28

Sum

Start indice of each bucket
(#items in previous buckets)

Case Study: Bucket Sort



$C_i[x] =$ Start indice of kind x for thread i

Buckets

Threads

C	0	1	2	3	4	5	6	7	8	9	10
0	0	2	3	6	10	11	17	19	22	25	28
1	1	3	5	7	10	12	17	20	22	25	29
2	1	3	6	8	10	12	17	21	23	26	31
3	2	3	6	10	11	14	19	21	23	26	32
	0	2	3	6	10	11	17	19	22	25	28

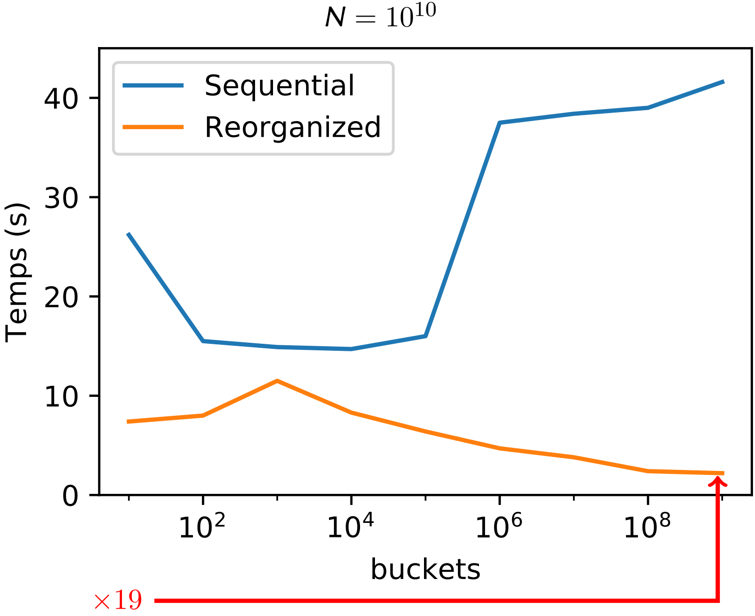
Start indice of each bucket
(#items in previous buckets)

Case Study: Bucket Sort

```
int C[T][M], S[M];

#pragma omp parallel
{
  int t = omp_get_thread_num();

  // Counting
  for (int i = 0; i < M; i++)
      C[t][i] = 0;
  #pragma omp for schedule(static)
  for (int i = 0; i < N; i++) {
      int bucket = f(A[i]);
      C[t][bucket]++;

  // <<COMPUTE POINTERS>> ------------>

  // Dispatch
  #pragma omp for schedule(static)
  for (int i = 0; i < N; i++) {
      int bucket = f(A[i]);
      int ptr = C[t][bucket]++;
      B[ptr] = A[i];
  }
}
```

```
// sum (columns)
#pragma omp for
for (int i = 0; i < M; i++) {
    S[i] = 0;
    for (int j = 0; j < T; j++)
            S[i] += C[j][i];
}

// horizontal prefix-sum (sequential)
#pragma omp single
{   int s = 0;
    for (int i = 0; i < M; i++) {
        int t = S[i];
        S[i] = s;
        s += t;
}   }

// prefix-sum (columns)
#pragma omp for
for (int i = 0; i < M; i++) {
    int s = S[i];
    for (int j = 0; j < T; j++) {
        int t = C[j][i];
            C[j][i] = s;
            s += t;
}   }
```

Case Study: Bucket Sort



An array of integers to sort?

Pro Tip

Practical parallel sorting algorithm

- *Parallel Bucket Sort* on the 8 most significant bits
- For $0 \leq i < 2^8$, do (in parallel) :
  - Sort the i-th Bucket (using a normal sequential sort)

Parallel Transactions

Read $A[i_1], A[i_2], \ldots \rightarrow$ **Compute** $\rightarrow$ **Write** $A[k_1], A[k_2], \ldots$.

Obstacle to "atomic" execution:

- The read data was modified before the writes
- Result of the computation is "outdated"

Pessimistic approach (*"Ask for Permission"*)

- "Lock" read data
- Read/Acquire locks $\rightarrow$ Compute $\rightarrow$ Write $\rightarrow$ Release locks
  - Prevent **potential** modification by other threads
- Assume that conflict **WILL** take place
- Useless overhead in the absence of conflict

Parallel Transactions

Read $A[i_1], A[i_2], \ldots \rightarrow$ **Compute** $\rightarrow$ **Write** $A[k_1], A[k_2], \ldots$.

Obstacle to "atomic" execution:

- The read data was modified before the writes
- Result of the computation is "outdated"

Optimistic Approach (*"Shoot First, Ask Questions Later"*)

- Read (without precaution!!!) $\rightarrow$ Compute $\rightarrow$ **Commit**:
  - Check freshness of read data,
  - If OK (=unmodified), write; otherwise, restart from the beginning
- Assume that conflict **WILL NOT** take place
- Lost work in case of conflict

General Principle: **Analyze Conflict Frequency**

- Take the risk to waste a little bit of computation...

