Notes

Monday, January 10, 2022 10:09 AM

- Moore's Law: Computing speed doubles every 18 months
  - For this to continue, we must use Parallel computing
  - Essentially, its about dividing the work among multiple CPU's
- No good programming languages for parallel
- Types of Parallel Computing:
  - Shared memory multiprocessors (thomas)
    - OpenMP / threads
  - GPU (NVIDIA)
    - CUDA: coding environment for GPU's
  - Distributed Parallel (cluster)
    - Big Dawg, MPI

Big-O Analysis

n = limit size

- Analyze Small primes Sieve
- 1st for loop → O(n)
- 2nd for loop → O(n)
- 3rd for loop → O(n)
- Big-OH → ignore lower order terms, = O(n)

```
for (d = 2, d ≤ n, d++)  ← O(√n)
    if (it[d])
        for (q = 2d, q ≤ n, q = q + d)  ← n/d
            it[d] = false;
```

For each prime ≤ √n

$$= \sum_{\text{primes } d \le \sqrt{n}} \frac{n}{d} = n\left[\sum_{\text{primes } d \le \sqrt{n}} \frac{1}{d}\right] \quad O(\log\log\sqrt{n})$$

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

To find primes ≤ n, it takes time $O(n \log\log n)$

If we use P processors how long should it take?

↳ $O\left(\frac{n \log\log n}{P}\right)$

How long does it take MPI to get our processors ready? P = processors

↳ MPI_INIT → O(log P)

→ MPI_Broadcast → O(log P)

→ MPI_Allreduce → O(2 · log P) or O(1.5 · log P)

→ MPI_finalize → O(log P)

function findSmallPrimes(sqrt n)

↳ $O(\sqrt{n} \log\log n)$

Blocks are size √n

There are $\frac{n}{\sqrt{n}} = \sqrt{n}$ blocks

Each processor does $\frac{\sqrt{n}}{P}$ blocks

Riemann Zeta Function

$$\zeta(s) = \frac{1}{1^s} + \frac{1}{2^s} + \frac{1}{3^s} + \dots = \frac{\pi^2}{6}$$

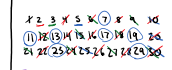
$$csize = \left\lceil \frac{limit}{nP} \right\rceil$$

Start = 1 + id · csize

Stop = Start + csize − 1

Finding Primes

Sieve of Eratosthenes



Target n

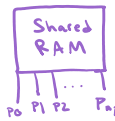
1. Find primes ≤ √n
2. Stripe segments of size near √n

A processor marks an array of bool of the segment size

Cross off multiples of the primes ≤ √n

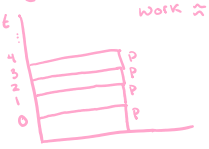
3. Share primes forward

PRAM Model:



The processors work in "Lockstep"

Time



work ≈ T × P

"Optimal" if work (T×P) matches the best Sequential time.

Goal of Designing Parallel Program:

work ≈ T × P

Maximum

given a list A[0]... A[n-1]

Find the max value.

↳ Like the sum problem, O(log n) time, O(n/log n) processors EREW PRAM

O(1) Time!! But how?

n processors for EACH A[i]

↳ initialize B[i] = 1

↳ Proc. i, j ≠ i

if A[i] > A[j] then B[j] = 0

if B[i] == 1

max = B[i]

NO LOOP

$O(n^2)$ work but O(1) run time

CRCW PRAM

Common

PRAM:

→ Adding up a list of numbers

O(log n) time

O(n/log n) processors

EREW PRAM

Work = time × procs = O(n)

goal is for work to match sequential

If we have $p < O(n/\log n)$ processors then time = work/p + parallel time

↳ $O(n/p + \log n)$

Maximum

-Crazy O(1) time $O(n^2)$ processors CRCW PRAM

```
for (int i to n in par; do)
    B[i] = 1
    for (int j = to n in par; do
    {
      if i=j
        if A[i] < A[j]
          B[i] = 0
for (i=1; to n in par do)
    if B[i] == 1
      output A
```

Parallel Random Access Machine:

PRAM Flavors:

- EREW → exclusive read exclusive write
- CREW → Concurrent read exclusive write
- CRCW → concurrent read and write
  - ↳ Common: must write same value
  - ↳ Arbitrary: can't predict which one wins
  - ↳ Priority: Small ID # wins

