# iThreads: A Threading Library for Parallel Incremental Computation

## Paper Reading Group

Pramod Bhatotia
Pedro Fonseca
Umut A. Acar
Björn B. Brandenburg
Rodrigo Rodrigues

Presents: Maksym Planeta

09.07.2015

# Table of Contents

# Goals

Make incremental computations easy to use:

- Convenient for user
- Legacy support
- Language independent
- No programmer intervention
- Multithreaded environment
- Use existing OS facilities
- Generic program model
- Low overhead

# Workflow

1. Inital run
2. Build Concurrent Dynamic Dependence Graph (CDDG)
3. Specify input changes
4. Incremental run uses change propagation
5. Update CDDG

# Workflow

1. Inital run
2. Build Concurrent Dynamic Dependence Graph (CDDG)
3. Specify input changes
4. Incremental run uses change propagation
5. Update CDDG

```
$ LD_PRELOAD=iThreads.so                          // preload iThreads
$./<program_executable> <input-file>              // initial run
$ emacs <input-file>                              // input modified
$ echo "<off> <len>" >> changes.txt               // specify changes
$./<program_executable> <input-file>              // incremental run
```

**Figure 1.** How to run an executable using `iThreads`

# Table of Contents

# System model

- Memory model
  - Release consistency
- Synchronization model
  - pthreads API
- Deterministic behavior

# Thunk

- Unit of sequential execution
- Surrounded by synchronization operations
- State
- Read and write sets
- Causally ordered (vector clocks)
- Thunk recomputed $\Rightarrow$ All thunks in the thread recomputed

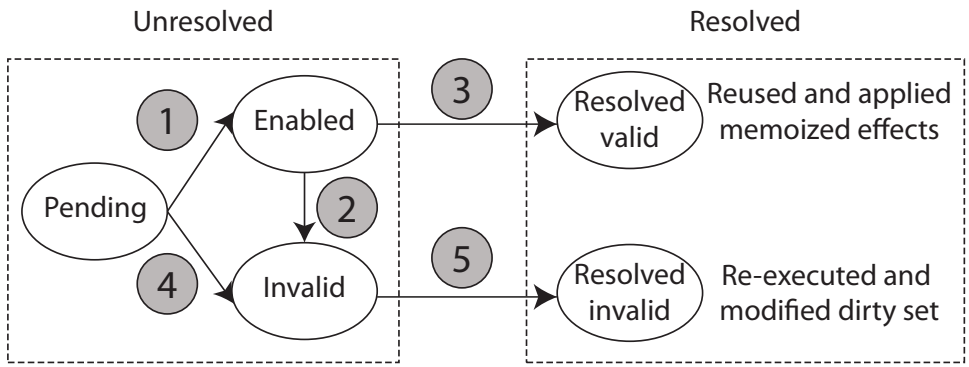


**Figure 4.** State transition for thunks during incremental run

# Example

| Case | Input | Thread schedule | Sub-computations | |
|---|---|---|---|---|
| | | | **Reused** | **Recomputed** |
| *A* | $x, y^*, z$ | $T_1.a \rightarrow T_2.a \rightarrow T_2.b$ | $T_2.a$ | $T_1.a, T_2.b$ |
| *B* | $x, y, z$ | $(T_2.a \rightarrow T_2.b \rightarrow T_1.a)^*$ | $T_2.a$ | $T_1.a, T_2.b$ |
| *C* | $x, y, z$ | $T_1.a \rightarrow T_2.a \rightarrow T_2.b$ | $T_1.a, T_1.b, T_2.a$ | — |

