Notes on Parallel Algorithms

July 29, 2012

1 Introduction

1.1 The PRAM Model

- CRCW PRAM
  (a) **Common CRCW PRAM**: all processors writing into the same location must write the same value
  (b) **Arbitrary CRCW PRAM**: any one processor participating in a common write may succeed, and the algorithm should work correctly regardless of which one succeeds
  (c) **Priority CRCW PRAM**: there's a linear ordering on the processors, and the minimum numbered processor writes its value in a concurre2nt write
- *A priority CRCW PRAM algorithm can be simulated by an EREW PRAM with the same number of processors with the parallel time within a factor of $O(\lg P)$ where P is the number of processors.*
- *A priority CRCW PRAM algorithm can be simulated by a common CRCW PRAM with the same parallel time provided that the number of processors are sufficiently many.*

1.2 Optimality and Efficiency of Parallel Algorithms

- Let a problem whose fastest (known) sequential algorithm A_s runs in Γ time be given. And suppose that there's a parallel algorithm A_p which runs in T time with P processors[1].
- The **speedup** S of a parallel algorithm is defined as

$$S = \frac{\Gamma}{T}.$$

  - We want as much speedup as possible.
  - We say that a parallel algorithm achieves **linear speedup** when $S = P$ or $S = \Theta(P)$.
  - Linear speedup means that we've got a P-processor parallel algorithm that is P times as fast as the sequential one.
  - Note that $S \leq P$. Since, otherwise, we could devise a sequential algorithm that is faster than A_s: we can change (T,P)-algorithm into $(TP,1)$-algorithm by unfolding all the parallel steps; since $\Gamma \leq TP$, $S = \Gamma/T \leq P$.

[1] Note that Γ, T and P are all functions of the input size n. That is, these are abbreivations for $\Gamma(n), T(n)$ and $P(n)$.

- The **work** W of a parallel algorithm is defined as
$$W = TP.$$
  - The work measures the total processing effor needed for a parallel algorithm and it *accounts for inefficiencies caused by one or more processors being idle.*
  - Alternatively, the work is sometimes defined to be $N_1 + N_2 + \cdots + N_T$, where N_i is the number of processors that are actively used during the ith step.
- The **efficiency** E of a parallel algorithm is defined as
$$E = \frac{\Gamma}{W}.$$
  - The efficiency measure the efficiency with which the *processors are utilized*.
  - Alternatively, the efficiency is the ration of the speedup to the number of processors used:
$$E = \frac{\Gamma}{W} = \frac{\Gamma}{TP} = \frac{S}{P}.$$
  - Note that $E \leq 1$ since $S \leq P$.
  - We want E to be as close to 1 as possible.
- Define $polylog(n) = \bigcup_{k \leq 0} O(\lg^k n)$.
- A parallel algorithm is **optimal** if
  (a) $T = polylog(n)$ and
  (b) $W = PT = \Gamma$, i.e. $S = P$ or $E = 1$.
- A parallel algorithm is **efficient** if
  (a) $T = polylog(n)$ and
  (b) $W = \Gamma \cdot polylog(n)$.
- *A major goal in parallel algorithm design is to find optimal and efficient algorithms with T as small as possible*.

1.3 Brent's Scheduling Principle

- Consider a computation C that can be done in T steps with X_i primitive operations at the ith step.
- C can be implemented using a parallel algorithm that runs in T time with $M = \max_i\{X_i\}$ processors.
- If we have $P \leq M$ processors, we can still simulate the ith step of C, in time $\lceil X_i/P \rceil \leq X_i/P + 1$.
- (**Brent's Lemma**) Hence the total parallel time to simulate C with P processors is no more than $\lceil X/P \rceil + T$ where $X = \sum_i X_i$, since
$$\lceil X_1/P \rceil + \cdots \lceil X_T/P \rceil \leq X_1/P + \cdots + X_T/P + T = X/P + T.$$
- ***The Brent's lemma is applicable only if the processor allocation is not a problem; i.e., it is possible for each of P processors to determine, on-line, the steps steps it needs to simulate***.
- When the Brent's lemma is applicable, *a parallel algorithm requiring work W and time T can be simulated using P processors in $W/P + T$ time*.
  - In practice, $P << N$, W/P dominates the magnitude of $W/P + T$.

2 Basic PRAM Techniques

2.1 Prefix Sums

Prefix sums problem

- (**Prefix Sums Problem**) Let $\oplus$ be an associative operation over a domain D. Given an ordered set

$$A = [a_1, a_2, \cdots, a_n]$$

of n elements from D, the prefix sums problem is to compute the n prefix sums

$$S_i = a_1 \oplus a_2 \oplus \cdots \oplus a_i = \bigoplus_{k=1}^{i} a_k$$

for $1 \leq i \leq n$.

- There are two variants the problem w.r.t. the representation of A:
  (a) **Vector prefix sums**: A is represented as an **array**[2]; algorithms for this variant is called the **scan** operation.
    – The **prescan** operation takes A in a vector form and returns $[I, a_1, (a_1 \oplus a_2), \cdots, (a_1 \oplus \cdots \oplus a_{n-1})]$ where I is the identity.
  (b) Linked list prefix sums: this is more often than not treated as a variation of the **list ranking problem**.