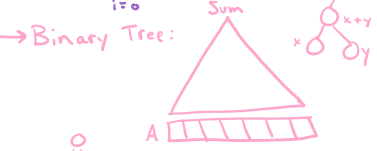
Hardware Power increases ↓

Summation Problem:

given a list A[0],...,A[n-1]

Compute $\sum_{i=0}^{i<n} A[i]$

↳ Binary Tree:

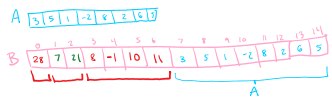


Parent(i) = $\lceil \frac{i}{2} \rceil - 1$

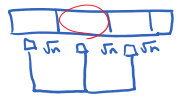
left(i) = 2i + 1

right(i) = 2i + 2

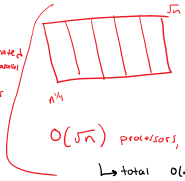
Time = O(log n)



Leveraging



apply crazy alg to these √n items O(n) procs, O(1) time



$O(\sqrt{n})$ processors, O(1) times

↳ total O(n) procs, O(1) time

n

√n pieces size √n

$n^{3/4}$ pieces size $n^{1/4}$ → How many pieces → log log n

n pieces size 1

Total: O(n) processors, O(log log n) time, CRCW PRAM

One More Tweak:

→ give each p chunks of size n/log log n to find maxima sequentially

↳ O(log log n) time, O(n/log log n) processors

↳ then, do the Asylum algorithm on input size n/log log n. Same time & procs.

Parallel Matrix Multiplication:

Recall:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -1+0+3 \\ -4+0+6 \\ -7+0+9 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}$$

How do we parallelize for n × n

→ Stripe the rows

→ PRAM → n^2 processors → O(log n) time

PRAM

Compute a sum of an array of length n

Matrix — Matrix Parallel

$[n \times n] \cdot [n \times n] = [n \times n]$

Normal Sequential Time: $O(n^3)$ → $n^2 \cdot n$

On the PRAM:

n^2 copies of dot-prod

$O(\log n)$ time

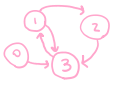
$O(n^3/\log n)$ procs

CREW PRAM

or $O(\log n)^2$ time EREW

time.

Graph Theory:



adjacency matrix

	0	1	2	3
0	0	0	0	1
1	0	0	1	1
2	0	0	0	1
3	0	1	0	0

$*$

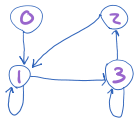
0	0	0	1
0	0	1	1
0	0	0	1
0	1	0	0

$=$

0	1	0	0
0	1	0	1
0	1	0	0
0	0	1	1

B

Reachability Graph distance = 2



0	1	0	0
0	1	0	1
0	1	0	0
0	0	1	1

For small example

$A^2 = R^{(2)}$

$B[i,j] = \bigvee_k A[i,k] \cdot A[k,j]$

entry (i,j) of A^2 is non-zero if there's a path of length $= 2$ from i to j

Let R^k be the reachability graph for paths of length $\leq k$

for a matrix M, let $b(M)$ be the matrix M with all non-zero entries set to 1

Then $R^k = b(A + A^2 + A^3 + \ldots + A^k)$

A graph G_1 is connected if its reachability graph R^{n-1} has 1's everywhere (except possibly diagonal)

Computing A^k

A^{12} $\quad 12 = 8 + 4$

Matrix exponentiation → binary algorithm to compute adjacency matricies

Fox's algorithm → multiplying matricies that are large

Matrix Exponentiation:

idea:

to compute A^n

we compute $A, A^2, A^4, \ldots A^{2^{\log n}}$

then we use the pieces we need based on writing n in binary

Example:

$A^{11} = A^8 \cdot A^2 \cdot A^1$

11 in binary is $8 + 2 + 1$

1011

$Y = I$ $\quad n = 1011$ $\quad B = A$

$Y = Y \cdot B$ $\quad n = n/2 = 101$ $\quad B = B^2 (= A^2)$ $\quad 101$

$Y = Y \cdot B$ $\quad n = n/2 = 10$ $\quad B = B^2 (= A^4)$

Y unchanged $\quad n = n/2$, $n = 1$ $\quad B = B^2 (= A^8)$

$Y = Y \cdot B$ $\quad n = n/2 = 0$ $\quad B = B^2 (= A^{16})$

Each processor does $\frac{\sqrt{n}}{p}$ blocks

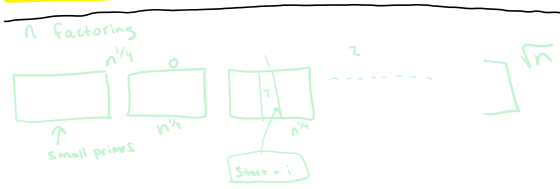
$\sum_{\text{Primes } d \le \sqrt{n}} \frac{\sqrt{n}}{d} = O(\sqrt{n} \log\log \sqrt{n})$