**Figure 3.** For the incremental run, some cases with changed input or thread schedule (changes are marked with *)

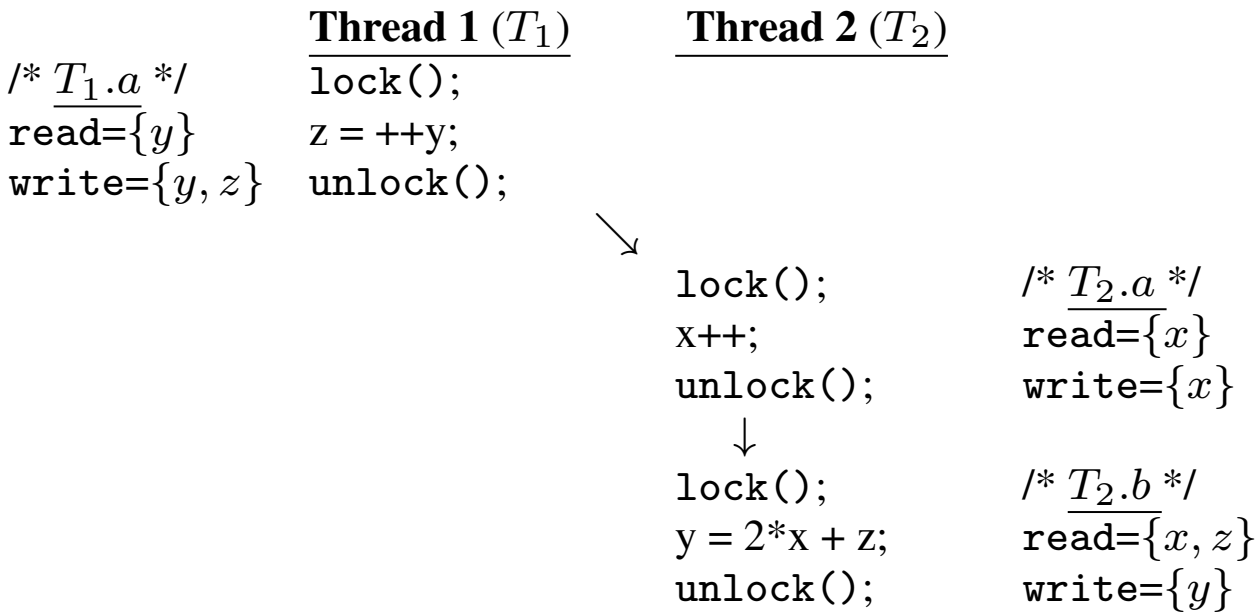


**Figure 2.** An example of shared-memory multithreading

# Architecture

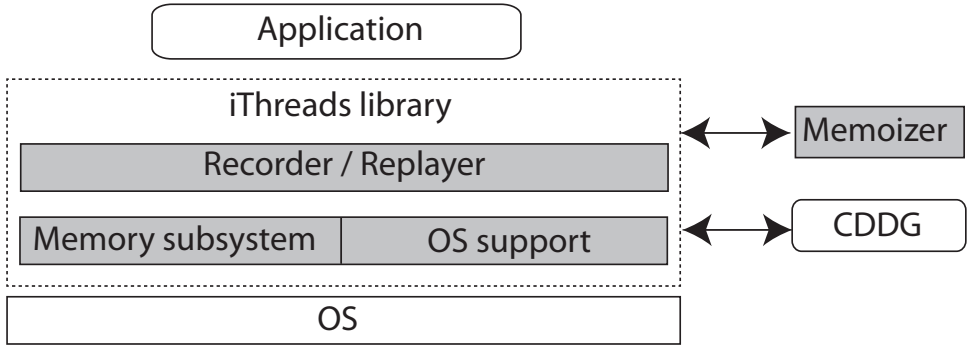


**Figure 5.** `iThreads` implementation architecture. Shaded boxes represent the main components of the system.

# Implementation

- Dthreads
- Separate address spaces for threads
- Page read/write protection
- Byte-level delta

