CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2014

Title page

Students should be familiar with inductive proofs, recursion, data structures, and programming at the CS3114 level.

Parallel Algorithms

- **Running time**: $T(n, p)$ where n is the problem size, p is number of processors.
- **Speedup**: $S(p) = T(n, 1)/T(n, p)$.
  - A comparison of the time for a (good) sequential algorithm vs. the parallel algorithm in question.
- Problem: Best sequential algorithm might not be the same as the best algorithm for p processors, which might not be the best for ∞ processors.
- Efficiency: $E(n, p) = S(p)/p = T(n, 1)/(pT(n, p))$.
- Ratio of the time taken for 1 processor vs. the total time required for p processors.
  - Measure of how much the p processors are used (not wasted).
  - Optimal efficiency $= 1 =$ speedup by factor of p.

As opposed to $T(n)$ for sequential algorithms.

Question: What algorithms should be compared?

$pT(n, p)$ is total amount of "processor power" put into the problem.

If $E(n, p) > 1$ then the sequential form of the parallel algorithm would be faster than the sequential algorithm being compared against – very suspicious!

So there are differing goals possible: Absolute fastest speedup vs. efficiency.

Parallel Algorithm Design

Approach (1): Pick p and write best algorithm.

- Would need a new algorithm for every p!

Approach (2): Pick best algorithm for $p = \infty$, then convert to run on p processors.

Hopefully, if $T(n, p) = X$, then $T(n, p/k) \approx kX$ for $k > 1$.

Using one processor to **emulate** k processors is called the **parallelism folding principle**.

no notes

Parallel Algorithm Design (2)

Some algorithms are only good for a large number of processors.

$$\begin{aligned} T(n, 1) &= n \\ T(n, n) &= \log n \\ S(n) &= n/\log n \\ E(n, n) &= 1/\log n \end{aligned}$$

For $p = 256$, $n = 1024$.
$T(1024, 256) = 4 \log 1024 = 40$.
For $p = 16$, running time $= (1024/16) * \log 1024 = 640$.
Speedup < 2, efficiency $= 1024/(16 * 640) = 1/10$.

Good in terms of speedup.

$1024/256$, assuming one processor emulates 4 in 4 times the time.
$E(1024, 256) = 1024/(256 * 40) = 1/10$.

But note that efficiency goes down as the problem size grows.

Amdahl's Law

Think of an algorithm as having a **parallelizable** section and a **serial** section.

Example: 100 operations.

- 80 can be done in parallel, 20 must be done in sequence.

Then, the best speedup possible leaves the 20 in sequence, or a speedup of $100/20 = 5$.

Amdahl's law:

$$\begin{aligned}\text{Speedup} &= (\mathcal{S} + \mathcal{P})/(\mathcal{S} + \mathcal{P}/N) \\ &= 1/(\mathcal{S} + \mathcal{P}/N) \leq 1/\mathcal{S},\end{aligned}$$

for $\mathcal{S}$ = serial fraction, $\mathcal{P}$ = parallel fraction, $\mathcal{S} + \mathcal{P} = 1$.

See John L. Gustafson "Reevaluating Amdahl's Law," CACM 5/88 and follow-up technical correspondance in CACM 8/89.

Speedup is Serial / Parallel.
Draw graph, speed up is Y axis, Sequential is X axis. You will see a nonlinear curve going down.

Amdahl's Law Revisited

However, this version of Amdahl's law applies to a fixed problem size.

What happens as the problem size grows?
Hopefully, $\mathcal{S} = f(n)$ with $\mathcal{S}$ shrinking as n grows.

Instead of fixing problem size, fix execution time for increasing number N processors (and thus, increasing problem size).

$$\begin{aligned}\text{Scaled Speedup} &= (\mathcal{S} + \mathcal{P} \times N)/(\mathcal{S} + \mathcal{P}) \\ &= \mathcal{S} + \mathcal{P} \times N \\ &= \mathcal{S} + (1 - \mathcal{S}) \times N \\ &= N + (1 - N) \times \mathcal{S}.\end{aligned}$$

How long sequential process would take / How long for N processors.

Since $S + P = 1$ and $P = 1 - S$.

