a question

If a man can plough a field in 25 days, how long does it take for 5 men to plough the same field?

5 days

a similar question

If a processor can add two n-bit numbers in n microseconds, how long does it take for n processors to add together two n-bit numbers?

hmm…

Dot products

a = (4 5 -2 1)

\[ \mathbf{a} = (a_1, a_2, \ldots, a_n) \]

b = (1 -3 3 7)

\[ \mathbf{b} = (b_1, b_2, \ldots, b_n) \]

Dot product of a and b

\[ \mathbf{a} \cdot \mathbf{b} = 4.1 + 5.(-3) + (-2).3 + 1.7 = 10 \]

Also called “inner product”.

In general, \( \mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} (a_i b_i) \)
Dot products

If we can add/multiply two numbers in time \( C \), how long does it take to compute dot products for \( n \)-length vectors?

- \( n \) multiplications
- \( n-1 \) additions
  hence, \( C \cdot (2n-1) \) time.

Parallel dot products

What if \( n \) people decided to compute dot products and they worked in parallel?

All the pairwise products can be computed in parallel (1 unit of time)

How to add these \( n \) products up fast?

Can add these numbers up in \( \lceil \log_2 n \rceil \) rounds
Hence dot products take \( \lceil \log_2 n \rceil +1 \) time in parallel.

Not enough people?

What if there were fewer than \( n \) people?

\[ m < n \]

(a) What can you do? Exercise

(b) Can you do better than \( \lceil \log_2 n \rceil \) rounds
if \( m \) people are using in parallel?
Another example: Matrix-vector multiplications

Suppose we were given a m*n matrix A and a n-length vector x. How much time does it take to compute Ax?

\[
Ax = \begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} A_{11}x_1 + \cdots + A_{1n}x_n \\ \vdots \\ A_{m1}x_1 + \cdots + A_{mn}x_n \end{pmatrix}
\]

---

How much time in parallel?

Since just m dot product computations and all of them can be done in parallel.

\[
\text{dot products computed in \(O(m)\) time in parallel. (very n people)}
\]

So if we had m*n people, we could compute the solution to Ax in \(O(\log_2 n)\) time.

---

Back to our question…

If a single processor can add two n-bit numbers in n microseconds, how long does it take n processors to add together two n-bit numbers?

---

How to add 2 n-bit numbers.

\[
\begin{array}{cccccccccccc}
* & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * \\
\hline \\
* & * & * & * & * & * & * & * \\
\end{array}
\]

---

How to add 2 n-bit numbers.

\[
\begin{array}{cccccccccccc}
* & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * \\
\hline \\
* & * & * & * & * & * & * & * \\
\end{array}
\]

---

How to add 2 n-bit numbers.

\[
\begin{array}{cccccccccccc}
* & * & * & * & * & * & * & * \\
\hline \\
* & * \\
\end{array}
\]

---

How to add 2 n-bit numbers.

\[
\begin{array}{cccccccccccc}
* & * & * & * & * & * & * & * \\
\hline \\
* & * \\
\end{array}
\]
Let $k$ be the maximum time that it takes you to do the addition. Time is proportional to $n$.

If $n$ people agree to help you add two $n$ bit numbers, it is not obvious that they can finish faster than if you had done it yourself.

The time grows linearly with input size.
Is it possible to add two \( n \) bit numbers in less than linear parallel-time?

Darn those carries.

Plus/Minus Binary (Extended Binary)

Base 2: Each digit can be -1, 0, 1,

Example:

\[
(1 -1 -1) + (4 -2 -1) = (0, 0, 1)
\]

Not a unique representation system

Extended binary means base 2 allowing digits to be from \{-1, 0, 1\}. We can call each digit a “trit”.

Fast parallel addition is no obvious in usual binary.

But it is amazingly direct in Extended Binary!

\( n \) people can add two \( n \)-trit plus/minus binary numbers in constant time!

An Addition Party to Add 110-1 to -111-1
An Addition Party

Invite \( n \) people to add two \( n \)-trit numbers
Assign one person to each trit position

Each person should add the two input trits in their possession.
Problem: 2 and -2 are not allowed in the final answer.

