These notes are a distillation from a number of different parts of Yikie Quin's book.
Compared to the communication part, the computation is trivial.

Send (i.e., communicate) the appropriate sums to the appropriate store locations.

- appropriate sums in time no worse than \( u \).

Send (i.e., communicate) the appropriate products to the appropriate processors in order to compute the

- appropriate products in one time step.

Send (i.e., communicate) the appropriate values \( a \) and \( b \) to the appropriate processors.

Use \( u \) processors.

As a naive example, let us assume that we wish to compute the product of two \( u \times u \) matrices. The naive

common computational result.

As a matrix product is a triple loop: each of the \( u^3 \) entries in the product matrix is the sum of \( u \) products

that allow independent processors can perform independent computations in a way that contributes to a

In a very realistic sense, all of parallel computation is really a question of the

4.1 Introduction
hears any specific information.

- There are constraints on communication—if you all talk at once, then you can't guarantee that everyone

- There exists some global state (the classroom blackboard).

- Each of you has some local state (your own scratchpad).

- Each of you has some unique identifier (your processor ID).

**Assumptions**

How can this be done in parallel?

- Each of you wishes to compute a global value based on everyone's local values.

- Each of you has your own local item of information.

- Each of you is a processer.

Everyone is putting in.

Know the total amount available. In order to keep each other honest, you need to know the contribution that
investigate. You each have an amount of money to contribute to the fund. In order to invest, you need to
let us imagine the following experiment. You, the entire class, are going to contribute to a mutual fund for
Total cost is time u for n people to share this information.

- Time u for each of n people to total up their items into a global sum.
- Time u for each of n people to copy down their items from the board.
- Constant time for everyone to simultaneously write one item on the board.

TIME:

Everyone totals the information and produces the global sum.

Everyone writes the information on local scratch pads, so everyone has a pencil.

Everyone then writes down and copies all the information from the blackboard to their local scratch pads.

Everyone goes to the board at once and writes their local information in their own local block.

Everyone erases the local information from their local scratch pads.

The blackboard is blocked off into squares, one for each processor ID.

Method 4.2: Least constrained
Total time is still only n, no worse than before.

- One time step to add the new number to the running sum.
- One time step for each person to read the number off the board.
- Time n to write the numbers on the board.

**TIME**

- The person with the chalk hands off the piece of chalk to another person, and the process continues.
- Everyone else writes down that number and ID.
- The person with the chalk writes his/her number and ID on the board.

We can control this by having only one piece of chalk which must be shared.

If everyone writes on a randomly chosen place on the board, you have no audit trail of who had which local information.

What if you don’t have the board blocked off into individual squares?
Total time is now \( u \), which is very bad for having \( u \) processes.

- One time step to add the new number to the running sum.

- For each number written, time \( u \) to read the numbers.

- Time \( u \) to write the numbers on the board.

**TIME:**

continues.

The person with the chalk hands off the piece of chalk to a randomly chosen person, and the process

- The person with the chalk writes down the number and then passes on the pencil to a randomly chosen

- The person with the chalk writes his/her number and ID on the board.

One shared pencil.

One shared piece of chalk.
But if everyone writes and sums while holding the card, the total time is just the u steps.

With n cards, it takes time u to circulate the cards.

Beginning: Arrange the class in a linear order and pass the cards along wrapped around from the end to the front of the class.

What if the information is written on index cards? Each of you has one card and must pass them around.

What if there’s no blackboard? (You all have eyeglasses! You all have eyeglasses!)

Method Four: No Common Memory

4.5
For each of 2^n processors.

We will study the PFT Butterfly communication pattern that has the effect of a simultaneous binary tree.

They each send to two more processors (cost = 2 and 2 processors now informed).

They each send to two more processors (cost = 2 and 4 processors now informed).

You send the information to two processors (cost = 2 and 2 processors informed).

We can send information to W processors in only \( \log W \) message stages by using a binary tree.

Almost everything good in algorithms comes from binary decompositions.

4.6 Method Five and Beyond—There Are Better Ways
All of these represent constraints on the nature of the parallel computer for which you are designing algorithms.

What if you read ahead to receive a card, but the card is coming instead from the person behind you?

What if you must hold a card in one hand to pass it to someone else and you must also hold out a hand with another card in order to receive one?

What if you must hold a card in one hand to pass it to someone else and you must also hold out a hand with another card in order to receive one?

What if you are glued to your chair and can only communicate to your nearest neighbors in the 2-dimensional grid?
An algorithm is said to be scalable if the running time of the algorithm decreases roughly linearly with the number of processors.

**Definition 4.1.** An algorithm is said to be scalable if the running time of the algorithm decreases roughly linearly with the number of processors.

We dearly want algorithms to be scalable. This is a valuable design term, as are many terms. Basically, we assume parallel computer algorithms, we have no choice but to make some assumptions up front, and those with parallel computer algorithms we have no choice but to make some assumptions up front, and these we normally do the opposite and then pay lip service to issues like memory, etc.

We count the number of operations executed by the processor.

With ordinary computers and serial (sequential) algorithms, we have one processor and one memory.