↳ Time for 1 block

↳ $O\left(\frac{n \log\log n}{p} + \log p + \sqrt{n} \log\log n\right)$

lower order terms

implies that $p \le \sqrt{n}$



n = 10000

n = 100

√100 = 10

for (int d = 2;

$O(\log n)$ running time → Work = time * processors

$O(n)$ processors

$O(\log n)$ time

$O(n/\log n)$ processors

$O(n)$ work

Aim: Work should match the Sequential run time

How do we do this?

1. each of the processors Sequential

→ PRAM → ... $O(\log n)$ time

PRAM

Compute a sum of an array of length n

↳ done in $O(\log n)$ time (tree structure)

↳ $O(n/\log n)$ processors

Work = (time · P) → $O(n)$

Dot Product

arrays a, b of length n;

Compute $\sum_{i=0}^{n-1} a[i] \cdot b[i]$

Same: $O(\log n)$ time

$O(n/\log n)$ processors

EREW

Matrix – Vector Product

$[n \times n] \; [n] = [n]$

↳ $O(\log n)$ time

↳ $O(n^2/\log n)$ processors

is possible! → EREW

Simple n-way copy of dot product recieves concurrent reads CREW

Before alg starts, copy the vector n times: 1 to put each vector entry into binary tree and copy down root to leaf in $O(\log n)$ time. n^2 processors

→ Then do log factor trick in Summation problem

Exponen

y = I

B = A

While

i

if

i

3

B

n

3

An

For

L

A

For

rewrite

$Y = Y \cdot B$ $\quad n = n/2 = 0$ $\quad B = B^2 (= A^{16})$

$Y = A^{11}$

ntiation :

(n > 0)

n is odd

$Y = Y \cdot B$

$= B^2$ $\quad = n/2$ $\quad$ if $(n > 0)$ $B = B^2$

is in Y

Example :

A^{20} $\quad n = 16 + 4 = 10100$

$Y = I$

$B = A$ $\quad n = 20$, even

$B = B^2$ $\quad n = n/2 = 10$

$B = B^2 = (A^4)$ $\quad n = n/2 = 5$

$Y = Y \cdot B$ $\quad (\text{so } Y = A^4)$

$B = B^2 (= A^8)$ $\quad n = 5/2 = 2$ $\quad$ even

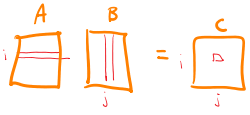
$B = B^2 (= A^{16})$ $\quad n = n/2 = 1$ $\quad$ odd

$Y = Y \cdot B$ $\quad (= A^4 \cdot A^{16})$

's Algorithm

When matricies are too big, we gonna use Fox!

$\cdot B = C$ $\quad$ assume they are $n \times n$



(int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

C[i, j] = 0;

```
for (int k=0; k<n; k++)
    C[i,j] += A[i,k] * B[k,j]
```

```
for (k = 0; k < n; k++)
{
    s = (i + k) mod n
    C[i,j] += A[i,s] * B[s,j]
}
```

Starting Somewhere else for dot product

Ex:

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$$

$P_{i,j}$ is the processor computing $C[i,j]$

→ Old version (Fox free) we would have:

P_{00} $A_{00} \cdot B_{00}$ $\quad P_{01}$ $A_{00} \cdot B_{01}$ $\quad P_{02}$

P_{10} $A_{10} \cdot B_{00}$ $\quad P_{11}$ $\quad P_{12}$

P_{20} $A_{20} \cdot B_{00}$ $\quad P_{21}$ $\quad P_{22}$

Fox Version:

P_{00} A_{00} B_{00} $\quad P_{01}$ $\quad P_{02}$

P_{10} A_{11} B_{10} $\quad P_{11}$ $\quad P_{12}$

P_{20} $A_{22} \cdot B_{20}$ $\quad P_{21}$ $\quad P_{22}$