Pass Left

If you have a 1 or a 2 subtract 2 from yourself and pass a 1 to the left. (Nobody keeps more than 0)
Add in anything that is given to you from the right. (Nobody has more than a 1)

After passing left

There will never again be any 2s as everyone had at most 0 and received at most 1 more

Passing left again

If you have a -1 or -2 add 2 to yourself and pass a -1 to the left (Nobody keeps less than 0)

After passing left again

No -2s anymore either. Everyone kept at least 0 and received at most -1.
Strategy

To add two n-bit binary numbers
Consider them to be in extended binary (EB)
no work required!

Sum them up to get an answer in EB.
constant parallel time!

Then convert them back to answer in binary
how do we do this fast in parallel???

Is there a fast parallel way to convert an Extended Binary number into a standard binary number?

Both problems not quite obvious:

Sub-linear time addition in standard Binary.

Sub-linear time EB to Binary

Let’s reexamine grade school addition from the view of a computer circuit.

Grade School Addition

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
+ & 1 & 0 & 1 \\
\hline
1 & 0 & 1 & 0
\end{array}
\]

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
+ & 1 & 0 & 1 \\
\hline
1 & 0 & 0 & 0
\end{array}
\]
Grade School Addition

\[
\begin{array}{cccccc}
C_5 & C_4 & C_3 & C_2 & C_1 \\
+ & a_4 & a_3 & a_2 & a_1 & a_0 \\
\hline
b_4 & b_3 & b_2 & b_1 & b_0
\end{array}
\]

Ripple-carry adder

Logical representation of binary: 0 = false, 1 = true

\[
s_1 = (a_i \text{XOR } b_i) \text{XOR } c_i \\
c_2 = (a_i \text{AND } b_i) \\
\quad \text{OR} (a_i \text{AND } c_i) \\
\quad \text{OR} (b_i \text{AND } c_i)
\]

Logical representation of binary: 0 = false, 1 = true
Ripple-carry adder

How long to add two n bit numbers?

Propagation time through the circuit will be $\Theta(n)$

Circuits compute things in parallel.

We can think of the propagation delay as **PARALLEL TIME**.

Is it possible to add two n bit numbers in less than linear parallel-time?

I suppose the EB addition algorithm could be helpful somehow.

Plus/minus binary means base 2 allowing digits to be from {-1, 0, 1}. We can call each digit a "trit".

$n$ people can add $2, n$-trit, plus/minus binary numbers in constant time!
Can we still do addition quickly in the standard representation?

Yes, but first a neat idea...

Instead of adding two numbers together to make one number, let’s think about adding 3 numbers to make 2 numbers.

 Carry-Save Addition
The sum of three numbers can be converted into the sum of 2 numbers in constant parallel time!

\[
\begin{align*}
1100111011 & \quad \text{Carries} \\
1011111101 & \quad \text{XOR} \\
1000000110 & \quad \text{Carries} \\
+ & \quad + \quad + \\
1111000000 & \quad 1011111101 & \quad 1000000110 \\
\hline
1111000000 & \quad 1011111101 & \quad 1000000110 \\
\end{align*}
\]

Cool!

So if we represent \( x \) as \( a+b \), and \( y \) as \( c+d \), then can add \( x, y \) using two of these (this is basically the same as that extended binary thing). 

\((a+b+c)+d=(e+f)+d=g+h\)
An aside:
Even in standard representation, this is really useful for multiplication.

Grade School Multiplication

\[
\begin{array}{c}
\times \\
10110111 \\
\hline
\end{array}
\]

\[
\begin{array}{cccccccc}
* & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * \\
\end{array}
\]

We need to add \( n \) 2n-bit numbers:
\[ a_1, a_2, a_3, \ldots, a_n \]

A tree of carry-save adders

T(n) \approx \log_{3/2}(n) + \text{[last step]}

So let's go back to the problem of adding two numbers.