4.8 Basic Concepts—Characteristics of Actual Computers
Programs have access to all of the machine's memory.

This is what we would like to have. But we can't have it because it is too expensive to interconnect processors.

Many processors, large box memory, all processors have genuine access to all memory locations with the same access time.

(e.g., Cray vector machines, Sun Enterprise, with caches)

4.8.1 Symmetric Multi Processors (SMPs)

Communication time with compute time and with memory access time. And this is not easy.

And in the real world, the analysis of parallel algorithms can be difficult because we have to balance getting useful work done.

In the real world, almost nothing is really scalable. Usually, good algorithms will scale to a certain point and then fail to scale beyond that point when overheads override the part of the computation that is in fact
Communications from one processor to another must be explicit. With other machines (CM-2, CM-5), processors are really limited to using only their local memory, and with some machines (T3E), processors' memories can be aggregated into a larger shared memory space. CM-2 and CM-5 had a logarithmic network. The processors are connected to one another with a fixed interconnect.

Many parallel compute nodes each with its own memory. (e.g., CM-2, T3E, Thinking Machines CM-2 and CM-5, Mach) Computing surface

4.8.2 Parallel Computers With an Interconnect Structure
Explicit message passing from processor to processor.

Computations must be primarily local.

Very loose coupling of processors.

SNOW is just "the usual" network linking ordinary workstations.

D ewulf is a dedicated network.

Individual computers on a network.

(e.g., Dewulf, Scalable Network of Workstations (SNOW))

4.8.4 Distributed Computers With Poor Interconnect

Space, and programs definitely work better if computations are local.

Problems see a shared memory space whose implementation may or may not act like a shared memory.

Memory access on the node is faster than access off node through the network.

Nodes of SMP computers connected to each other

(e.g., Sun Enterprise, SCI Origin, Pittsburgh Terascale)

Cluster Computers with Non-Uniform Memory Access (NUMA)
There is no difference except for the multiplicity of processors in a PRAM.

You should contrast this definition with the definition of a RAM in Chapter 1 of the text. In essence, there

processors have access to the entire global memory.

But the control unit. Processors may be idle in a given time step rather than execute the operation. We assume that all processors execute the same operation with each time step, under the control of the

an interconnect between the processors and the global memory

a global memory (assumed to be relatively large compared to local memory)

an unbounded number of processors each with its own (assumed relatively small) local memory

a control unit

Definition 4.2. A Parallel Random Access Machine, or PRAM, comprises

Theoretical Machines—The Assumptions of Parallel Processing

4.9
Algorithm analysis is therefore only a guide to what ought to be done in real life, not a prescription for what must be done.

Algorithm analysis on the more realistic machines on the RAM or PRAM becomes a very complicated multi-disciplinary optimization problem. Second, you probably can’t do the analysis on the more complicated machine because the single-variable needs on this idealized machine then you probably can’t do the analysis on the more complicated machine. If you can’t analyze an algorithm on this idealized machine then it’s probably for two reasons. First, you can’t analyze an algorithm that actually behaved like this was the VAX. For the most part, however, algorithm analysis must be done on idealized machines like this, probably for two reasons. Second, you probably can’t do the analysis on the more realistic machines on the RAM or PRAM becomes a very complicated multi-disciplinary optimization problem.

Perhaps the last machine that actually behaved like this was the VAX. For the most part, however, algorithm analysis must be done on idealized machines like this, probably for two reasons. First, you can’t analyze an algorithm on this idealized machine then it’s probably for two reasons. Second, you probably can’t do the analysis on the more realistic machines on the RAM or PRAM becomes a very complicated multi-disciplinary optimization problem.

Only one word is desired, etc.

etc. Memory access is taken to be of a single data item at a time—no cache line fetches of doublewords when

etc. Memory is taken to be a single, large, flat, memory—no cache, no memory banks, no virtual memory, no overlapped execution, assumptions to be executing a single instruction, start to finish, at a time—no pipelining, no overlapping execution, time and computer organization questions that are always taken into account in real machines. Processes are

CAVEATS: The definition of a PRAM (and of a RAM) ignores most of the lower-level computer architecture.
only increases the running time by a factor of two, which will not affect the asymptotic running time.

Note that if we have to split all the steps into two in order to accommodate read-and-write steps, this
or that all steps take effect before any writes, and thus that in a given instance there is no synchronization problem,
the withdrawal is made. In the case of reads and writes, we assume that all writes take effect before any reads,
for a given day and then process all withdrawals, so that deposits to cover withdrawals will take effect before
reading. This is handled in the same way that banks handle credits and debits. Banks make all deposits first
We need to exercise some care if one processor wants to write in the same timestep that another processor is
assumed to be writing the same value, in which case we don’t care about the order.

If the parallel writing of data that causes arithmetic problems. If two processors are writing to the
processors does not depend on the order in which they read.
requests to be sequenced, but since the same value is to be read by all processors, the value returned to all
parallel reading of data by multiple processors. There are hardware restrictions that prohibit multiple
same memory locations in the same time steps. In general, there should be no algorithmic restriction on the
The critical question with a PRAM is what happens when parallel processors choose to read or write the
- **Priority write**: Only the processor with the highest priority (in some scheme of static priorities) within the value (or the processor that happens to write last) is chosen arbitrarily.

