5 Parallel Algorithms

5.1 Basic Concepts

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

We count the number of operations executed by the processor.

We normally do the opcount and then pay lip service to issues like memory, etc.

With parallel computer algorithms, we have no choice but to make some assumptions up front, and these assumptions have a major affect on the kind of algorithms that we design, analyse, and use.

We clearly want algorithms to be scalable. This is a slightly vague term, as are many terms. Basically, we would like running time $N$ with $k$ processors to become time $N/2$ with $2k$ processors.

**Definition 5.1.** An algorithm is said to be **scalable** if the running time of the algorithm decreases roughly linearly with the number of processors.

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 overhead overtakes the part of the computation that is in fact getting useful work done.

And in the real world, the analysis of parallel algorithms can be difficult because we’d have to balance communication time with compute time and with memory access time. And this is not easy.
5.1.1 Symmetric Multi Processors (SMPs)

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

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

This is what we would like to have. But we can’t have it because it’s too expensive to interconnect processors with memory and to guarantee Programs have access to all of the machine’s memory.

5.1.2 Parallel Computers with an Interconnect Structure

(e.g., Cray T3E, Thinking Machines CM-2 and CM-5, Meiko computing surface)

Many parallel compute nodes, each with its own memory.

The processors are connected to one another with a fixed interconnect. E.g.:

T3E has 3-dimensional toroidal nearest neighbor connections.
CM-2 and CM-5 had a logarithmic network

With some machines (T3E), processors’ memories can be aggregated into a larger shared memory space.

With other machines (CM-2, CM-5), processors are really limited to using only their local memory, and communications from one processor to another must be explicit.

5.1.3 Cluster Computers with Non-Uniform Memory Access (NUMA)

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

Nodes of SMP computers connected to each other
Memory access on the node is faster than access off node through the network.

Programs see a shared memory space whose implementation may or may not act like a shared memory space, and programs definitely work better if computations are local.

5.1.4 Distributed Computers With Poor Interconnect

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

Individual computers on a network.

Beowulf is a dedicated network.

SNow is just “the usual” network linking ordinary workstations.

Very loose coupling of processors.

Computations must be primarily local.

Explicit message passing from processor to processor.
5.2 The Assumptions of Parallel Processing

Definition 5.2. A Parallel Random Access Machine, or PRAM, comprises

- a control unit
- an unbounded number of processors each with its own local memory
- a global memory
- an interconnect between the processors and the global memory

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

The critical question with a PRAM is what happens when parallel processors choose to read or write the same memory locations in the same time step.

In decreasing order of restriction and increasing order of power, we have the following:

- **Exclusive Read Exclusive Write (EREW):** Only one processor may read or write a given location in a given time step.

- **Concurrent Read Exclusive Write (CREW):** Only one processor may write to a given location in a given time step, but any number of processors may read the same location in the same time step. This the usual PRAM model.
• Concurrent Read Concurrent Write (CRCW):
  
  – **Common** write: So long as all processors are writing the same value to the same location, the write is permitted.
  
  – **Arbitrary** write: All processors can write to the same location. The processor that succeeds in writing the value (or the processor that happens to write last) is chosen arbitrarily.
  
  – **Priority** write: Only the processor with the highest priority (in some scheme of static priorities) gets to write.

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

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

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

• A common algorithm works on an arbitrary machine with no slowdown.

**Lemma 5.3.** A $p$ processor EREW PRAM can sort a $p$-element array in global memory in $\Theta(\log p)$ time.

**Theorem 5.4.** An $p$-processor EREW machine can simulate a $p$ processor priority algorithm with no more than a $\Theta(\log p)$ slowdown.

*Proof.*  
• The priority machine (PPRAM) uses processors $P_1, \ldots, P_p$ and memory $M_1, \ldots, M_m$.

• The EREW PRAM uses extra global locations $T_1, \ldots, T_p$ and $S_1, \ldots, S_p$ to simulate each step of the priority algorithm.
• When $P_i$ in PPRAM accesses location $M_j$, the EREW PRAM writes $(j, i)$ in extra location $T_i$.

• The EREW PRAM then sorts the extra location values in time $\log p$.

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

• Now processor $P_1$ reads $(i_1, j_1)$ from location $T_1$ and writes a 1 in location $S_{j_1}$.

• Processors $P_2$ through $P_p$ read locations $T_k$ and $T_{k-1}$

• If $i_k \neq i_{k-1}$ then $P_i$ writes a 1 into $S_{j_k}$, else write a 0.