In particular, if we can add two numbers in \( O(\log n) \) parallel time, then we can multiply in \( O(\log n) \) parallel time too!
If we knew the carries it would be very easy to do fast parallel addition

What do we know about the carry-out before we know the carry-in?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>C_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C_in</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>C_in</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Hey, this is just a function of a and b. We can do this in parallel.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>C_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>←</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>←</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

What do we know about the carry-out before we know the carry-in?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>C_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>←</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>←</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Idea #1: do this calculation first.

\[
\begin{array}{cccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
+ & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
\]

This takes just one step!
Idea #1: do this calculation first.

\[
\begin{align*}
  10 & \quad 1.0 \\
  + & \quad 1011111101 \\
& \quad 1000000110 \\
\end{align*}
\]

Also, once we actually have the carries, it will only takes one step more:

\[
s_i = (a_i \text{ XOR } b_i) \text{ XOR } c_i
\]

But we only have the carries in this peculiar notation!

How fast can we convert from this notation back to the standard notation?

\[
1011111100
\]

Called the “parallel prefix problem”

Idea #2: Can think of \(10\quad \leftarrow \quad 1.0\) as all partial results in:

\[
\begin{array}{c|c}
\text{for the operator } \odot: & 0 & 1 \\
\hline
\leaf x = x & 0 & 0 & 0 \\
1 \odot x = 1 & 1 & 1 & 1 \\
0 \odot x = 0 & 0 & 1 & \\
\end{array}
\]

And, the \(\odot\) operator is associative.

\[
\begin{align*}
(\leftarrow \odot (\leftarrow \odot (\leftarrow \odot (1 \odot (\leftarrow \odot 0))))))
& = \\
(\leftarrow \odot \leftarrow \odot (\leftarrow \odot 1) \odot (\leftarrow \odot 0)
& = \\
\leftarrow \odot 1 \odot 0
& = 1
\end{align*}
\]

Examples of binary associative operators

\begin{itemize}
\item Addition on the integers
\item Min(a,b)
\item Max(a,b)
\item Left(a,b) = a
\item Right(a,b) = b
\item Boolean AND
\item Boolean OR
\end{itemize}

Just using the fact that we have an Associative, Binary Operator

Binary Operator: an operation that takes two objects and returns a third.

\(A \odot B = C\)

Associative:

\[(A \odot B) \odot C = A \odot (B \odot C)\]
In what we are about to do “+” will mean an arbitrary binary associative operator.

Prefix Sum Problem

Input: \(X_{n-1}, X_{n-2}, \ldots, X_1, X_0\)
Output: \(Y_{n-1}, Y_{n-2}, \ldots, Y_1, Y_0\)
where
\[
Y_0 = X_0
\]
\[
Y_1 = X_0 + X_1
\]
\[
Y_2 = X_0 + X_1 + X_2
\]
\[
Y_3 = X_0 + X_1 + X_2 + X_3
\]
\[\vdots\]
\[
Y_{n-1} = X_0 + X_1 + X_2 + \ldots + X_{n-1}
\]

Prefix Sum example when + = addition

Input: 6, 9, 2, 3, 4, 7
Output: 31, 25, 16, 14, 11, 7
where
\[
Y_0 = X_0
\]
\[
Y_1 = X_0 + X_1
\]
\[
Y_2 = X_0 + X_1 + X_2
\]
\[
Y_3 = X_0 + X_1 + X_2 + X_3
\]
\[\vdots\]
\[
Y_{n-1} = X_0 + X_1 + X_2 + X_3 + \ldots + X_{n-1}
\]

Example circuitry (\(n = 4\))

Divide, conquer, and glue for computing \(y_{n-1}\)

Slightly more fancy construction coming up…
The above construction had small parallel run-time

But it used a lot of addition gates

\[ X_1\quad X_0 \quad y_1 \]

Let’s calculate how many we used...