The point is that this equation drops off much less slowly in N: Graphing (sequential fraction for fixed N) vs. speedup, you get a line with slope $1 - N$.

All of this seems to assume the same algorithm for sequential and parallel. But that's OK – we want to see how much parallelism is possible for the parallel algorithm.

Models of Parallel Computation

Single Instruction Multiple Data (SIMD)

- All processors operate the same instruction in step.
- Example: Vector processor.

Pipelined Processing:

- Stream of data items, each pushed through the same sequence of several steps.

Multiple Instruction Multiple Data (MIMD)

- Processors are independent.

Vector: IBM 3090, Cray

Pipelined: Graphics coprocessor boards

MIMD: Modern clusters.

MIMD Communications (1)

Interconnection network:

- Each processor is connected to a limited number of neighbors.
- Can be modeled as (undirected) graph.
- Examples: Array, mesh, N-cube.
- It is possible for the cost of communications to dominate the algorithm (and in fact to limit parallelism).
- **Diameter**: Maximum over all pairwise distances between processors.
- Tradeoff between diameter and number of connections.

no notes

MIMD Communications (2)

Shared memory:

- Random access to global memory such that any processor can access any variable with unit cost.
- In practice, this limits number of processors.
- Exclusive Read/Exclusive Write (EREW).
- Concurrent Read/Exclusive Write (CREW).
- Concurrent Read/Concurrent Write (CRCW).

no notes

Addition

Problem: Find the sum of two n-bit binary numbers.

Sequential Algorithm:

- Start at the low end, add two bits.
- If necessary, carry bit is brought forward.
- Can't do ith step until $i-1$ is complete due to uncertainty of carry bit (?).

Induction: (Going from $n-1$ to n implies a sequential algorithm)

no notes

Parallel Addition

Divide and conquer to the rescue:

- Do the sum for top and bottom halves.
- What about the carry bit?

Strengthen induction hypothesis:

- Find the sum of the two numbers **with** or **without** the carry bit.

After solving for $n/2$, we have L, L_c, R, and R_c.

Can combine pieces in constant time.

Two possibilities: carry or not carry.

Also, for each a boolean indicating if it returns a carry.

If right has carry then
  Sum $= L_c|R$
Else
  Sum $= L|R$
If Sum has carry then
  Carry $=$ TRUE
For Sum$_c$

Do the same using R_c since it is computing value having received carry.

Parallel Addition (2)

The $n/2$-size problems are independent.
Given enough processors,

$$T(n,n) = T(n/2, n/2) + O(1) = O(\log n).$$

We need only the EREW memory model.

Not $2T(n/2, n/2)$ because done in parallel!

Maximum-finding Algorithm: EREW

"Tournament" algorithm:

- Compare pairs of numbers, the "winner" advances to the next level.
- Initially, have $n/2$ pairs, so need $n/2$ processors.
- Running time is $O(\log n)$.

That is faster than the sequential algorithm, but what about efficiency?

$$E(n, n/2) \approx 1/\log n.$$

Why is the efficiency so low?

Since $\frac{T(n,1)}{nT(n,n)} = \frac{n}{n \log n}$

Lots of idle processors after the first round.

More Efficient EREW Algorithm

Divide the input into $n/\log n$ groups each with $\log n$ items.

Assign a group to each of $n/\log n$ processors.

Each processor finds the maximum (sequentially) in $\log n$ steps.

Now we have $n/\log n$ "winners".

Finish tournament algorithm.
$T(n, n/\log n) = O(\log n)$.
$E(n, n/\log n) = O(1)$.

In $\log n$ time.

More Efficient EREW Algorithm (2)

But what could we do with more processors?
A parallel algorithm is **static** if the assignment of processors to actions is predefined.

- We know in advance, for each step i of the algorithm and for each processor p_j, the operation and operands p_j uses at step i.

This maximum-finding algorithm is static.

- All comparisons are pre-arranged.

Cannot improve time past $O(\log n)$.

Doesn't depend on a specific input value.

As an analogy to help understand the concept of static: Bubblesort and Mergesort are static in this way. We always know the positions to be compared next.
In contrast, Insertion Sort is not static.

