Parallel Algorithms and Programming

Parallel algorithms in shared memory

Thomas Ropars

Email: thomas.ropars@univ-grenoble-alpes.fr

Website: tropars.github.io

References

The content of this lecture is inspired by:

- *Parallel algorithms* (Chapter 1) by H. Casanova, Y. Robert, A. Legrand.
- *A survey of parallel algorithms for shared-memory machines* by R. Karp, V. Ramachandran.
- *Parallel Algorithms* by G. Blelloch and B. Maggs.
- *Data Parallel Thinking* by K. Fatahalian

Outline

- The PRAM model
- Some shared-memory algorithms
- Analysis of PRAM models

Need for a model

A parallel algorithm

- Defines multiple operations to be executed in each step
- Includes communication/coordination between the processing units

The problem

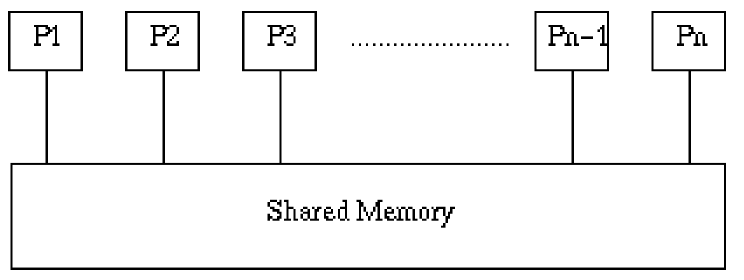
- A wide variety of parallel architectures
  - Different number of processing units
  - Multiple network topologies

- **How to reason about parallel algorithms?**
- **How to avoid designing algorithms that would work only for one architecture?**

- A model can be used to abstract away some of the complexity
  - Should still capture enough details to predict with a reasonable accuracy how the algorithm will perform

A model for shared memory computation

The PRAM model



- *Parallel RAM*
- A shared central memory
- A set of processing units (PUs)
    - Any PU can access any memory location in one unit of time
- The number of PUs and the size of the memory is unbounded

Details about the PRAM model

Lock-step execution

- A 3-phase cycle:
  1. Read memory cells
  2. Run local computations
  3. Write to the shared memory
- All PUs execute these steps synchronously
  - No need for explicit synchronization

About concurrent accesses to memory: 3 PRAM models

- **CREW**: Concurrent Read, Exclusive Write
- **CRCW**: Concurrent Read, Concurrent Write
  - Semantic of concurrent writes?
- **EREW**: Exclusive Read, Exclusive Write

About the CRCW model

Semantic of concurrent writes:

- *Arbitrary mode* : Select one value from the concurrent writes
- *Priority mode* : Select the value of the PU with the lowest index
- *Fusion mode* : A commutative and associative operation is applied to the values (logical OR, AND, sum, maximum, etc.)

How powerful are the different models:

$$CRCW > CREW > EREW$$

A model is more powerful if there is one problem for which this model allows implementing a strictly faster solution with the same number of PUs

Some shared-memory algorithms

List ranking

Description of the problem

- A linked list of n objects
    - Doubly-linked list
- We want to compute the distance of each element to the end of the list

The sequential solution

- Iterate through the list from the end to the beginning
- Assign each element a distance from the last element while iterating

This solution has a complexity (execution time) in $O(n)$

Can we do better with a parallel algorithm?

List ranking

A solution based on pointer jumping

```
# the list is stored in array *next*
# the distances are stored in array *d*
Ranking()
    forall i in parallel:           # initialization
        if next[i] is None:
            d[i] = 0
        else:
            d[i] = 1

    while there exists a node i such that next[i] != None:
        forall i in parallel do:
            if next[i] != None:
                d[i] = d[i] + d[next[i]]
                next[i] = next[next[i]] # pointer jumping
```

This solution has an execution time in $O(\log n)$

- Note that the solution requires `n` PUs
- We note that the parallel version requires more work than the sequential version of the algorithm

Credit: *Parallel algorithms*, Casanova, Robert, Legrand.

Comments on the previous algorithm

Implementing pointer jumping

```
forall i in parallel:
    next[i] = next[next[i]]
```

- In practice, if all processors do not execute synchronously, `next[next[i]]` may be overwritten by another PU before it is read here.
- To make the algorithm safe in practice, we would have to implement:

```
forall i in parallel:
    temp[i] = next[next[i]]
forall i in parallel:
    next[i] = temp[i]
```

Comments on the previous algorithm

About the termination test

- Note that the test in the while loop can be done in constant time only in the CRCW model
- The problem is about having all PUs sharing the result of their local test (`next[i] != None`)
- In a **CW** model, all PUs can write to the same variable and a fusion operation can be used
- In a **EW** model, the results of the tests can only aggregated two-by-two leading to a solution with a complexity in $O(\log n)$ for this operation

Point to root

Description of the problem

- A tree data structure
- Each node should get a pointer to the root

Use of pointer jumping

```
PointToRoot(P):
    for k in 1..ceiling(log(sizeof(P))):
        forall i in parallel:
            P[i] = P[P[i]]
```