- **Arbitrary write**: All processors can write to the same location. The processor that succeeds in.

- **Common write**: So long as all processors are writing the same value to the same location, the write.

• **Concurrent Read Concurrent Write (CRCW)**: This is the usual PRAM model.

  - In a given time step, but any number of processors may read the same location in the same time step.

• **Concurrent Read Exclusive Write (CREW)**: Only one processor may write to a given location in a given time step.

• **Exclusive Read Exclusive Write (EREW)**: Only one processor may read or write a given location.

In decreasing order of restriction and increasing order of power, we have the following:
• A common algorithm works on an arbitrary machine with no slowdown.

• An arbitrary algorithm works on a priority machine with no slowdown.

• An EREW algorithm works on a CREW machine with no slowdown.

• An EREW algorithm works on a CREW machine with no slowdown.
the algorithm but not necessarily lead to an incorrect final answer.

If the algorithm generates the correct answer, for example, for a Common White medicine:

Note the almost circular definition of the Concurrent White medicine.
If \( \gamma \neq \gamma' \) then \( P \) writes a 1 into \( S_{\gamma}^{\gamma'} \) else write a 0.

Processors \( P \) through \( P \) read locations \( L_{\gamma} \) and \( L_{\gamma'} \).

Now processor \( P \) reads \( (i, j) \) from location \( L_{\gamma} \) and writes a 1 in location \( S_{\gamma} \).

Constant time to find the highest priority processor for any particular location.

The EREW PRAM then sorts the extra location values in time \( \log p \).

When \( P \) in PPRAW accesses location \( W_{\gamma} \), the EREW PRAM writes \( W_{\gamma} \) in extra location \( L_{\gamma} \).

Algorithm.

The EREW PRAM uses extra global locations \( L_{\gamma} \), \( L_{\gamma'} \), and \( S_{\gamma} \) to simulate each step of the priority

Proof. The priority machine (PPRAW) uses processors \( P \), \( \ldots, P \), memory \( W_{\gamma} \), \( \ldots, W_{\gamma'} \) and \( S_{\gamma} \) to simulate the priority algorithm with no more than a \( \Theta(\log p) \) slowdown.

Theorem 4.4. A \( p \)-processor EREW PRAM can simulate a \( p \)-processor priority algorithm with no time.

Lemma 4.3. A \( p \)-processor EREW PRAM can sort a \( p \)-element array in global memory in \( \Theta(\log p) \) time.
Processors in \( O(d \log t) \) time:

- For a read, the highest priority processor gets to read and then can communicate the value to the other.

- For a write operation, the highest priority processor does the write, then the other.

Now the elements of \( S \) with value 1 correspond to the highest priority processors accessing each memory location.
a special memory location. In the next time step, all other processes can read from that location.

**Proof.** In a CREW PRAM, processor 0 can broadcast to all other processors by writing in one time step to

**Proposition 4.7.** In a CREW PRAM, broadcast takes constant time.

**Definition 4.6.** The total exchange problem is the problem on a parallel computer of having every processor exchange a value with every other processor.

**Definition 4.5.** The broadcast problem is the problem on a parallel computer of having one processor broadcast a value to all other processors.

This works fine on PRAM machines, but not so well on machines with fixed communication structures.

and so forth.

time using a binary tree. A control processor can spawn two processes, each of those can spawn two more.

We first can observe that with any of the PRAM models, we can spawn **d** processes on processors in **d** steps. Parallel Communications and Process Spawning
Proposition 4.9. In any architecture in which information is communicated one unit at a time, total exchange takes at least time linear in the number of processors.

Proof. Each processor now reads the upper values in the next $n - 1$ time steps.

Proposition 4.8. In a CREW PRAM, total exchange takes time linear in the number of processors.

Proof. Similar to broadcast, all processors can write their information to their specific location in time step $n - 1$.
After 16 steps, every processor has received information from every other processor. Information in time steps goes down.

Proof: Consider the following communication pattern. We have eight processors running across, exchanging information in the number of processors.

In an architecture or with a problem in which information can be aggregated for multiple processors before being communicated, total exchange can be done at least as fast as time.
might not.

As always with parallel algorithms, sometimes we might care (about cost-optimality), and sometimes we

times & comparisons is asymptotically smaller than the parallel n & in cost.

For example, parallel merge as discussed below is not cost optimal since the sequential time (t processor

of the number of processors and the time complexity of the computation.

Definition 4.12. A parallel computation (algorithm) is said to be cost optimal if the cost is in the

the same time complexity as the optimal sequential algorithm. of a (PRAM) computation is the product

Definition 4.11. The cost of a (PRAM) computation is the product of the number of processors and

However, in a two-dimensional mesh/8x8 connected machine, things are much different. Think about this,

an objective function.

Another situation in which this communication pattern could be useful would be a tree search to optimize

with the Fast Fourier Transform (FFT).

The communication pattern described above is called a butterfly and is associated, among other things,
Either way, we can never really know what the rules are for communications among processors. Consider:

- If we are not lucky, then we need to look at queueing theory or at parameter measurement (e.g., power)

- If we are lucky, then there is a regular structure that will help us with the information flow (e.g., Butterfly)

The problem in general is one of having local information on separate processors and needing global information distributed to all processors or else concentrated in one processor.