- ... To reduce the cost of handling conflicts

Generic Technique: *Versioning*

- Shared data structure, with a version number
- $v = 0$ initially (odd during writes)

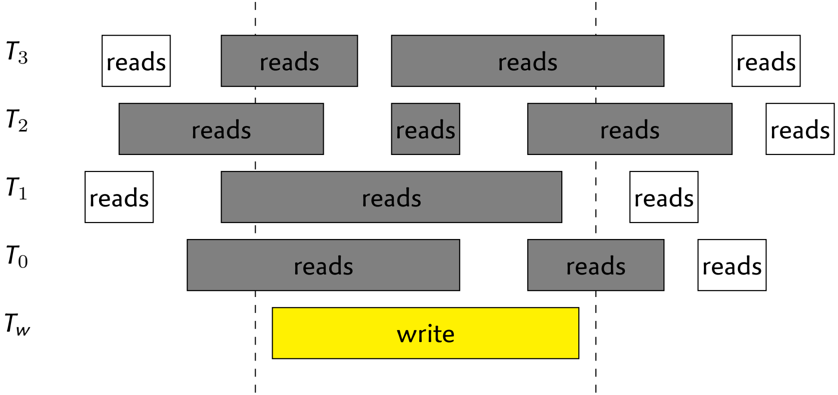
Writer

1. Enter critical section; increment v
2. Do the writes
3. increment v; exit critical section

Reader

1. $v_{before} \leftarrow v$
2. Do the reads
3. $v_{after} \leftarrow v$
4. If v_{before} is odd or $v_{before} \neq v_{after}$, retry

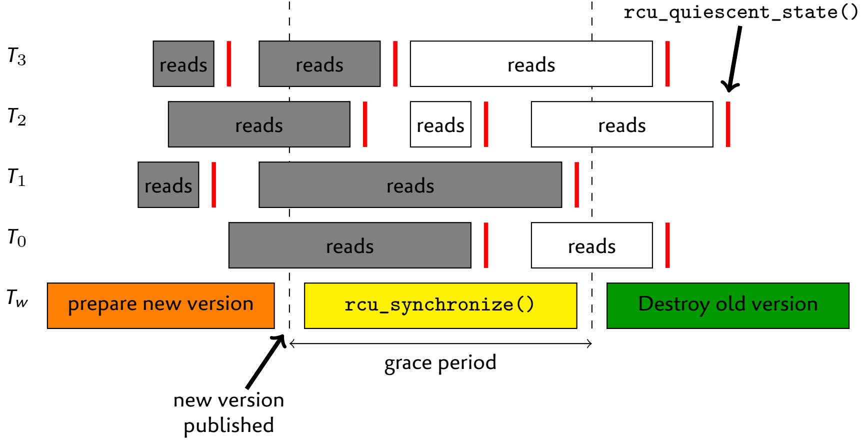
Generic Technique: *Versioning*



- Writers have priority over readers

Generic Technique: *Read-Copy-Update*

- Shared data structure, readers access via a pointer
- Readers grab the pointer, read, call `rcu_quiescent_state()`
- Writers make new copy, atomically update the pointer
- Cannot release old copy right now (readers still read it)



```
int tid = omp_get_num_thread();
int N = omp_get_max_threads();
int gc = 1;
int rc[N];  // {0, ..., 0}
```

```
void rcu_quiescent_state()
{
  #pragma omp atomic write
  rc[tid] = gc;
}

void rcu_thread_offline()
{
  #pragma omp atomic write
  rc[tid] = 0;
}

void rcu_thread_online()
{
  #pragma omp atomic write
  rc[tid] = gc;
}
```

```
void rcu_synchronize()
{
  bool was_online = (rc[tid] > 0);
  if (was_online)
    rcu_thread_offline();
  #pragma omp critical
  {
    #pragma omp atomic update
    gc++;
    for (int i = 0; i < N; i++)
      for (;;) {
        int s;
        #pragma omp atomic read
        s = rc[i];
        if (s == 0 || s == gc)
          break;
      }
  }
  if (was_online)
    rcu_thread_online();
}
```

Transactional Memory

- Similar problems in database servers
- Many concurrent transactions management techniques

(very) modern CPUs:
transactional memory

```c
#include <immintrin.h>
unsigned int status = _xbegin();
if (status == _XBEGIN_STARTED) {
    // Access shared data ...
    if (problem)     // give up ?
        _xabort(0);
    // Access more shared data ...
    _xend();
    /* <-------- Success !!! */
} else { /* <--- Failure */
    if (status & _XABORT_EXPLICIT)
        ...
    if (status & _XABORT_CONFLICT)
        ...
    if (status & _XABORT_CAPACITY)
        ...
}
```

- `_xbegin()` starts a transaction
  - Returns `_XBEGIN_STARTED`
  - Flush the cache...
- `_xend()` attemps to "commit"
  - OK $\rightarrow$ execution continues
- `_xabort(cst)` aborts transaction
- In case of failure:
  - Returns after `_xbegin()`
  - Error code (conflict, resources, ...)
- Still not a panacea
  - Non-negligible cost
  - False positives, ...
- See also `TinySTM` library