Implementing Arbitrary/Common Concurrent Writes of CRCW PRAM

Fady Ghanim, Wael Elwasif, David Bernholdt
Oak Ridge National Laboratories
Introduction

• Parallel Random Access Machines (PRAM) provides a convenient and powerful parallel computational model to study the algorithmic complexity of parallel algorithms

• Many believe that CRCW PRAM algorithms are impossible to support on general-purpose parallel machines.

• Recent research has refuted these claims
  – A new specialized architecture designed around principles of PRAM
  – High level compiler to enable implementing PRAM algorithms as is.
Parallel Random Access Machines (PRAM)

• A shared memory abstraction machine - The Parallel analog to the random access machines

• PRAM makes certain assumptions to abstract the underlying architecture:
  – Unlimited shared memory
  – Unlimited processors
  – Uniform memory access to any memory location from all processors
  – Perfect synchrony between all processors – execution is lockstep

• In PRAM an algorithm is described as a series of rounds/time-steps within which multiple operations are executed concurrently.
PRAM Memory Access Modes

• PRAM memory access modes are:
  – Exclusive Read Exclusive Write (EREW)
  – Concurrent Read Exclusive Write (CREW)
  – Concurrent Read Concurrent Write (CRCW)

• CRCW PRAM conflict resolution strategies
  – Priority Concurrent Write
  – Arbitrary Concurrent Write
  – Common Concurrent Write
Motivating Example

P: is a parent node
L: is the round when a node was visited
Other examples

- Topological sort
- Connected components
- Constant time Maximum / minimum
- All Nearest Smaller Value (ANSV)
- String matching
- Maximal matching
- And many more!
Concurrent Write Primitive

• Arbitrary concurrent writes
  - Winner thread doesn’t matter
  - Pick first Thread
• Requires 3 components
  - An auxiliary memory to indicate if a writer was picked
  - Track if a new round of CW was initiated
  - Atomic to force ordering – used sparingly

1. lastRoundUpdated : Last round the associated cell was updated
2. round: Current write round
3. atomic_cas(memory, old, new): Compare-exchange; return true if successful

4. bool canConWrite (&lastRoundUpdated, round) {
   5        x = false;
   6        if ((current = lastRoundUpdated) < round)
   7        x = atomic_cas (lastRoundUpdated, current, round)
   8        return x;
   9    }

10. While (!done) {
11       #pragma omp parallel for
12       For (unsigned i = 0; i < N; i++) {
13           If (canConWrite (&lastRoundUpdated[i], round) )
14           ConWriteTarget[i] = ....
15       }
16       round++;
17    }
Concurrent Write Primitive

- A thread checks if it can concurrently write
  - Force atomic execution – only one winner \( \rightarrow \) Write
  - The rest will lose \( \rightarrow \) Skip write
  - Threads attempting after a winner was picked will fail \( \rightarrow \) skip atomic and write
  - Increment round to initiate a new round of writes.

```
1  lastRoundUpdated: Last round the associated cell was updated
2  round: Current write round
3  atomic_cas(memory, old, new): Compare-exchange; return true if successful

4  bool canConWrite (&lastRoundUpdated, round) {
5    x = false;
6    if ((current = lastRoundUpdated) < round)
7      x = atomic_cas (lastRoundUpdated, current, round)
8    return x;
9  }

10  while (!done) {
11    #pragma omp parallel for
12    for (unsigned i = 0; i < N; i++) {
13      if (canConWrite (&lastRoundUpdated[i], round))
14        ConWriteTarget[i] = ...
15    }
16    round++;
17  }
```
Concurrent Write Primitive

- **Generic and flexible**
  - Independent of the algorithm being used

- **Guarantees Correctness**
  - avoids race conditions

- **Efficient**
  - Limit atomic execution

```
1 lastRoundUpdated : Last round the associated cell was updated
2 round: Current write round
3 atomic_cas(memory, old, new): Compare-exchange; return true if successful

4 inline bool canConWrite (unsigned &lastRoundUpdated, unsigned round) {
5    bool x = false;
6    if (unsigned current = lastRoundUpdated < round)
7       x = atomic_cas(lastRoundUpdated, current, round)
8    return x;
9 }

10 While (!done) {
11    #pragma omp parallel for
12    For (unsigned i = 0; i < N; i++) {
13       If (canConWrite (&lastRoundUpdated[i], round) )
14          ConWriteTarget[i] = ......
15    }
16    Round++;
17 }
```
Experimental environment

• Implemented OpenMP benchmarks for PRAM based algorithms.
  – Naïve implementation: relying on underlying HW. (Naïve)
  – Atomic implementation: All N writers perform Atomic operation to pick a winner (Prefixsum)
  – Using our method. (CAS-LT)

• Used Andes system at Oak Ridge Leadership Computing Facility (OLCF) to measure runtime performance
Experimental Results

• Linear-time Maximum: Extreme case of write concurrency – $O(N)$ concurrent writes per memory location

• Speed up: Geometric mean 1.98x over the naive approach

• Atomic is 1.7x slower than naïve - ~3.5x slower than our method

• Scales well with number of threads – Max speed up 1.8x at 32 threads
Experimental Results

- Breadth First Search: We use the Rodinia benchmark’s OpenMP implementation.
- Better performance across all problem sizes. Geometric mean of 2x when compared to Rodinia.
- Better scalability with number of threads reaching a speedup of 2.24x
Experimental Results

• Connected Components (Awerbuch-Shiloach algorithm)

• Naïve implementation failed and produced wrong results.

• Better performance across all problem sizes. Geometric mean speedup of 4x compared to prefixsum

• Better scalability with number of threads reaching a max speedup of 4x
Thank you!!