We have had for 25 years the problem of getting „bitridden through the bandwidth“ (the network connection in a SNOW).

Memory is slow and becoming slower relative to processors.

Communications are either fast and expensive to build (the switch in an SMP) or inexpensive but slow.

4.11 Complications and General Issues in Parallelism
E.8. How do we do a binary tree on a mesh?

What's the slowdown in going from one architecture to another?

Are we going in lockstep, or are we running separate processes at their own pace?

- What does the average transfer rate unloading and loading?
- What does MPI (or PWI) do to reduce hot spots?
- What is the actual size of the memory block we can transfer?

Or, for example, on a bowenii cluster or SNOW, we can ask

- Can we read and write simultaneously on a link?
- Can we read from all four links at once?
- Can we write to all four links at once?

For example, on a 2-dimensional mesh, we can ask
One such implementation is as follows, of four processors, to which we input the string 8, 5, 3, 7. We will

will those separate operations take separate time steps?

A precise implementation depends on the communication rules and operating procedures. (For example,

new value with the current value, and pump the larger of the two to the processor to the right.

each processor receive an item from the processor to the left and place it in the new register, compare the
for the data item received from the last time step, and a comparator. Processing will take place by having

Each basic processor will consist of a new register for a data item input to the processor, a current register

The processors will be arranged in a one-dimensional linear array. Data will be pumped in at one end (the

in time 2n.

We shall use this concept to construct a parallel "machine" with n processors that will sort n data elements

the way that blood is pumped through the vascular system.

We shall loosely refer to an algorithm as systolic if it operates by pumping the data past processors in much

4.12 A Systolic Sorting Algorithm and Machine
Some things to notice.

At this point we start pumping the sorted array out the end of the array.

<table>
<thead>
<tr>
<th>p_4</th>
<th>q_4</th>
<th>N_4</th>
<th>p_3</th>
<th>q_3</th>
<th>N_3</th>
<th>p_2</th>
<th>q_2</th>
<th>N_2</th>
<th>p_1</th>
<th>q_1</th>
<th>N_1</th>
<th>Tick</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

dummy values so that they can be distinguished for this example.
Finally, we note that with another bit for indicating breakpoint initialization, we can follow the 8 with the bit, we will take the 8 seriously as genuine data. The bit will indicate that the data is in fact valid data, so when the 8 appears alone with the validity data, then we have no guarantee that "∞ isn't a legitimate value, it is necessary to pass along a bit with the
And of course, "and" can be extended to any associative operation:

$$\begin{align*}
\neg \neg b & \equiv b \\
\neg \neg \neg p & \equiv p \\
\neg \neg \neg q & \equiv q \\
\neg \neg \neg \neg a & \equiv a \\
\neg \neg (b \land c) & \equiv (b \land c) \\
\neg \neg (p \land q) & \equiv (p \land q) \\
\neg \neg (c \land q) & \equiv (c \land q) \\
\neg \neg (a \land c \land p \land c \land q \land a) & \equiv (a \land c \land p \land c \land q \land a)
\end{align*}$$

Clearly, we can apply the basic binary tree structure to anything like summation on a parallel machine. For example, we can add $n$ items in $\log n$ time in parallel, in the obvious way.
location.

Proper location in the merged array. This can be done in parallel because each element goes into a different

This involves concurrent reading, but no writing at all. We now write in parallel, each element into its

Hence A and B in \( T(n^2) \) steps.

assigned to list \( A \), we can find in parallel the correct position in the sorted list of every element in each of

Using the \( n \) processors assigned to list \( A \) in this fashion, and using analogously the \( n/2 \) processors

Q1 through Q6. (Note that duplicates don't matter in this argument.)

the merged list, because it should be proceeded by the \( \lfloor \frac{n}{2} \rfloor \) elements through \( q \) and by the \( \lceil \frac{n}{2} \rceil \) elements of list \( B \).

If we have indexed both lists from 1 through \( \lfloor n/2 \rfloor \), this means that \( q \) should be placed in location \( \lfloor n/2 \rfloor \).

For two elements \( q \) and \( q+1 \) in list \( B \),

Processor \( P \). Using this processor, we can determine by binary search within \( \lfloor n/2 \rfloor \) comparisons that

Proof. Consider two sorted lists \( A \) and \( B \), each of length \( n/2 \). To each element \( q \) in list \( A \) we assign a

\( \Theta(n^2) \) elements in time \( \Theta(n^2) \).