Brent's Lemma

Lemma 12.1: If there exists an EREW static algorithm with $T(n, p) \in O(t)$, such that the total number of steps (over all processors) is s, then there exists an EREW static algorithm with $T(n, s/t) \in O(t)$.

Proof:

- Let $a_i, 1 \leq i \leq t$, be the total number of steps performed by all processors in step i of the algorithm.
- $\sum_{i=1}^{t} a_i = s$.
- If $a_i \leq s/t$, then there are enough processors to perform this step without change.
- Otherwise, replace step i with $\lceil a_i/(s/t) \rceil$ steps, where the s/t processors emulate the steps taken by the original p processors.

Note that we are using t as the actual number of steps, as well as the variable in the big-Oh analysis, which is a bit informal.

Brent's Lemma (2)

- The total number of steps is now

$$\sum_{i=1}^{t} \lceil a_i/(s/t) \rceil \le \sum_{i=1}^{t} (a_i t/s + 1) = t + (t/s)\sum_{i=1}^{t} a_i = 2t.$$

Thus, the running time is still $O(t)$.

Intuition: You have to split the s work steps across the t time steps somehow; things can't **always** be bad!

If s is sequential complexity, then the modified algorithm has $O(1)$ efficiency.

Maximum-finding: CRCW

- Allow concurrent writes to a variable only when each processor writes the same thing.
- Associate each element x_i with a variable v_i, initially "1".
- For each of $n(n-1)/2$ processors, processor p_{ij} compares elements i and j.
- First step: Each processor writes "0" to the v variable of the smaller element.
  - Now, only one v is "1".
- Second step: Look at all $v_i, 1 \le i \le n$.
  - The processor assigned to the max element writes that value to MAX.

Efficiency of this algorithm is **very** poor!

- "Divide and crush."

Need $O(n^2)$*processors*
Need only constant time.
Efficiency is $1/n$.

Maximum-finding: CRCW (2)

More efficient (but slower) algorithm:

- Given: n processors.
- Find maximum for each of $n/2$ pairs in constant time.
- Find max for $n/8$ groups of 4 elements (using 8 proc/group) each in constant time.
- Square the group size each time.
- Total time: $O(\log\log n)$.

n/2 processors
n processors, using previous "divide and crush" algorithm.

This leaves n/8 elements which can be broken into n/128 groups of 16 elements with 128 processors assigned to each group. And so on.

Efficiency is $1/\log\log n$.

Parallel Prefix

- Let $\cdot$ be any associative binary operation.
  - Ex: Addition, multiplication, minimum.
- Problem: Compute $x_1 \cdot x_2 \cdot \ldots \cdot x_k$ for all $k, 1 \le k \le n$.
- Define $\mathrm{PR}(i, j) = x_i \cdot x_{i+1} \cdot \ldots \cdot x_j$.
  We want to compute $\mathrm{PR}(1, k)$ for $1 \le k \le n$.
- Sequential alg: Compute each prefix in order
  - $O(n)$ time required (using previous prefix)
- Approach: Divide and Conquer
  - IH: We know how to solve for $n/2$ elements.
    1. $\mathrm{PR}(1, k)$ and $\mathrm{PR}(n/2+1, n/2+k)$ for $1 \le k \le n/2$.
    2. $\mathrm{PR}(1, m)$ for $n/2 < m \le n$ comes from $\mathrm{PR}(1, n/2) \cdot \mathrm{PR}(n/2+1, m)$ – from IH.

We don't just want the sum or min of all – we want all the partials as well.

We have the lower half done, and the upper half values are each missing the contribution from the lower half.

Parallel Prefix (2)

- **Complexity**: (2) requires $n/2$ processors and CREW for parallelism (all read middle position).
- $T(n, n) = O(\log n)$; $E(n, n) = O(1/\log n)$.
  Brent's lemma no help: $O(n \log n)$ total steps.