• Now the elements of $S$ with value 1 correspond to the highest priority processors accessing each memory location.

• For a write operation, the highest priority processor doing the write gets to write.

• For a read, the highest priority processor gets to read and then can communicate the value to the other processors in $\Theta(\log p)$ time.
5.3 Parallel Communications and Process Spawning

We first can observe that with any of the PRAM models, we can spawn \( p \) processes on \( p \) processors in \( \lg p \) time using a binary tree. A control processor can spawn two processes, each of those can spawn two more, and so forth.

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

**Definition 5.5.** The broadcast problem is the problem on a parallel computer of having one processor broadcast a value to all other processors.

**Definition 5.6.** The total exchange problem is the problem on a parallel computer of having every processor exchange a value with every other processor.

**Proposition 5.7.** In a CREW PRAM, broadcast takes constant time.

*Proof.** In a CREW PRAM, processor \( P_0 \) can broadcast to all other processors by writing in one time step to a specific memory location. In the next time step, all other processes can read from that location. \( \square \)

**Proposition 5.8.** 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 1. Each processor now reads the \( n - 1 \) values in the next \( n - 1 \) time steps. \( \square \)

**Proposition 5.9.** 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. The best possible situation is that of the CREW PRAM. Each processor must receive \( n - 1 \) units of information, and therefore regardless of how fast that information can be made available to be read by each processor, it cannot in fact be read by the processors in fewer than \( n - 1 \) time steps. \( \square \)

**Proposition 5.10.** 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 logarithmic in the number of processors.

Proof. Consider the following communication pattern. We have eight processors running across, exchanging information in time steps going down.

Processor \( i \) starts with an 8-bit string with only the \( i \)-th bit set, which we write in hex. The arcs going down represent addition (or XOR). After \( \lg n \)
steps, every processor has received information from every other processor.

The communication pattern described above is called a butterfly and is associated, among other things, with the Fast Fourier Transform (FFT).

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

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

**Definition 5.11.** The cost of a (PRAM) computation is the product of the number of processors and the time complexity of the computation.

For example, parallel merge (discussed below) takes time $\log n$ on $n$ processors and thus has cost $n \log n$.

**Definition 5.12.** A parallel computation (algorithm) is said to be cost optimal if the cost is in the same time complexity as the optimal sequential algorithm. of a (PRAM) computation is the product of the number of processors and the time complexity of the computation.

For example, parallel merge as discussed below is not cost optimal since the sequential time (1 processor times $n$ comparisons) is asymptotically smaller than the parallel $n \log n$ cost.

As always with parallel algorithms, sometimes we might care (about cost-optimality), and sometimes we might not.
5.4 Parallel Summation

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 $\lg n$ time in parallel, in the obvious way.

$$a + b + c + d + e + f + g + h$$

$$a + b + c + d, e + f + g + h$$

$$a + b, c + d, e + f, g + h$$

$$a, b, c, d, e, f, g, h$$

And of course “add” can be extended to any associative operation.
5.5 Parallel Merge

**Theorem 5.13.** On a CREW PRAM, we can with \( n \) processors perform a merge of two sorted lists of \( n/2 \) elements in time \( \Theta(\lg n) \).

*Proof.* Consider two sorted lists, \( A \) and \( B \), each of length \( n/2 \). To each element \( a_i \) in list \( A \) we assign a processor \( P_{a_i} \). Using this processor, we can determine by binary search with \( \lceil \lg(n/2) \rceil \) comparisons that \( b_j \leq a_i \leq b_{j+1} \) for two elements \( b_j \) and \( b_{j+1} \) in list \( B \).

If we have indexed both lists from 1 through \( n/2 \), this means that \( a_i \) should be placed in location \( i + j \) in the merged list, because it should be preceded by the \( i - 1 \) elements \( a_1 \) through \( a_{i-1} \) and by the \( j \) elements \( b_1 \) through \( b_j \). (Note that duplicates don’t matter in this argument.)

Using the \( n/2 \) processors assigned to list \( A \) in this fashion, and using analogously the \( n/2 \) processors assigned to list \( B \), we can find in parallel the correct position in the sorted list of every element in each of lists \( A \) and \( B \) in \( \lceil \lg(n/2) \rceil \) steps.

All this involves concurrent reading, but no writing as yet. We now write, in parallel, each element into its proper location in the merged array. This can be done in parallel because each element goes into a different location.