Size of Circuit (number of gates)

\[ X_{n-1}\quad X_{n-2}\quad \ldots \quad X_{n/2}\quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \]

\[ X_{n/2+1}\quad \ldots \quad X_1\quad X_0 \quad y_{n-1} \]

Sum on \[ \lceil n/2 \rceil \] items

Sum on \[ \lfloor n/2 \rfloor \] items

\[ S(1)=0 \]
\[ S(n) = S(\lfloor n/2 \rfloor) + S(\lceil n/2 \rceil) + 1 \]
\[ S(n) = n-1 \]

Recursive Algorithm

n items (n = power of 2)

If \( n = 1 \), \( Y_0 = X_0 \);

\[ X_{n-1}\quad X_{n-2}\quad X_{n-3}\quad \ldots \quad X_2\quad X_1\quad X_0 \quad Y_{n-1}\quad \ldots \quad Y_0 \]

Prefix sum on \( n/2 \) items

Recursive Algorithm

n items (n = power of 2)

If \( n = 1 \), \( Y_0 = X_0 \);

\[ X_{n-1}\quad X_{n-2}\quad X_{n-3}\quad \ldots \quad X_2\quad X_1\quad X_0 \quad Y_{n-1}\quad \ldots \quad Y_0 \]

Prefix sum on \( n/2 \) items

Sum of Sizes

\[ S(n) = 0 + 1 + 2 + 3 + \ldots + (n-1) = n(n-1)/2 \]
Parallel time complexity

\[ T(1) = 0; \quad T(2) = 1; \quad T(n) = T(n/2) + 2 \]
\[ T(n) = 2 \log_2(n) - 1 \]

Prefix sum on \( n/2 \) items

End of fancier construction

Size

\[ S(1) = 0; \quad S(n) = S(n/2) + n - 1 \]
\[ S(n) = 2n - \log_2 n - 2 \]

Prefix sum on \( n/2 \) items

Putting it all together: Carry Look-Ahead Addition

To add two \( n \)-bit numbers: \( a \) and \( b \)

1 step to compute \( x \) values
2 \( \log_2 n - 1 \) steps to compute carries \( c \)
1 step to compute \( c \) XOR (\( a \) XOR \( b \))

2 \( \log_2 n + 1 \) parallel steps total

Putting it all together: multiplication

\[ T(n) = \log_2(n/2) + 2 \log_2 n + 1 \]
For a 64-bit word that works out to a parallel time of 22 for multiplication, and 13 for addition.

Addition requires linear time sequentially, but has a practical $2\log_2 n + 1$ parallel algorithm.

And this is how addition works on commercial chips.....

<table>
<thead>
<tr>
<th>Processor</th>
<th>n</th>
<th>$2\log_2 n + 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>80186</td>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>Pentium</td>
<td>32</td>
<td>11</td>
</tr>
<tr>
<td>Alpha</td>
<td>64</td>
<td>13</td>
</tr>
</tbody>
</table>

In order to handle integer addition/subtraction we use 2’s compliment representation, e.g.,

```
-44 =
```

```
  6 3 1 8 4 2 1
  1 0 1 0 1 0 0
```

Addition of two numbers works the same way (assuming no overflow).

```
-6 4 3 2 1 8 4 2 1
1 0 1 0 1 0 0
```
To negate a number, flip each of its bits and add 1.

\[
\begin{array}{cccccccc}
6 & 4 & 3 & 2 & 1 & 8 & 4 & 2 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

So, \(-x = \text{flip}(x) + 1\).

Most computers use two’s complement representation to perform integer addition and subtraction.

If millions of processors, how much of a speed-up might I get over a single processor?

Brent’s Law

At best, \(p\) processors will give you a factor of \(p\) speedup over the time it takes on a single processor.
If $n^2$ people agree to help you compute the GCD of two $n$ bit numbers, it is not obvious that they can finish faster than if you had done it yourself.

No one knows.

Parallel computation
- addition in extended binary
  - one-bit adder
  - ripple-carry adders
- computing carries using parallel prefix sum
- addition in parallel $O(\log n)$ time
- mult. in parallel $O(\log n)$ time
- one’s complement

Is GCD inherently sequential?