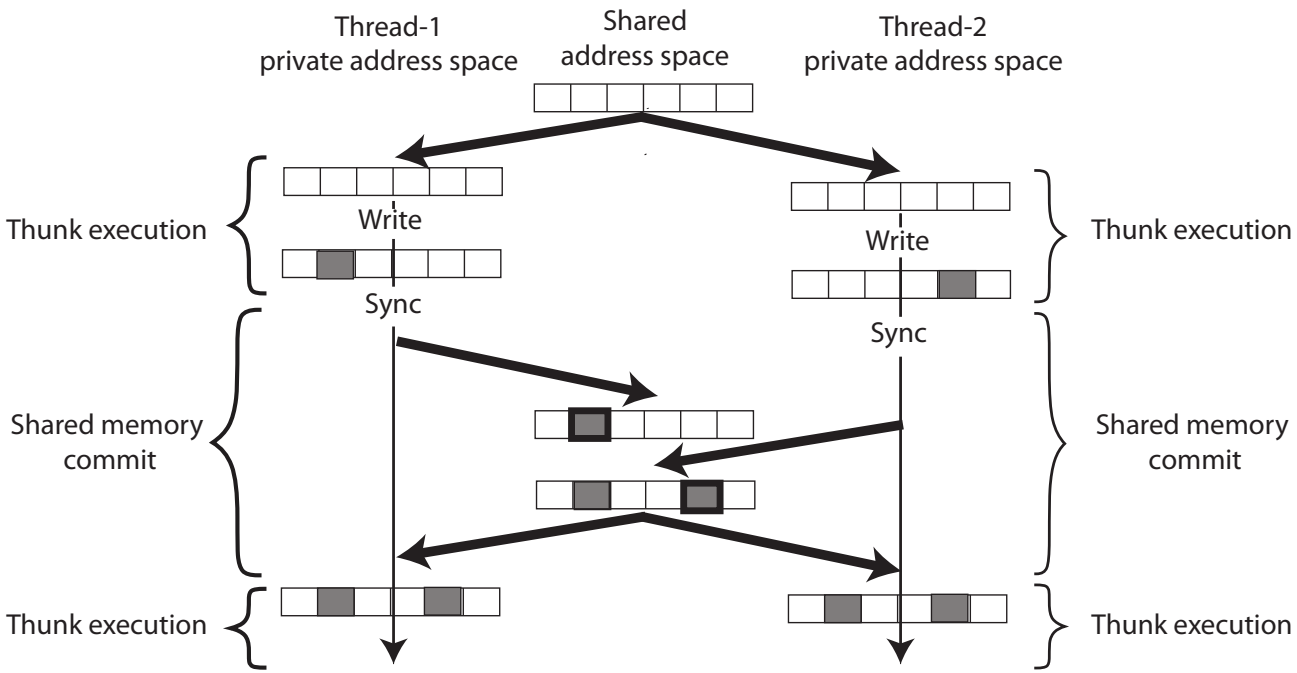


**Figure 6.** Overview of the RC model implementation

# Table of Contents

# Metrics

**Time** runtime of the slowest thread

**Work** sum of the total runtime of all threads

Benchmarks: PARSEC and Phoenix

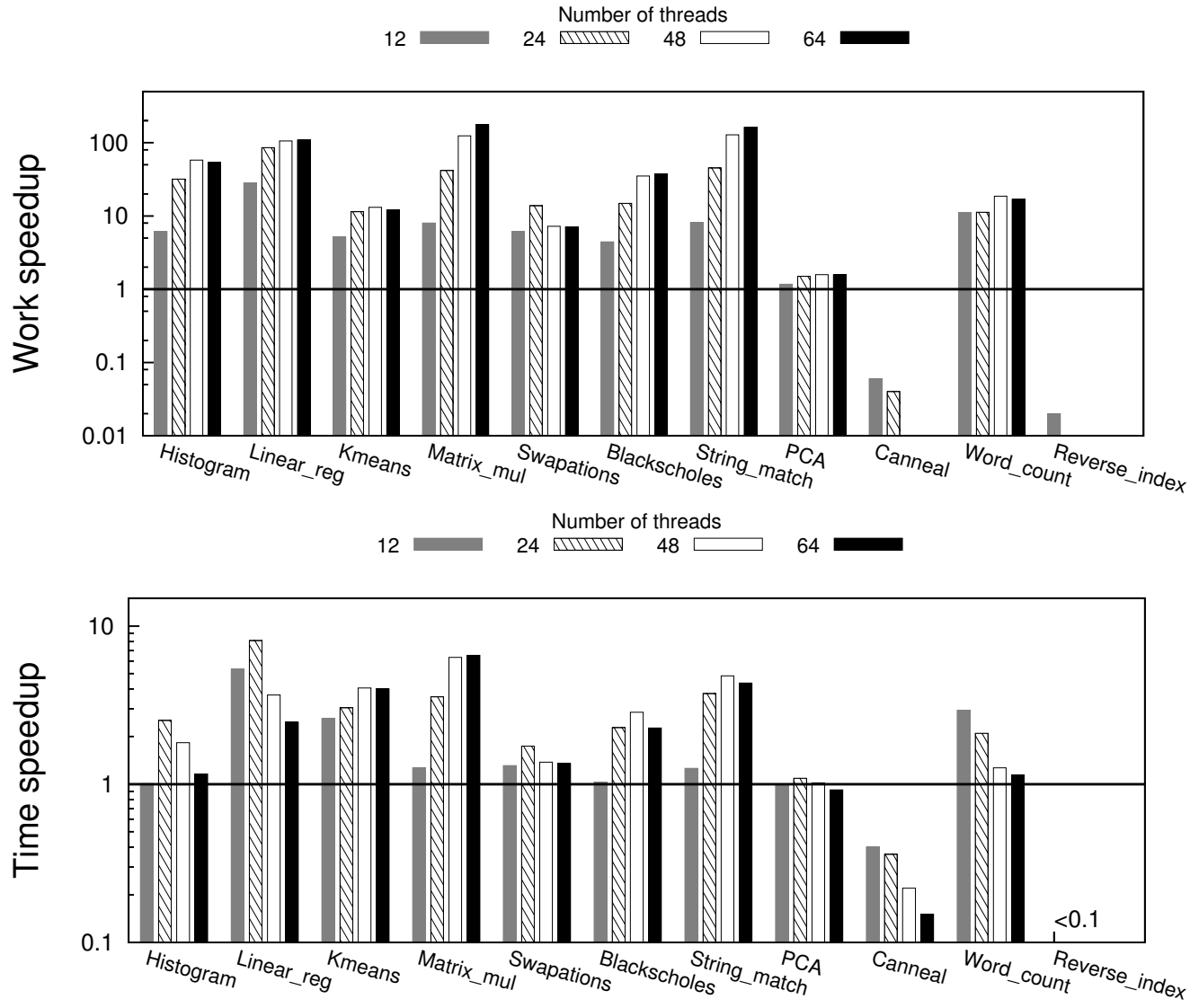


**Figure 7.** Performance gains of `iThreads` with respect to `pthreads` for the incremental run

# Single modified page

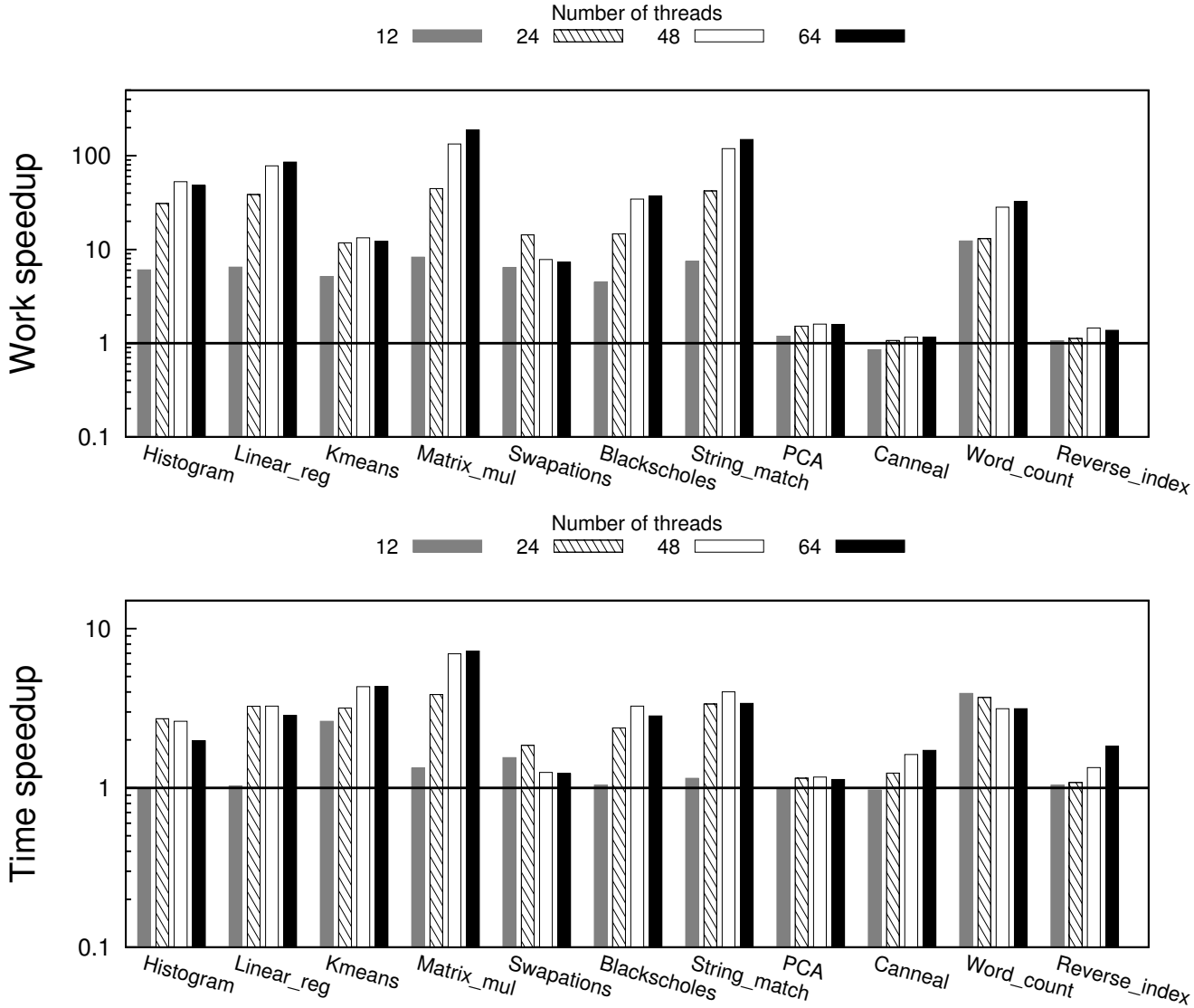