(We have fudged ever so slightly on the question of duplicate values. To make this truly rigorous we must specify a consistent convention for breaking ties. Otherwise, in the case of \( a_i = a_{i+1} \) and \( b_j = b_{j+1} \), we would run the risk of matching \( i \) with \( j + 1 \) and \( i + 1 \) with \( j \), say, and writing the same value to location \( i + j + 1 \) but no value to \( i + j \).)

Note that we do \( n \lg n \) comparisons in the parallel algorithm instead of
the $n$ of the serial algorithm. But we do them on $n$ processors and hence $n$
comparisons at a time, resulting in the $\lg n$ execution time. So this is not
cost optimal.
5.6 Parallel Prefix

The *prefix sums* of an array

\[ a_1, a_2, \ldots, a_n \]

are the sums

\[ a_1 \]
\[ a_1 \oplus a_2 \]
\[ a_1 \oplus a_2 \oplus a_3 \]
\[ \ldots \]
\[ a_1 \oplus a_2 \oplus \ldots \oplus a_n \]

where we can interpret \( \oplus \) to be any associative operation.

For example, we might have

<table>
<thead>
<tr>
<th>array</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>prefix sums</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
<td>28</td>
<td>36</td>
</tr>
</tbody>
</table>
We can compute all the prefix sums in parallel in the following way.

<table>
<thead>
<tr>
<th>array</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>(a_i \leftarrow a_{i-1} + a_i)</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>(a_i \leftarrow a_{i-2} + a_i)</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td>(a_i \leftarrow a_{i-4} + a_i)</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
<td>28</td>
<td>36</td>
</tr>
<tr>
<td>prefix sums</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
<td>28</td>
<td>36</td>
</tr>
</tbody>
</table>

where in each of the intermediate steps we only do the addition if in fact the \(i - k\) subscript is positive.

**Theorem 5.14.** The prefix sums of an array of \(n\) elements can be computed on a CREW PRAM in \([\lg n]\) steps.

**Proof.** This should be obvious. \(\square\)
Now, why on earth would we want to do this? What does this rather odd looking prefix sum function do for us?

Look at it in exactly the opposite way. Parallel prefix is a fast operation. Let’s look for ways in which it could be incorporated into some other algorithm.

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 an array and all the values greater than the pivot to the last part of an array, then we can make use of parallel prefix in the following way.

Assume that the pivot element is 4, and that we have the array

```
[7 2 9 4 3 1 7 8]
```

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.

```
[7 2 9 4 3 1 7 8]
[0 1 0 1 1 1 0 0]
```

Now do the parallel prefix.

```
[7 2 9 4 3 1 7 8]
[0 1 0 1 1 1 0 0]
[0 1 1 2 3 4 4 4]
```

Now, if processor $P_i$ reads locations $i$ and $i - 1$ and finds different values in the prefix sums $s_i$ and $s_{i-1}$, it knows that it should write its value into location $i$ in the “first part” array.
We create the “last part” of the array with an analogous operation, differing
only in that we flip 0 and 1 in creating the mask array.

Compare this to the data movement part of the quicksort on a serial
machine. There, we use \( n \) comparisons to move \( n \) elements. Here, we use
\( 2 \lg n + 2 \) for creating the two masks and doing the two parallel prefix op-
erations.

So yes, prefix sums seem a little odd, but they can be useful!
5.7 Parallel List Ranking

Now consider the suffix sum problem in a linked list. Actually, we'll make the problem slightly easier by making the values to be summed all 1 except for the end of the list, to which we'll assign 0.

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

What we will do is iterate

\[
\text{suffix sum} += \text{value}[i] \\
\text{next}[i] = \text{next}[\text{next}[i]]
\]

\(\lg n\) times, provided that the \text{next}[\text{next}[i]] operation doesn't go beyond the end of the list. This gives us the following.
Iteration 0.

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

Iteration 1.

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

Iteration 2.

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

Iteration 3.