Theorem 4.13. On a CREW PRAM we can with \( n \) processors perform a merge of two sorted lists of

4.14 Parallel Merge
is not cost optimal. We do them on \(n\) processors and hence \(n\) comparisons at a time, resulting in the \(n\) execution time. So this note that we do \(n\) comparisons in the parallel algorithm instead of the \(n\) of the serial algorithm. But

\[ i + j + k \]

but no value to \(i\), \(j\), or \(k\).

the risk of matching \(i, j, i\) with \(i, j, i\) and \(i, j, i\) with \(i, j, i\), say, and writing the same value to location \(i + j + k\). We would specify a consistent convention for breaking ties. Otherwise, in the case of a tie, we would have to judge ever so slightly on the question of duplicate values. To make this truly rigorous we must...
For example, we might have
where we can interpret \( \oplus \) to be any associative operation,

\[
\begin{align*}
    u_0 \oplus \cdots \oplus u_k \oplus a_1 \\
    \cdots \\
    \bar{u}_3 \oplus \bar{a}_1 \\
    \bar{a}_1 \\
\end{align*}
\]

are the sums

\[
\begin{align*}
    u_0 \oplus \cdots \oplus \bar{a}_1 \\
\end{align*}
\]

The prefix sums of an array
Theorem 4.14. The prefix sums of an array of $n$ elements can be computed on a CREW PRAM in $O(n)$ time. Where in each of the intermediate steps we only do the addition if in fact the $i - j$ subscripts are positive.

$$
\begin{array}{cccccccc}
  & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 0 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 8 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 2 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 4 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 6 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 9 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 12 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 15 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 18 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 21 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 24 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 27 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 30 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 33 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
 36 & 36 & 28 & 21 & 15 & 10 & 6 & 3 & 1 \\
\end{array}
$$

We can compute all the prefix sums in parallel in the following way:

Prefix sums

$\text{array}$

$\text{$i\sigma + 1$}$
We assign a different processor to each element in the array, do all the comparisons against the pivot in parallel, and set a bit in a mask array for those elements less than or equal to the pivot.

Assume that the pivot element is 4, and that we have the array make use of parallel prefix in the following way.

If, for example, we are doing the "split" part of the quicksort, moving all the values less than the pivot to the first part of the array and all the values greater than the pivot to the last part of an array, then we can look at exactly the opposite way. Parallel prefix is a fast operation. Let’s look for ways in which it could be incorporated into some other algorithms.

Now, why on earth would we want to do this? What does this rather odd looking prefix sum function do?

**Proof:** This should be obvious.

[The n steps]
So yes, prefix sums seem a little odd, but they can be useful operations. To move \( n \) elements, Here, we use \( 2^k \) to create the two masks and do the two parallel prefix comparison. Compare this to the data movement part of the quicksort on a serial machine. There, we use \( n \) comparisons creating the mask array.

We create the "last part" of the array with an analogous operation, different only in that we flip 0 and 1 in the "first part" array.

Knowing that it should write its value into location \( i \) in the "first part" array:

Now, if processor \( P \) reads locations \( i \) and \( i - 1 \) and finds different values in the prefix sums \( s_i \) and \( s_{i-1} \), it
```latex
next[i] = next[next[i]]

\text{suffix sum} \text{ sum} \text{ value}[i]
```

<table>
<thead>
<tr>
<th>Points to</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

What we will do is iterate making the values to be summed all 1 except for the end of the list, to which we'll assign 0.

Now consider the suffix sum problem in a linked list. Actually, we'll make the problem slightly easier by...
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Sub</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**Iteration 0:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Val</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sub</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**Iteration 1:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Val</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Sub</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**Iteration 2:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Val</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Sub</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**Iteration 3:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Val</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Sub</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Takes \( \text{\textit{O(n)}} \) operations in \( \text{\textit{Le}} \) \( \text{\textit{n}} \) time.

the suffix sum is the sequence position of that element measured from the end of the list.

When we're done, all elements point to the end of the list, and the value held is the suffix sum. In this case,
The standard depth-first search would be something like the following.

Consider depth-first search (this is also called preorder traversal) through a graph such as the one below.

Suffix Sums for Depth-First Search

4.47
whether or not the tree structure is binary, ternary, or mixed.

We use a data structure for each node as follows. Note among other things that this has constant space.

1. Every processor links its node to its successor node in the DFS traversal.

Hence 2n – 2 processors.

2. Processors for a downward edge write a 1 into a mask array. Processors for an upward edge write a 0.

3. Run suffix sum in parallel on the weights of step 2.

4. Use suffix sums in parallel to assign traversal values.

Let’s do this in parallel (CREW PRAM) by focusing not on the nodes but on the edges.
edge to the leftmost child node, if that child exists, else to the upward edge between the same nodes.

from the parent to the grandparent, else to itself (at the end of the search). Downward edges link to the
upward edges link to the edge from the parent to the right sibling, if that sibling exists, else to the edge

1. Every processor links its node to its successor node in the DFS traversal.

Otherwise, we are a downward edge.

If the edge is (i, j) (i = j = 0 or thinking nodes i and j), and if the parent of node i is j, then we are an upward edge.

We use the processors to whether we are matched with an upward edge or a downward edge for the following

In greater detail:

<table>
<thead>
<tr>
<th>Parent</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right-sibling</td>
<td>null</td>
<td>null</td>
<td>C</td>
<td>null</td>
<td>H</td>
</tr>
<tr>
<td>Left-most-child</td>
<td>B</td>
<td>D</td>
<td>F</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We saw how to do this in the previous section.