- We assume that we know `sizeof(P)`

Divide and conquer

- Split the problems into sub-problems that can be solved independently
- Merge the solutions

Example: Mergesort

```
Mergesort(A):
    if sizeof(A) is 1:
        return A
    else:
        Do in parallel:
            L = Mergesort(A[0 .. sizeof(A)/2])
            R = Mergesort(A[sizeof(A)/2 .. sizeof(A)])
        Merge(L,R)
```

It is usually important to parallelize the divide and the merge step:

- In the algorithm above, the merge step is going to be the bottleneck

Analysis of PRAM models

Comparison of PRAM models

CRCW vs CREW

To compare CRCW and CREW, we consider a *reduce* operation over `n` elements with an associative operation.

- Example: the sum of `n` elements

    - With CRCW: $O(1)$ steps
    - With CREW: $O(\log n)$ steps

Comparison of PRAM models

CREW vs EREW

To compare CREW and EREW, we consider the problem of determining whether an element `e` belongs to a set $(e_1, \ldots e_n)$.

- Solution with CREW:
  - A boolean `res` is initialized to false and `n` PUs are used
  - PU `k` runs the test ($e_k == e$)
  - If one PU finds `e`, it sets `res` to true
- Solution with EREW:
  - Same algorithm except `e` cannot be read simultaneously by multiple PUs
  - `n` copies of `e` should be created (*broadcast*)

- With CREW: $O(1)$ steps
- With EREW: $O(\log n)$ steps

Limits of the PRAM model

- Unrealistic memory model
    - Constant time access for all memory location

- Synchronous execution
    - Removes some flexibility

- Unlimited amount of resources
    - Might not allow devising an algorithm that works well on a real system

Study of Parallel scans

Scans (Prefix sums)

Description of the problem

- Inputs:
  - A sequence of elements $x_1, x_2 \ldots x_n$
  - A associative operation *
- Output:
  - A sequence of elements $y_1, y_2 \ldots y_n$ such that $y_k = x_1 * x_2 \ldots * x_k$

Solution applying the pointer jumping technique

```
Scan(L):
    forall i in parallel:       # initialization
        y[i] = x[i]

    for k in 1..ceiling(log(sizeof(L))):
        forall i in parallel:
            if next[i] != None:
                y[next[i]] = y[i] * y[next[i]]
                next[i] = next[next[i]]
```

Scans (Prefix sums)

Performance of this algorithm

- Work:

$$W(n) = O(n) \times log(n)$$

- Depth:

$$D(n) = log(n)$$

If we do not have `n` processing units in practice, the large value of `n` can be an issue for performance

For instance, what would be a good algorithm on two processing units?

Parallel scan with 2 processing units

Solution

```
Scan(L):
    # input: x; output: y
    # first phase
    half = sizeof(L)/2
    for i in 0..1 in parallel
        SequentialScan(x[half*i .. half*(i+1)-1])

    # second phase
    base = y[half]
    quarter = half / 2
    for i in 0..1 in parallel
        add base to elems in y[half+quarter*i .. half+quarter*(i+1)-1]
```

Performance of this algorithm

- Work: W(n) = O(n)
- Depth: D(n) = O(n)
- It will perform better in practice due to the reduced amount of work
- Improves the locality of the data accesses (good for prefetchers)

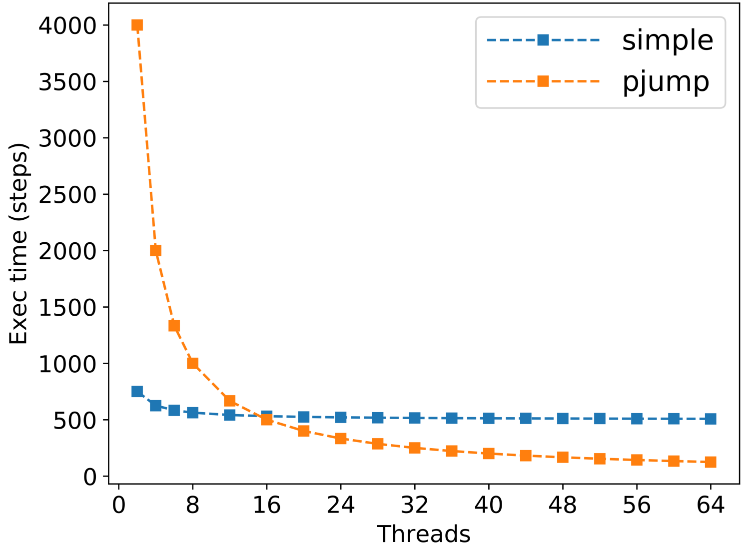
Credit: *Lecture -- Data parallel thinking*, Fatahalian.

Performance comparison

Assumptions for the computation

- Read 2 elements, compute the sum, and write back the result in 1 step
- Array of 1000 elements

Execution time as a function of the number of PUs



The algorithm with a larger depth and less work per iteration performs better up to 16 PUs