- Applications of prefix sums:
  1. To compact a sparse array.
  2. To lexically compare strings of characters.
  3. To add multiprecision numbers
  4. To evaluate polynomials
  5. To solve recurrences.
  6. To implement radix sorts.
  7. To implement quicksort.
  8. To search for regular expressions.
  9. To dynamically allocate processors.
  10. To label components in 2-dimensional images.

- (**Segmented Prefix Sums Problem**) Given an array $A = [a_1, a_2, \cdots, a_n]$ of n elements from D with *flag* array $F = [f_1, f_2, \cdots, f_n]$, where $f_i = 0$ or 1, the segmented prefix sums problem is to compute the n segmented prefix sums

$$S_i = \begin{cases} a_1 & i = 1, \\ a_i & i > 1 \text{ and } f_i = 1, \\ S_{i-1} \oplus a_i & i > 1 \text{ and } f_i = 0. \end{cases}$$

for $1 \leq i \leq n$.

[2] Usually, prefix sums problems assumes an array as its input representation.

– The segmented prefix sums problem can be reduced to the prefix sums problem as follows:

$$S_i \begin{cases} a_1 & i = 1, \\ (S_{i-1} \times_s f_1) \oplus a_i & i > 0. \end{cases}$$

where

$$S \times_s f = \begin{cases} I_\oplus & f = 1, \\ S & f = 0. \end{cases}$$

Parallel prefix sums algorithm

- Assume $n = 2^k$ for some $k > 0$.

```
ParallelPrefixSums([a_1, ..., a_n])
 1  if n = 1 then
 2     S_1 ← a_1
 3  else
 4     for i ← 1, ..., n/2 pardo
 5        b_1 ← a_{2i-1} ⊕ a_{2i}
 6     (B_1, ..., B_{n/2}) ← ParallelPrefixSums([b_1, ..., b_{n/2}])
 7     for even i pardo
 8        S_i ← B_{i/2}
 9     for odd i pardo
10        S_i ← B_{(i-1)/2} ⊕ a_i
11     return (S_1, ..., S_n)
```

- ParallelPrefixSums runs in EREW PRAM since there are no memory conflicts.
- The parallel time satisfies the recurrence $T(n) = T(n/2) + O(1)$ with $T(1) = 1$. Thus,

$$T(n) = O(\lg n).$$

- The work satisfies $W(n) = W(n/2) + O(n)$ with $W(1) = 1$. Thus

$$W(n) = O(n).$$

- Using Brent's lemma, we can see that ParallelPrefixSums is an optimal EREW PRAM algorithm with $P = O(n/\lg n)$ processors.
  - Let $P \leq n/\lg n$ processors be given and let $Q = \lfloor n/P \rfloor$.
  - We assign the ith processor to elements $a_{(i-1)q+1}, a_{(i-1)q+2}, \cdots, a_{iq}$.

Cole & Vishkin's algorithm for prefix sums

- Cole and Vishkin [1] devised an optimal CRCW PRAM algorithm with $T = O(\lg n/\lg\lg n)$ with a restriction that a_i are $O(\lg n)$-bit numbers and $\oplus$ is an $O(1)$-time operation.

2.2 List Ranking

List ranking problem

- (**List Ranking Problem**) Given a linked list of n elements, compute the suffix sums of the last i elements of the list, $1 \leq i \leq n$.
- The list ranking problem can be thought of as a variant of prefix sums problem where $a_i = 1$, the associative operator $\oplus$ is $+$, the input is represented as a ***linked list***, and the sum is computed from the end.
  - That is, this problem is determining the *rank* of each elements, where the **rank** of an element is defined to be the number of elements preceding it in the linked list.

List ranking algorithm based on pointer jumping

- Assume that the linked list represented by a contents array $C[1..n]$ and a successor array $S[1..n]$, where $c(i)$ gives the value of the ith element and $s(i)$ gives the index of the successor of the ith element.

```
BasicListRanking(C,S)
1  repeat
2        for i ← 1,···,n pardo
3           c(i) ← c(i) ⊕ c(s(i));
4           s(i) ← s(s(i));
5    until ⌈lg n⌉ iterations
```

- The step 4 is called **pointer jumping** (or **pointer doubling** or **shortcutting**).
- After the execution of BasicListRanking, $c(i)$ is the *distance of the ith element from the end of the linked list*.
  - Note that we can compute the rank of the ith element, $R(i)$, by

$$R(i) = c(i) - n + 2.$$

- BasicListRanking runs in $T = O(\lg n)$ time using $P = O(n)$ processors. So, this algorithm is not work-optimal.
- *Though this problem seems very similar to the prefix sums problem, we cannot apply the same idea of prefix sums algorithm since a* ***given element has no way of knowing whether it is at an odd or even position on the list.***

Sketch of optimal list ranking algorithms

- We need not necessarily locate the elements at 'even' positions.
- *It suffices to construt a set S of no more than $k \cdot n$ elements in the list, with $k < 1$, such that the distance in the list between any two consecutive elements of S is small.*
  - That is, instead of choosing $n/2$ even-position elements, we choose $k \cdot n$ elements distributed uniformly in the list.
- List ranking algorithm

1. **List contraction**: create a contracted list composed of the elements in S, in which each element of S has as its successor the first element of S that follows it in the original list, and a value equal to its own value in the original list, plus the sum of the values of the elements that lie between it and this successor.
2. Recursively, solve the list ranking problem for the contracted list. The suffix sum for each element in the contracted list is the same as its suffix sum in the original list.
3. Extend this solution to all elements of the original list. The time to do this is proportional to the maximum distance between two elements of S in the original list, and the work is proportional to the length of the original list.