**Figure 8.** Performance gains of `iThreads` with respect to `Dthreads` for the incremental run

# Single modified page, different input sizes

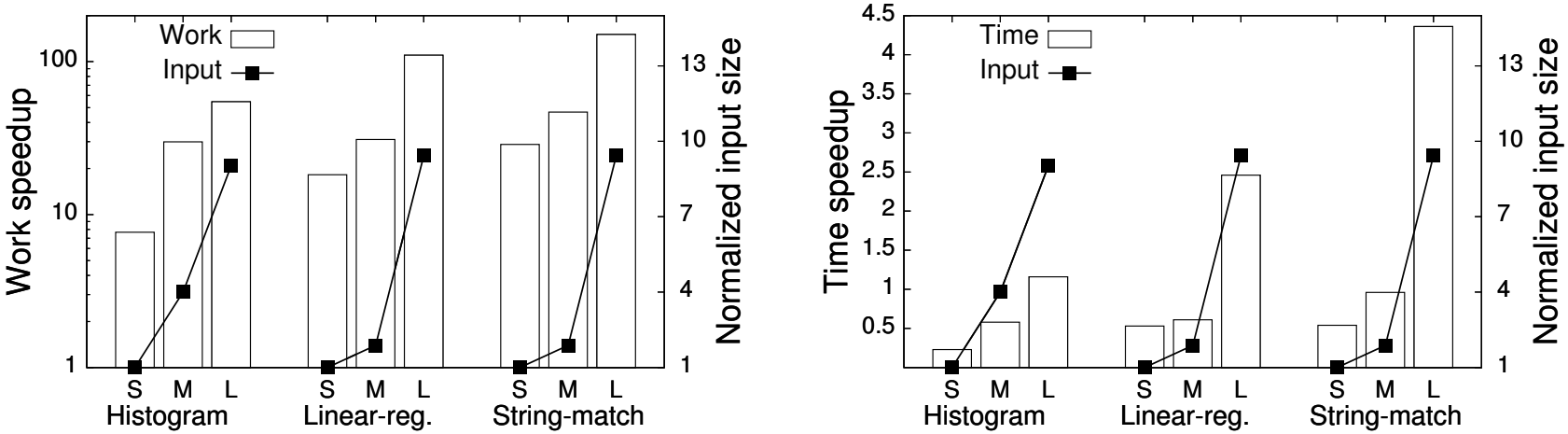


**Figure 9.** Scalability with data (work and time speedups)

# Single modified page, different work amount

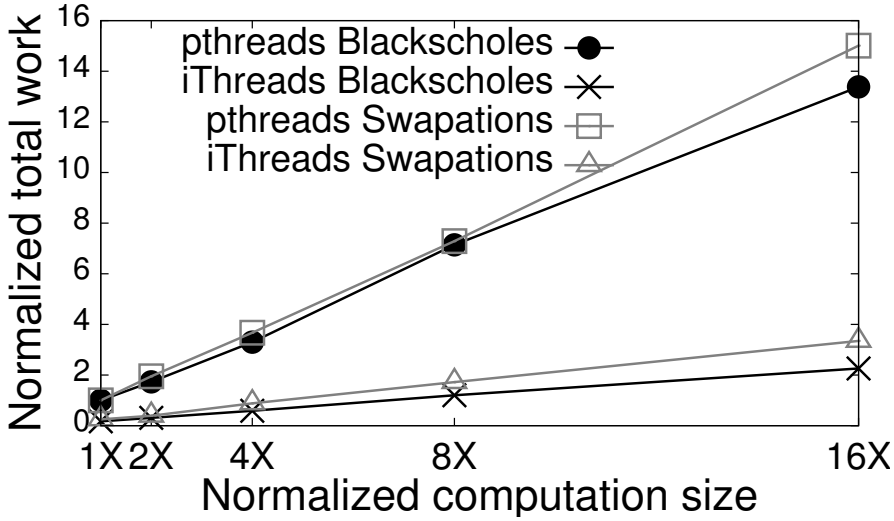


**Figure 10.** Scalability with work

# Several modified pages

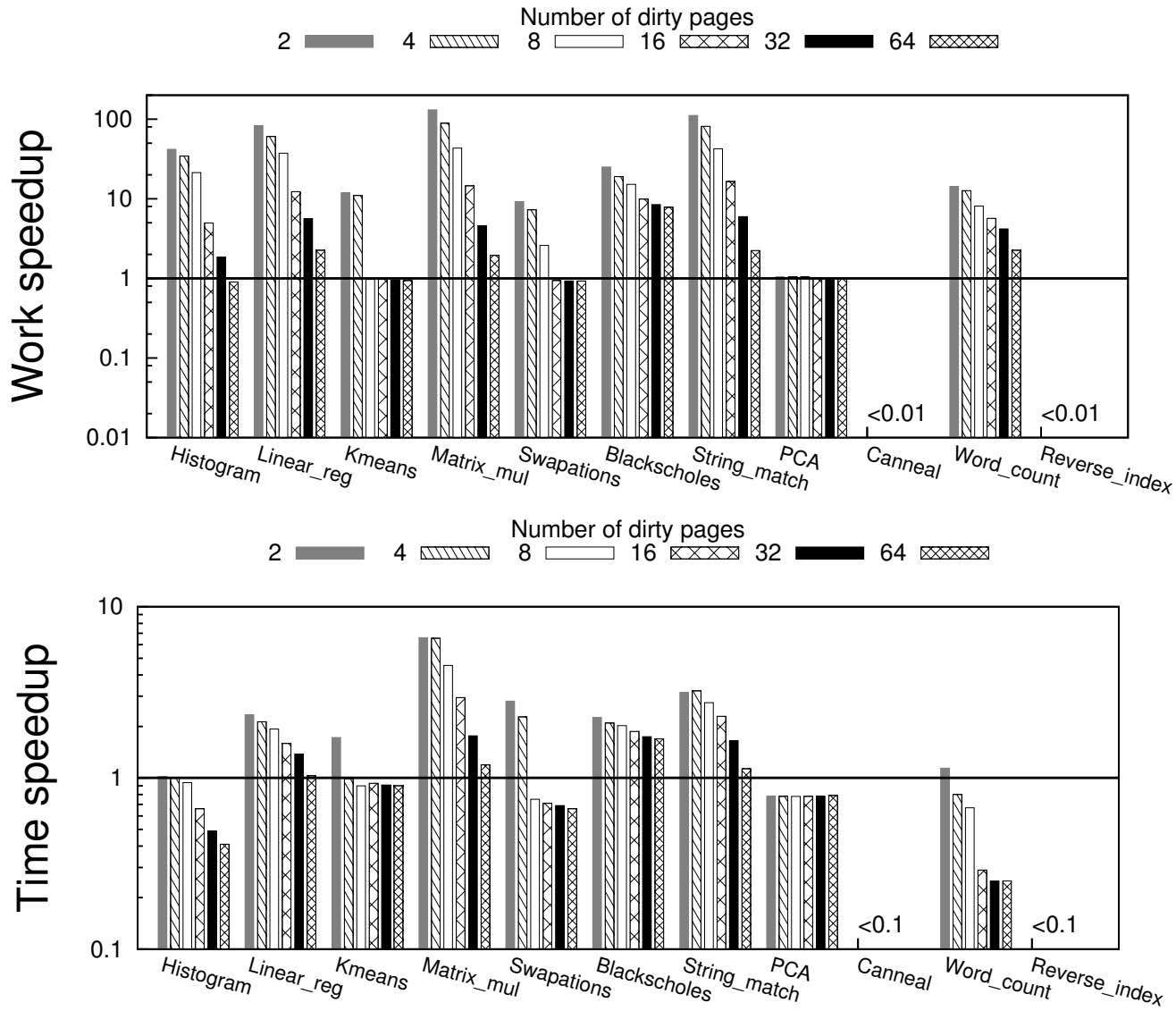


**Figure 11.** Scalability with input change compared to `pthreads` for 64 threads

# Overhead of iThreads system data

| Application | Input size | Memoized state | | CDDG | |
|---|---|---|---|---|---|
| Histogram | 230400 | 347 | (0.15%) | 57 | (0.02%) |
| Linear-reg. | 132436 | 192 | (0.14%) | 33 | (0.02%) |
| Kmeans | 586 | 1145 | (195.39%) | 27 | (4.61%) |
| Matrix-mul. | 41609 | 4162 | (10.00%) | 64 | (0.15%) |
| Swapations | 143 | 1473 | (1030.07%) | 1 | (0.70%) |
| Blackscholes | 155 | 201 | (129.68%) | 1 | (0.65%) |
| String match | 132436 | 128 | (0.10%) | 33 | (0.02%) |
| PCA | 140625 | 3777 | (2.69%) | 43 | (0.03%) |
| Canneal | 9 | 15381 | (170900.00%) | 4 | (44.44%) |
| Word count | 12811 | 10191 | (79.55%) | 24 | (0.19%) |
| Rev-index | 359 | 260679 | (72612.53%) | 64 | (17.83%) |