That is – no processors are "excessively" idle. This is because we needed to copy PR(1, $n/2$) into $n/2$ positions on the last step.

$$E = \frac{n}{n \cdot \log n} = \frac{1}{logn}$$

Better Parallel Prefix

- E is the set of all x_is with i even.
- If we know $\mathrm{PR}(1, 2i)$ for $1 \leq i \leq n/2$ then $\mathrm{PR}(1, 2i+1) = \mathrm{PR}(1, 2i) \cdot x_{2i+1}$.
- Algorithm:
  - Compute in parallel $x_{2i} = x_{2i-1} \cdot x_{2i}$ for $1 \leq i \leq n/2$.
  - Solve for E (by induction).
  - Compute in parallel $x_{2i+1} = x_{2i} \cdot x_{2i+1}$.
- Complexity:
  $T(n, n) = O(\log n)$.
  $S(n) = S(n/2) + n - 1$, so $S(n) = O(n)$ for $S(n)$ the total number of steps required to process n elements.
- So, by Brent's Lemma, we can use $O(n/\log n)$ processors for $O(1)$ efficiency.

Since the E's already include their left neighbors, all info is available to get the odds.

There is only one recursive call, instead of two in the previous algorithm.

Need EREW model for Brent's Lemma.

Routing on a Hypercube

Goal: Each processor P_i simultaneously sends a message to processor $P_{\sigma(i)}$ such that no processor is the destination for more than one message.

Problem:

- In an n-cube, each processor is connected to n other processors.
- At the same time, each processor can send (or receive) only one message per time step on a given connection.
- So, two messages cannot use the same edge at the same time – one must wait.

Need a figure

Randomizing Switching Algorithm

It can be shown that any deterministic algorithm is $\Omega(2^{n^a})$ for some $a > 0$, where 2^n is the number of messages.

A node i (and its corresponding message) has binary representation $i_1 i_2 \cdots i_n$.

Randomization approach:

(a) Route each message from i to j to a random processor r (by a randomly selected route).
(b) Continue the message from r to j by the shortest route.

n-dimensional hypercube has 2^n nodes.

Remember that we want parallel algorithms with cost $\log n$, not cost n^a!
The distance from any processor i to another processor j is only $\log n$ steps.

Randomized Switching (2)

Phase (a):

```
for (each message at i)
cobegin
  for (k = 1 to n)
    T[i, k] = RANDOM(0, 1);
  for (k = 1 to n)
    if (T[i, k] = 1)
      Transmit i along dimension k;
coend;
```


Randomized Switching (2)

no notes

Randomized Switching (3)

Phase (b):

```
for (each message i)
cobegin
  for (k = 1 to n)
    T[i, k] =
       Current[i, k] EXCLUSIVE_OR Dest[i, k];
  for (k = 1 to n)
    if (T[i, k] = 1)
      Transmit i along dimension k;
coend;
```


Randomized Switching (3)

no notes

Randomized Switching (4)

With high probability, each phase completes in $O(\log n)$ time.

- It is possible to get a really bad random routing, but this is unlikely (by chance).
- In contrast, it is very possible for any correlated group of messages to generate a bottleneck.

Randomized Switching (4)

no notes

Sorting on an array

Given: n processors labeled $P_1, P_2, \cdots, P_n$ with processor P_i initially holding input x_i.

P_i is connected to P_{i-1} and P_{i+1} (except for P_1 and P_n).

- Comparisons/exchanges possible only for adjacent elements.

```
Algorithm ArraySort(X, n) {
  do in parallel ceil(n/2) times {
    Exchange-compare(P[2i-1], P[2i]); // Odd
    Exchange-compare(P[2i], P[2i+1]); // Even
  }
}
```

A simple algorithm, but will it work?

Sorting on an array

Any algorithm that correctly sorts 1's and 0's by comparisons will also correctly sort arbitrary numbers.

Parallel Array Sort

7	3	6	5	**8**	1	4	2
3	7	5	6	1	**8**	2	4
3	5	7	1	6	2	**8**	4
3	5	1	7	2	6	4	**8**
3	1	5	2	7	4	6	**8**
1	3	2	5	4	7	6	**8**
1	2	3	4	5	6	7	**8**
1	2	3	4	5	6	7	**8**
1	2	3	4	5	6	7	**8**

Manber Figure 12.8.

Correctness of Odd-Even Transpose

Theorem 12.2: When Algorithm ArraySort terminates, the numbers are sorted.

Proof: By induction on n.

Base Case: 1 or 2 elements are sorted with one comparison/exchange.