- Once a contracted list of length less than $n/\lg n$ is obtained, list contraction is no longer needed; instead the list ranking problem for the contracted list can be solved using BasicListRanking algorithm.
  - This requires $O(\lg n)$ time using $n/\lg n$ processors.
- **How can we construct S?**
  - We need to consider two points:
    1. How can we choose the set S?
    2. How can we compact the elements of S into consecutive locations, in preparation for the recursive solution of the list ranking problem on the compacted list?

Constructing S using random mate algorithm

- Each element chooses a gender, *male* or *female* with equal probability.
- *An element is in set S iff it is female or has a male predecessor.*
  - Then, with probability $1-o(1)$, $|S| \leq 15n/16$.
  - Each element in S can find its successor in S in constant time, since the distance to its successor is at most 2.
- With random mating, each list contraction tends to shrink the length of the list by a constant factor, and thus the number of contractions needed to pass from the original list of length n to a list of length less than $n/\lg n$ is $O(\lg\lg n)$.
- We can *compact S* into consecutive locations using *prefix sums* in $O(\lg n)$ time.

An optimal $O(\lg n)$-time deterministic list ranking algorithm

- A *symmetry breaking* technique known as **deterministic coin tossing** can be used for this purpose.
- Given an n-element list, a subset S of these elements is an **r-ruling set** if S contains no two adjacent elements of the list, and every element not in S is at a distance no more than $k \cdot r$ on the list from an element in S, where k is a suitable constant.

RulingSetAlgorithm(C,S,P)
1 **for** $i \leftarrow 1$ **to** n **pardo**
2 $\quad c(i) \leftarrow i$
3 **repeat**
4 $\quad$ **for** each i **pardo**
5 $\quad\quad$ `//` $c_q(i)$ `denotes the` q`th bit of` $c(i)$
6 $\quad\quad$ `//` $c_q(s(i))$ `denotes the` q`th bit of` $c(s(i))$
7 $\quad\quad$ `//` bin(q) `denotes the binary representation of` q
8 $\quad\quad$ $q \leftarrow$ rightmost position q s.t. $c_q(i) \neq c_q(s(i))$
9 $\quad\quad$ $b \leftarrow$ the qth bit of $c(i)$
10 $\quad\quad$ $c(i) \leftarrow b$ concatenated with the bin(q))
11 $\quad$ **until** k iterations
12 **for** each i **pardo**
13 $\quad$ **if** $c(p(i)) \leq c(i)$ and $c(s(i)) \leq c(i)$ **then**
14 $\quad\quad$ assign i to the ruling set

- With appropriate choice of k, we can devise an optimal $O(\lg n)$-time EREW PRAM algorithm for the list ranking problem.

2.3 Tree Contraction

- (*Expression Evaluation Problem*) Given a parenthesized arithmetic expression E (using $+$ and $\cdot$ operations) with values assigned to the variables, evaluate E and all subexpressions of E.
- Note that prefix sums problem is the expression evaluation problem on the parenthesized expression

$$(\cdots(x_1+x_2)+x_3\cdots)+x_n).$$

- *Tree contraction* is a method of evaluating expression trees efficiently in parallel.
- The **SHUNT** operation applied to a leaf in an n-left binary tree T results in a *contracted tree T' in which l and $p(l)$ are deleted, and the other child l' of $p(l)$ has the parent of $p(l)$ as its parent*, while leaving the relative ordering of the remaining leaves unchanged.

TreeContraction(T)
1 `//` T `is a rooted, ordered, binary` n`-leaf tree`
2 Label the leaves in order from left to right as $1,\cdots,n$
3 **for** $\lceil \lg n \rceil$ iterations **pardo**
4 $\quad$ apply SHUNT to all odd-numbered leaves that are left child
5 $\quad$ apply SHUNT to all odd-numbered leaves that are right child
6 $\quad$ shift the rightmost bit in the labels of all remaing leaves

- Step 2 of TreeContraction can be implemented using *Euler tour techinque*.
- The work done in **for**-loop is

$$O\left(\sum_{i=1}^{\lceil \lg n \rceil} \frac{n}{2^i}\right) = O(n).$$

References

[1] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proceedings of FOCS'86, 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[2] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, pages 869–932. MIT Press, Cambridge, MA, 1990.

[3] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kauffman, San Mateo, CA, 1992.

[4] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Mateo, CA, 1992.