**Table 1.** Space overheads in pages and input percentage

# Initial run overhead

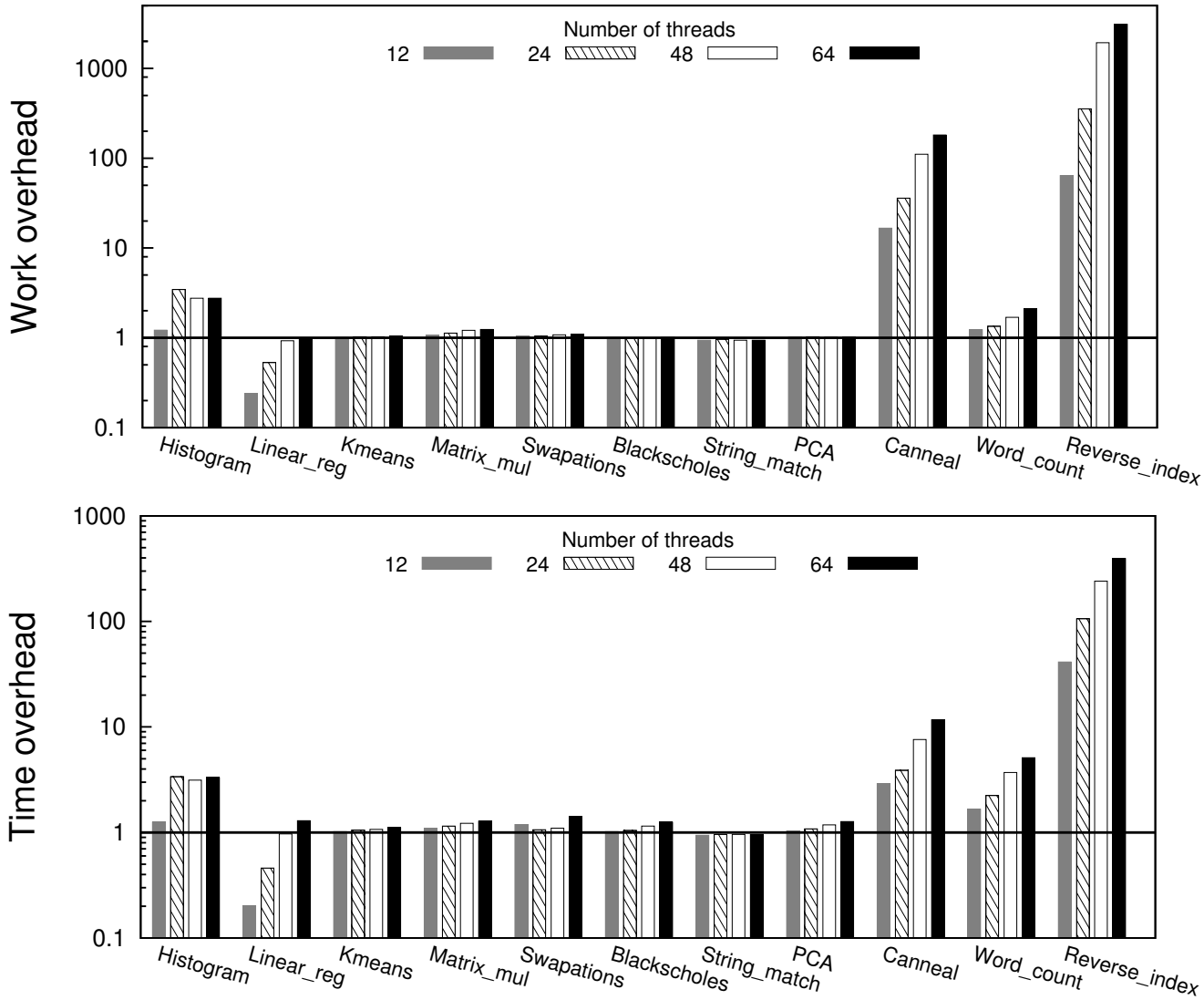


**Figure 12.** Performance overheads of `iThreads` with respect to `pthreads` for the initial run

# Initial run overhead

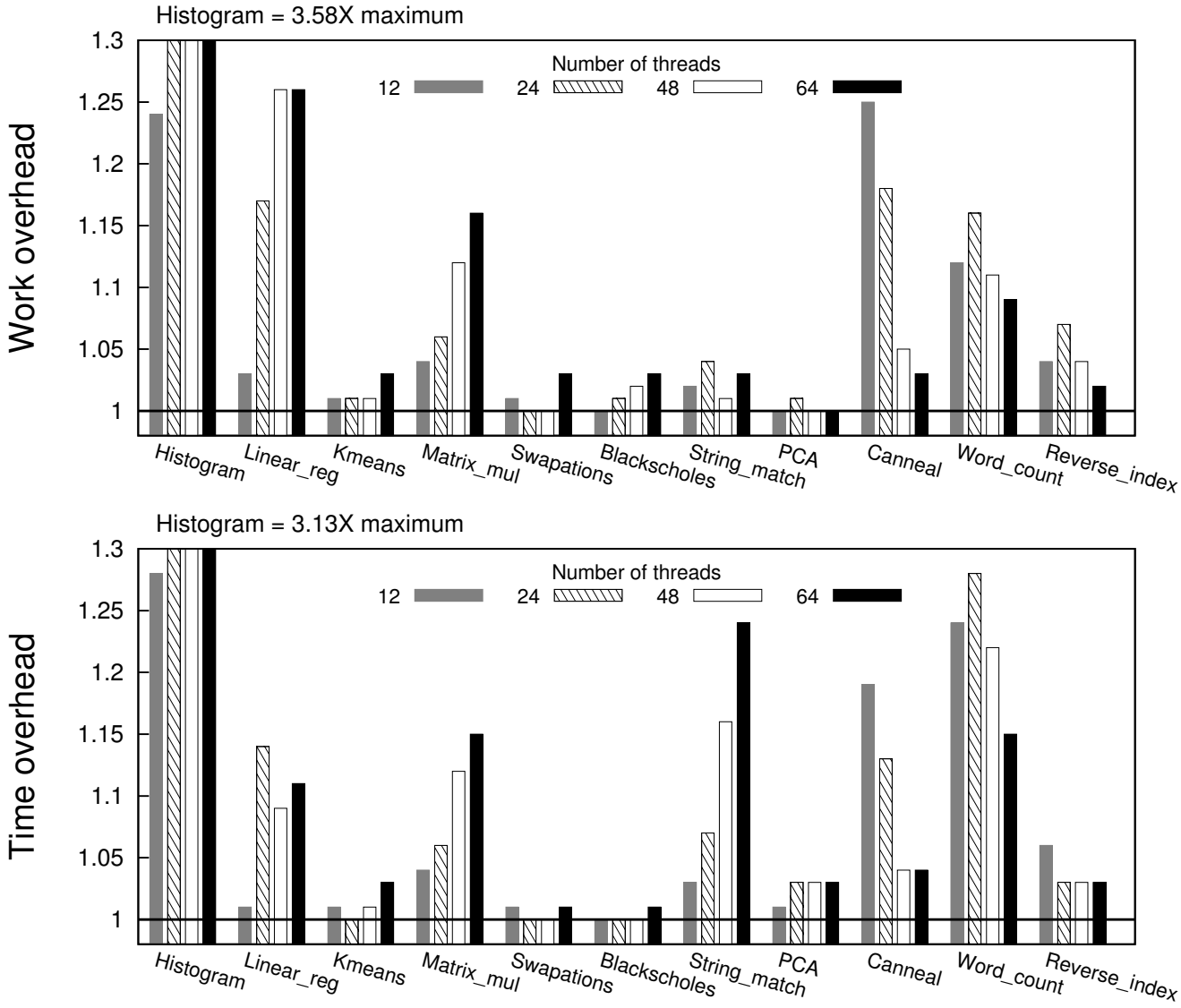


**Figure 13.** Performance overheads of `iThreads` with respect to `Dthreads` for the initial run

# Case-study applications

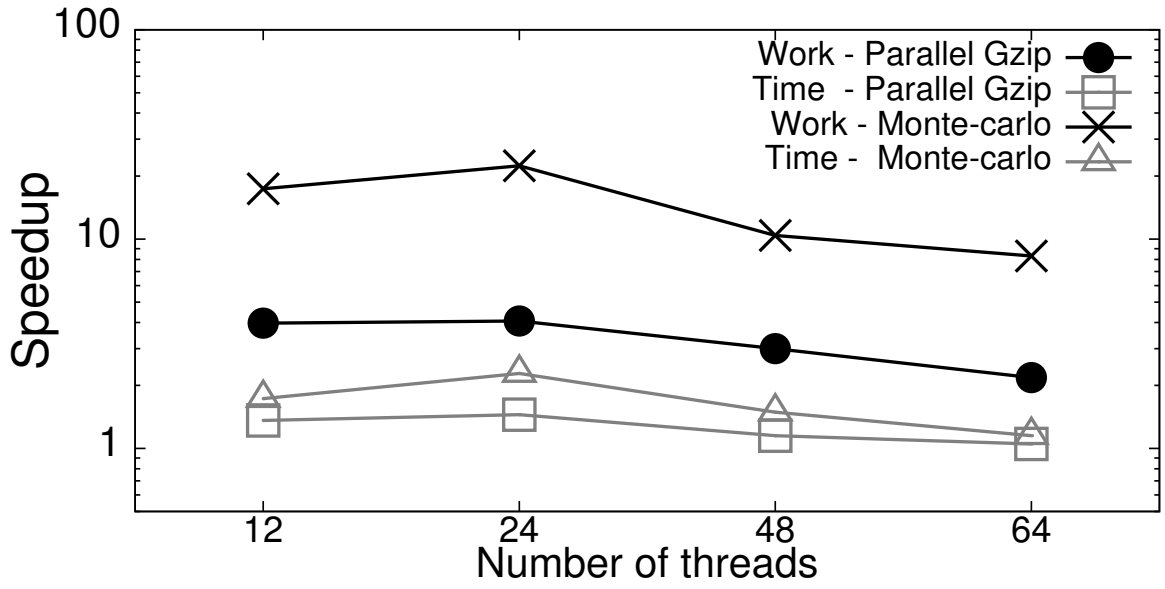


**Figure 15.** Work & time speedups for case-studies

# Table of Contents

# Limitations

- No support for ad-hoc synchronization
  - No C++ atomics
- No support for small localized insertions
- Assumes constant amount of threads
- May have significant overhead
- Narrow application area

# Outcome

- Nice idea
- **Practical**
- **Transparent**