3. Run suffix sum in parallel on the weights of step 2.

\[
\begin{align*}
\langle \forall, C \rangle &\leftarrow 0, \langle C, F \rangle &\leftarrow 1, \langle F, C \rangle &\leftarrow 1, \langle C, \forall \rangle \\
&\leftarrow 0, \langle \forall, B \rangle &\leftarrow 0, \langle B, F \rangle &\leftarrow 0, \langle F, H \rangle \\
&\leftarrow 1, \langle H, F \rangle &\leftarrow 0, \langle F, C \rangle &\leftarrow 1, \langle C, F \rangle \\
&\leftarrow 1, \langle F, B \rangle &\leftarrow 0, \langle B, D \rangle &\leftarrow 1, \langle D, B \rangle &\leftarrow 1, \langle B, \forall \rangle \\
\end{align*}
\]

We know who we are, and write 0 or 1 accordingly:

\[
\begin{align*}
\text{a 0 into a mask array.} \\
\end{align*}
\]

2. Processors for a downward edge write a 1 into a mask array. Processors for an upward edge write

\[
\begin{align*}
\langle \forall, C \rangle &\leftarrow (C, F) &\leftarrow (F, C) &\leftarrow (C, \forall) \\
&\leftarrow (\forall, B) &\leftarrow (B, F) &\leftarrow (F, H) \\
&\leftarrow (H, F) &\leftarrow (F, C) &\leftarrow (C, F) \\
&\leftarrow (F, B) &\leftarrow (B, D) &\leftarrow (D, B) &\leftarrow (B, \forall) \\
\end{align*}
\]
gets assigned.

traversal value. Note that destination nodes of downward arcs are unique; so only one value per node

If we are a downward edge, we assign value to the destination node of the arc as the

4. Use suffix sums in parallel to assign traversal values.

\[ 0, 0, 0, 0, 0 \]

\[ 0, 0, 0, 0, 0 \]

\[ 0, 0, 0, 0, 0 \]

\[ 0, 0, 0, 0, 0 \]

\[ 0, 0, 0, 0, 0 \]
\[ \begin{align*}
1 &= \forall_\emptyset,0,0_\emptyset,\forall_\emptyset,\therefore \\
&\quad \leftarrow *^\emptyset,0_\emptyset,\forall_\emptyset,\therefore \\
&\quad \leftarrow \varphi = \forall_\emptyset,0_\emptyset,\forall_\emptyset,\therefore \\
&\quad \leftarrow \varphi,\therefore = \forall_\emptyset,0_\emptyset,\forall_\emptyset,\therefore \\
&\quad \leftarrow \exists = \forall_\emptyset,0_\emptyset,\forall_\emptyset,\therefore \\
&\quad \leftarrow *^\emptyset,\exists_\emptyset,\forall_\emptyset,\therefore \\
&\quad \leftarrow 3 = \forall_\emptyset,6_\emptyset,\forall_\emptyset,\therefore \\
&\quad \leftarrow 2 = \forall_\emptyset,7_\emptyset,\forall_\emptyset,\therefore \\
&\quad \leftarrow \forall_\emptyset,\forall_\emptyset,\therefore \\
\end{align*} \]
{ 
    if (Tj archetype) {  
        for all processors (Tj) begin  
            for all edges in the tree begin  
                /*  
                /* DFS - value [Tj] in [n]  
                /* edge rank after suffix sum  
                /* edge number of successor edge  
                /* vertex number of right subtree node  
                /* vertex number of leftmost child node  
                /* vertex number of parent node  
                /* number of neighbors in the trees  
                */  
                /* global variables */  
                Depth First Search
DFS-value[y] = u + 1 - position
if parent[y] == i
/* phase 4: assign preorder values */

/* for end */ {

for (k = i + 1; k < ceiling(num/2)-1; k++)
/* phase 3: suffix sum */

1 = position
else
0 = position
if (j == [parent]i)
/* phase 2: weight the edges based on upward or downward */

/* for end */ {

(succesor[j], i) = [succesor]j
else
(succesor[j], child[j]) = [succesor]j, child[j] = num
if (parent[j] == i || parent[j])
/* else */
/* phase */
end

/ * end for all processors * / {
\[
\frac{d}{t - \alpha} + t = \\
\frac{d}{t} \sum_{i=1}^{l} - \frac{d}{s} \sum_{i=1}^{l} + \frac{d}{d} \sum_{i=1}^{l} = \\
\frac{d}{1 - d + \frac{1}{s}} \sum_{i=1}^{l} \geq \left[ \frac{d}{s} \right] \sum_{i=1}^{l}
\]

processors in time \( d \) processors we can simulate \( \frac{1}{s} \) of time \( d \) in time \( s \). So the entire algorithm can be simulated by \( d \) processors.

\[ m = \frac{1}{s} \]

processors in time \( A \) in time \( t \).