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

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

Takes $n \log n$ operations in $\log n$ time.
5.8 Brent’s Theorem

Theorem 5.15. Given a parallel algorithm A that performs m computation steps in time t, then p processors can execute algorithm A in time

\[ t + \frac{m - t}{p}. \]

Proof. Let \( s_i \) denote the number of computation steps performed in time step \( i \), for \( 1 \leq i \leq t \). Then

\[ \sum_{i=1}^{t} s_i = m. \]

Using \( p \) processors we can simulate step \( i \) in time \( \lfloor s_i / p \rfloor \). So the entire algorithm can be simulated by \( p \) processors in time

\[
\sum_{i=1}^{t} \left\lfloor \frac{s_i}{p} \right\rfloor \leq \sum_{i=1}^{t} \frac{s_i + p - 1}{p} = \sum_{i=1}^{t} \frac{p}{p} + \sum_{i=1}^{t} \frac{s_i}{p} = t + \frac{m - t}{p}.
\]

\[ \square \]
5.9 Reducing Processor Count

We can apply Brent’s theorem as follows.

5.9.1 Parallel Summation

We can add $n$ items in time $\lg n$, but this requires $n$ processors doing $n \lg n$ additions, which is not cost optimal. Consider, however, that the $n$ processors are only needed in the first step. After that, we need fewer and fewer processors. So we balance the first part, needing lots of processors, with the latter part, needing several steps but fewer processors.

Applying Brent’s theorem, if we use $\left\lfloor \frac{n}{\lg n} \right\rfloor$ processors, we can do parallel summation in time

$$[\lg n] + \frac{n - 1 - [\lg n]}{[\frac{n}{\lg n}]} = \Theta(\lg n + \lg n - \frac{\lg n}{n} - \frac{\lg^2 n}{n}) = \Theta(\lg n)$$

5.9.2 Parallel Prefix

**Theorem 5.16.** Using $\Theta(n/\lg n)$ processors, parallel prefix can be computed with optimal cost and with minimum execution time

$$\Theta\left(\frac{n}{p} + \lg p\right)$$

**Proof.** Exercise. □
5.10 Lessons from Parallel Algorithms

On the one hand, PRAM models are unrealistic. Very often, PRAM algorithms have little real bearing on how we might choose to compute things on a parallel computer.

On the other hand, besides having four fingers and a thumb, sometimes we either can get some useful insight from a PRAM algorithm or else we could adapt it to be useful on a real machine.

One way in which we can apply parallel algorithms is to determine the cost in changing an unrealistic parallel algorithm into a more realistic parallel algorithm.
5.11 Embeddings and Dilations

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 in fact that of a binary tree, permitted such “unrealistic” algorithms to be implemented on those “realistic” machines.

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 of a binary tree.

Definition 5.17. An embedding of a graph $G = (V, E)$ into a graph $G' = (V', E')$ is a function $\phi$ from $V$ to $V'$.

Definition 5.18. Let $\phi$ be the function that embeds a graph $G = (V, E)$ into a graph $G' = (V', E')$. The dilation of the embedding is defined as follows:

$$\text{dil}(\phi) = \max\{\text{dist}(\phi(u), \phi(v)) | (u, v) \in E\}$$

where $\text{dist}(a, b)$ is the distance between vertices $a$ and $b$ in $G$.

Clearly, our goal is to find dilation-1 embeddings. These would permit an algorithm for communication structure $G$ to be implemented without slowdown on a machine with communication structure $G'$.

Theorem 5.19. There exists a dilation-1 embedding of a ring of $n$ nodes into a two-dimensional mesh of $n$ nodes if and only if $n$ is even.

Proof. Exercise. \qed
Example

\begin{center}
\includegraphics[width=0.4\textwidth]{example.png}
\end{center}

**Theorem 5.20.** A complete binary tree of height \(> 4\) cannot be embedded into a 2-dimensional mesh without increasing the dilation beyond 1.

*Proof.* We count mesh points. In a 2-D mesh the number of nodes that are \(k\) or fewer hops away from a given node is \(2k^2 + 2k + 1\). The total number of nodes in a complete binary tree of height \(k\) is \(2^k - 1\). But \(2^k - 1 > 2k^2 + 2k + 1\) for \(k > 4\).

We note that this is a proof that doesn’t tell us anything about the structure of embeddings or dilations.
Example

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

![Binary tree of height 3](image1)

The following is a dilation-1 embedding of a binary tree of height 4 into a 2-D mesh.

![Binary tree of height 4](image2)
This is why the next one won’t work.

Count up the unused dots. We will need 32 dots for the next level of the binary tree, but there are only 30 dots unused. The structure is unimportant; no structure is going to provide enough dots to allow for the next tree level.
But now consider the following embedding using an H-tree.

![H-tree diagram]

**Theorem 5.21.** A complete binary tree of height $n$ has a dilation-$\lceil n/2 \rceil$ embedding into a 2-D mesh.

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