- **Efficient**
- Works for some applications
- Way significantly decrease required work

# Discussion

- Units for scales are not specified:
  Sometimes *percentage*, sometimes *times*
- Interactive applications
- Vector clock for each thunk – not too much?
- IO memory – can you do something? For instance frame buffer.
- Can be combined with dynamic algorithms?

# Explanation of Dthreads high overhead

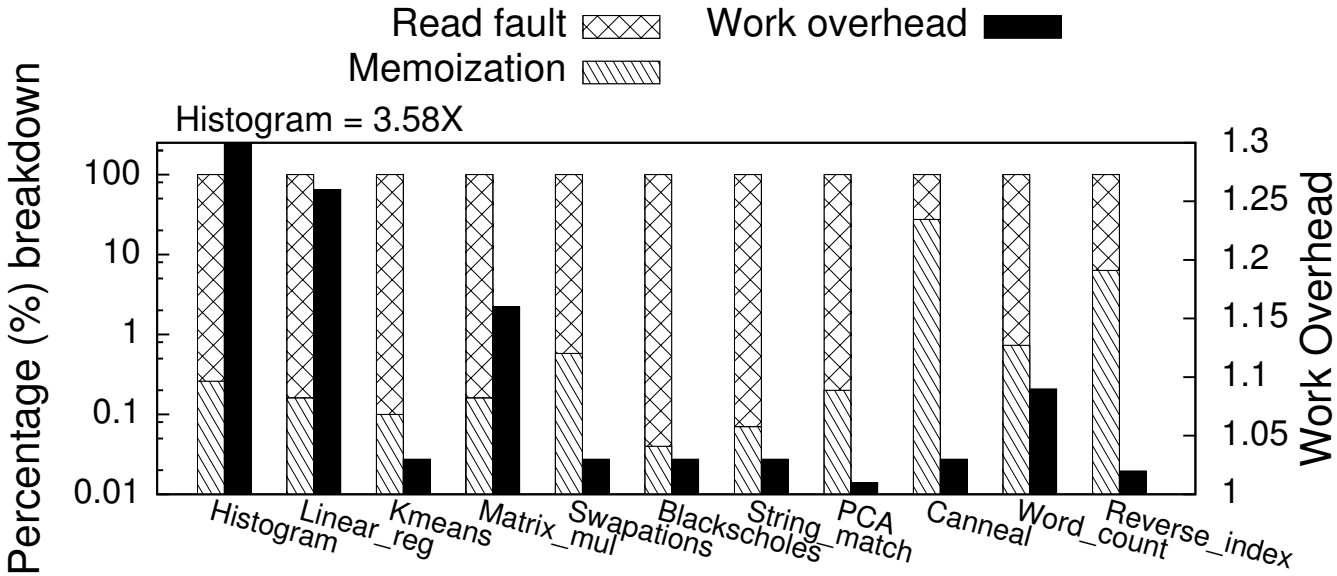


**Figure 14.** Work overheads breakdown w.r.t `Dthreads`

# Release consistency

- Objects are *acquired* and *released*
- Critical section between *acquire* and *release*
- Guaranteed correctness and liveness for data-race-free programs

# Vector clocks

- Used for invalidation propagation
- Maintained for:
  - Objects
  - Threads
  - Thunks