\textbf{Proof. Let} \( s \) denote the number of computation steps performed in time step \( i \) for \( i \geq 1 \). Then

\[
\frac{d}{t - \alpha} + t
\]

processors can execute algorithm \( A \) in time \( t \). Then \( p \)

\textbf{Theorem 4.15. Given a parallel algorithm \( A \) that performs \( m \) computation steps in time \( t \), then \( p \)}

\textbf{Brent's Theorem 4.18}
\[(d \frac{d^I}{u} + \frac{d}{u})\Theta\]

minimum execution time

**Theorem 4.16.** Using \(\Theta(\frac{u}{s})\) in (u) processors' parallel prefix can be computed with optimal cost and with

\[\Theta(\frac{u}{s}) = \left(\frac{u}{u - s} - \frac{u}{u + s} - u \frac{s}{u} + u \frac{d^I}{u}\right) = \frac{\lceil \frac{u}{s} \rceil}{\lceil u \frac{s}{u} \rceil - 1 - u} + \frac{u \frac{s}{u}}{\lceil u \frac{s}{u} \rceil}\]

Applying Brent's theorem, if we use \(\left\lceil \frac{u}{s} \right\rceil\) processors, we can do parallel summation in time

Applying Brent's theorem, if we use fewer processors, we can do parallel summation in time

Several steps but fewer processors. So we balance the first part, needing lots of processors, with the latter part, needing fewer processors. So we balance the first part, needing lots of processors, with the latter part, needing fewer processors. Consider, however, that the \(u\) processors are only needed in the first step. After that, we need fewer processors. We can add \(u\) items in time \(\Theta(u)\), but this requires \(u\) processors doing \(u\) in parallel, which is not cost effective. We can add \(u\) items in time \(\Theta(u)\), but this requires \(u\) processors doing \(u\) in parallel, which is not cost effective.

**4.19.1 Parallel Summation**

We can apply Brent's theorem as follows.

**4.19 Reducing Processor Count**
\[
\left( d \log \log + \frac{d}{u} \right) \Theta = \left[ \frac{d}{u} \right] + \left( 1 - d \log \log \right) + 1 - \left[ \frac{d}{u} \right]
\]

Total time is
\[\frac{d}{u}\] time steps.

Taking time steps.

Thus is done in \( \frac{d}{u} \) time steps.

This is done in \( \frac{d}{u} \) time steps.

\( \frac{d}{u} \) time steps.

3. The processors use the parallel algorithm to compute the prefix sums of the blocks in \( \Theta \) time.

2. Each processor (except the last) sequentially computes the overall sum of the values in its block of data.

1. Break up the \( u \) items into \( u \) blocks of \( \frac{d}{u} \) items each.

Let's do the prefix sums of \( u \) items using processors.  

This is not cost optimal, because a single processor can do all the prefix sums in \( u - 1 \) operations.

\[
(u \log u) \Theta = (1 - \left[ u \log u \right] \log u) - u \log u = (\log u - u)^{0=1}_{[1-u \log u]}
\]

operations. So the total number of operations is

\( u - \left[ u \log u \right] \log u \) iterations, and in each iteration \( i \) we do \( u \).
So we want to minimize both time and cost together.

\[
\left( \frac{u \log u}{u} \right) \Theta = d \quad \text{and is no larger than } u \text{ provided } \log u \leq (u) \beta
\]

\[
((u) \beta \log u - u \log u)((u) \beta / u) + u
\]

If we let \( (u) \beta / u = d \), then the cost looks like

\[
(u) \beta \log u - u \log u + (u) \beta
\]

If we let \( (u) \beta / u = d \), then the time looks like

\[
\text{constraint each function with the other.}
\]

From looking at these two functions, it should be clear that the optimal value is going to be based on

\[
d = u \text{. The cost} \]

Now, optimizing this isn't entirely simple. The execution time function has a minimum at \( u \). The cost

\[
\left( d \log d + \frac{d}{u} \right) \Theta = \left( \frac{d}{(d \log d + \frac{d}{u})} - d \log d + u \right) + d \log (d + u) \Theta
\]

Now plug this in to Brent's Theorem, which says we can set an execution time of

\[
(d \log d + u) \Theta
\]

The total number of addition steps is
Parallel algorithm into a more realistic parallel algorithm.

One way in which we can apply parallel algorithms is to determine the cost in characterizing an unrealistic insight from a PRAM algorithm or else we could adapt it to be usable on a real machine.

On the other hand, besides having four fingers and a thumb, sometimes we either can get some useful how we might choose to compute things on a parallel computer.

On the one hand, PRAM models are unrealistic. Very often, PRAM algorithms have little realism on
Clearly, our goal is to find dilation-1 embeddings. These would permit an algorithm for communication structure $\mathcal{G}$.

\textbf{Definition 4.18.} Let $f$ be the function that embeds a graph $\mathcal{G}$ into a graph $\mathcal{G}'$ from $\Lambda$ to $\Lambda'$.

\textbf{Definition 4.17.} An embedding of a communication structure of a binary tree.