Induction Step:

- Consider the maximum element, say x_m.
- Assume m odd (if even, it just won't exchange on first step).
- This element will move one step to the right each step until it reaches the rightmost position.

no notes

Correctness (2)

- The position of x_m follows a diagonal in the array of element positions at each step.
- Remove this diagonal, moving comparisons in the upper triangle one step closer.
- The first row is the nth step; the right column holds the greatest value; the rest is an $n-1$ element sort (by induction).

Map the execution of n to an execution of $n-1$ elements.

See Manber Figure 12.9.

Sorting Networks

When designing parallel algorithms, need to make the steps independent.

Ex: Mergesort split step can be done in parallel, but the join step is nearly serial.

- To parallelize mergesort, we must parallelize the merge.

no notes

Batcher's Algorithm

For n a power of 2, assume $a_1, a_2, \cdots, a_n$ and $b_1, b_2, \cdots, b_n$ are sorted sequences.

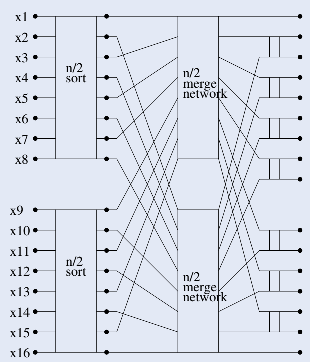
Let $x_1, x_2, \cdots, x_{2n}$ be the final merged order.

Need to merge disjoint parts of these sequences in parallel.

- Split a, b into odd- and even- index elements.
- Merge a_{odd} with b_{odd}, a_{even} with b_{even}, yielding $o_1, o_2, \cdots, o_n$ and $e_1, e_2, \cdots, e_n$ respectively.

No notes

Batcher's Sort Image



No notes

Batcher's Algorithm Correctness

Theorem 12.3: For all i such that $1 \leq i \leq n-1$, we have $x_{2i} = \min(o_{i+1}, e_i)$ and $x_{2i+1} = \max(o_{i+1}, e_i)$.

Proof:

- Since e_i is the ith element in the sorted even sequence, it is $\geq$ at least i even elements.
- For each even element, e_i is also $\geq$ an odd element.
- So, $e_i \geq 2i$ elements, or $e_i \geq x_{2i}$.
- In the same way, $o_{i+1} \geq i+1$ odd elements, $\geq$ at least $2i$ elements all together.
- So, $o_{i+1} \geq x_{2i}$.
- By the pigeonhole principle, e_i and o_{i+1} must be x_{2i} and x_{2i+1} (in either order).

See Manber Figure 12.11.

Batcher Sort Complexity

- Total number of comparisons for merge:
  $$T_M(2n) = 2T_M(n) + n - 1; \quad T_M(1) = 1.$$
  Total number of comparisons is $O(n \log n)$, but the depth of recursion (parallel steps) is $O(\log n)$.
- Total number of comparisons for the sort is:
  $$T_S(2n) = 2T_S(n) + O(n \log n), \quad T_S(2) = 1.$$
  So, $T_S(n) = O(n \log^2 n)$.
- The circuit requires n processors in each column, with depth $O(\log^2 n)$, for a total of $O(n \log^2 n)$ processors and $O(\log^2 n)$ time.
- The processors only need to do comparisons with two inputs and two outputs.

$O(\log n)$ sort steps, with each associated merge step counting $O(\log n)$.

Matrix-Vector Multiplication

Problem: Find the product $x = A\mathbf{b}$ of an m by n matrix A with a column vector $\mathbf{b}$ of size n.

Systolic solution:

- Use n processor elements arranged in an array, with processor P_i initially containing element b_i.
- Each processor takes a partial computation from its left neighbor and a new element of A from above, generating a partial computation for its right neighbor.

Cost: $O(n + m)$

Matrix-Vector Multiplication
Problem: Find the product $x = A\mathbf{b}$ of an m by n matrix A with a column vector $\mathbf{b}$ of size n.
Systolic solution:
Use n processor elements arranged in an array, with processor P_i initially containing element b_i.
Each processor takes a partial computation from its left neighbor and a new element of A from above, generating a partial computation for its right neighbor.
Cost: $O(n + m)$

See Manber Figure 12.17.