On other machines, such as the T3E, in which the interconnect supports a 2-dimensional mesh, it may be desirable to implement the communication structure on those "realistic" machines. In fact that of a binary tree, permitted such "unrealistic" algorithms to be implemented on those "realistic" machines. A number of PRAM algorithms have already been seen to rely upon a binary tree structure. Computers such as the Thinking Machines CM-2 and CM-5, in which the interconnect and communication structure were...
\[ 2^h - 1 < 2^h^2 + 2^h + 1 \text{ for } h > 1. \]

Given node is \( 2^h^2 + 2^h + 1 \). The total number of nodes in a complete binary tree of height \( h \) is \( 2^h - 1 \). But

**Proof.** We count mesh points. In a 2-D mesh the number of nodes that are less than or fewer hops away from a

**Theorem 4.20.** A complete binary tree of height \( h \) cannot be embedded into a 2-dimensional mesh

**Example**

**Proof.** Theorem 4.19. There exists a dilation-

embeddings of a ring of nodes into a two-dimensional mesh

of \( n \) nodes if and only if \( n \) is even.
We note that this is a proof that does't tell us anything about the structure of embeddings or dilations.
The following is a dilation 1 embedding of a binary tree of height 4 into a 2-D mesh.

The following is a dilation 1 embedding of a binary tree of height 3 into a 2-D mesh.
next tree level.

30 dots unused. The structure is unimportant; no structure is going to provide enough dots to allow for the

Count up the unused dots. We will need 32 dots for the next level of the binary tree, but there are only

This is why the next one won't work.
In general we will want to compare and measure network communication capability based on four criteria.

Definition 4.2.3. The bisection width of a network is the minimum number of arcs that one must cut in order to separate the network into two halves ("halves" to within one node).

Definition 4.2.2. The diameter of a network is the maximum distance between any two nodes.

Proof. This is done recursively with an H-tree as above.

Theorem 4.2.1. A complete binary tree of height $n$ has a dilation $\lceil n/2 \rceil$ embedding into a $2$-D mesh.

But now consider the following embedding using an H-tree.
the hypercube and to two nodes of the k-cycle.

Definition 4.25. A cube connected cycles network is a k-dimensional hypercube in which the nodes
node numbers.

are connected if they differ by 1 in the Hamming distance between the binary representations of their

Definition 4.24. A hypercube is a network with n nodes, numbered 0, 1, ..., in which nodes

4. The maximum edge length needed to realize the network
3. Number of edges per node
2. Bisection width
1. Diameter
Theorem 4.28. A dilatation-1 embedding of a complete binary tree of height \( h \) into a hypercube of size \( n = 2^h \):

<table>
<thead>
<tr>
<th>( N )</th>
<th>( \lambda )</th>
<th>( \gamma )</th>
<th>( 2h - 1 )</th>
<th>( 2^h )</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>( \lambda )</td>
<td>( \gamma )</td>
<td>( 2h - 1 )</td>
<td>( 2^h )</td>
<td>cube conm cubes</td>
</tr>
<tr>
<td>( N )</td>
<td>( N )</td>
<td>( \gamma )</td>
<td>( 2h )</td>
<td>( 2^{h+1} )</td>
<td>hypercube</td>
</tr>
<tr>
<td>( N )</td>
<td>( \lambda )</td>
<td>( \gamma )</td>
<td>( 2h )</td>
<td>( 2^{h+1} )</td>
<td>butterfly</td>
</tr>
<tr>
<td>( N )</td>
<td>( \lambda )</td>
<td>( 1 )</td>
<td>( 2h - 1 )</td>
<td>( 2^h )</td>
<td>binary tree</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( \lambda )</td>
<td>( \gamma )</td>
<td>( 2h )</td>
<td>( 2^{h+1} )</td>
<td>3-D mesh</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( \lambda )</td>
<td>( \gamma )</td>
<td>( 2h )</td>
<td>( 2^{h+1} )</td>
<td>2-D mesh</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( \lambda )</td>
<td>( 1 )</td>
<td>( h - 1 )</td>
<td>( h )</td>
<td>1-D mesh</td>
</tr>
</tbody>
</table>

Definition 4.27. An omega network is a composition of \( k \) shuffle exchange networks.

Definition 4.26. A shuffle exchange network consists of \( n = 2^h \) nodes with two kinds of connections:

- Significant bit. The shuffle connections are unidirectional links from node \( k \) and node \( k \) (mod \( n \)).
Theorem 4.30. A dilation-1 embedding of a balanced binary tree of height $n$ into a hypercube of dimension $n + 1$ exists for all $n < 1$.

Theorem 4.29. A dilation-2 embedding of a complete binary tree of height $n$ into a hypercube of dimension $n + 1$ exists for all $n < 1$.

We can't do as well as we would like to. However, we can do almost as well.

$\square$

If, on the other hand, the height is even, then more than half the nodes are an even distance away from the root. What if the embedding isn't possible?

Embed the tree into the cube.

Now, a tree of height $n + 1$ has $2^n + 1$ nodes in all, and $2^n$ nodes that are the leaves at height exactly $n$. It is therefore not possible to embed the tree into the cube. And the optimal embedding would have to use all the nodes of the hypercube except one.

Proof. Note that a complete binary tree of height $n + 1$ has $2^{n+1}$ nodes. So the optimal embedding, if it were to exist, would be of a height-1 tree into an $n + 1$-dimensional hypercube.
Theorem 4.31. Any 2-dimensional mesh with n vertices can be embedded into a \( \mathbb{R}^n \) in \( n \)-dimensional hyperplane with dilation \( 2 \).

In a different direction, we cannot do as well as we would like for a mesh, but once again we can do the