Let P be a computational problem that can be solved in $T_1(n)$ time with one processor. If using p-processors we can solve P in $T_p(n)$ time, then we are said to have achieved a *speed-up* of $T_1(n)/T_p(n)$. If we have p processors, we would expect to get a speed-up of p; but this is often impossible, because we have to reckon with the inherent sequentialities of the problem. (For example, what 100 men build in 100 days won't be built in one day by 10000 men.) Nevertheless, many times a non-trivial $\omega(1)$ speed-up would still be achievable.

Parallel Random Access Machine

The concerns of parallel algorithm design are often very different from those of sequential algorithm design. The performance of a parallel algorithm depends on a variety of factors like processor allocation, synchronisation and resource sharing, which are not present in sequential computation. Because of these, a consensus has not yet been reached on the ideal model of computation to be used in designing parallel algorithms. The Parallel Random Access Machine (PRAM) is probably the most widely used model of parallel computation.

A PRAM has p processors, all having access to a common memory. A PRAM is synchronous in that all processors are fed the same clock. Each processor is similar to a random access machine, the standard model of sequential algorithm design, and is assigned a unique index from the range $[1 \ldots p]$. PRAM models can be classified according to the constraints they impose on global memory access. An Exclusive Read Exclusive Write (EREW) PRAM does not allow simultaneous access by more than one processor to the same memory location, for read or write purposes. A Concurrent Read Exclusive Write (CREW) PRAM allows simultaneous access for reads, but not for writes. A Concurrent Read Concurrent Write (CRCW) PRAM allows simultaneous access for both reads and writes.

Based on the write conflict resolution schemes used, CRCW PRAMs can be further subclassified.

PRIORITY CRCW PRAM: in any concurrent write, the processor with the lowest index succeeds.

ARBITRARY CRCW PRAM: in any concurrent write, an arbitrary one of the processors succeeds; the programs should work irrespective of the identity of the successful processor.

COMMON CRCW PRAM: all processors involved in a concurrent write should write the same value; otherwise, the PRAM may behave unpredictably.

COLLISION CRCW PRAM: when multiple processors write in the same memory location a special collision symbol appears in that location.

TOLERANT CRCW PRAM: when multiple processors write in the same memory location, the content of that location remains unchanged.

Simulations among CRCW Models

If an algorithm that runs on model $\mathcal{A}$ in T time can be simulated on model $\mathcal{B}$ in $O(T)$ time, then $\mathcal{B}$ is at least as powerful as $\mathcal{A}$; $\mathcal{A} \preceq \mathcal{B}$.

Since all CRCW PRAMs are similar except in their write conflict resolution rules, we need to bother about the simulation of writes only. Suppose in a particular concurrent write of the simulated PRAM, processor i wants to write value $V[i]$ in cell $M[i]$.

ARBITRARY on PRIORITY: An algorithm written for ARBITRARY can be run on PRIORITY without any change. ARBITRARY assumes only that in every concurrent write some one processor succeeds, and that is guaranteed by PRIORITY. So, ARBITRARY $\preceq$ PRIORITY.
COMMON on ARBITRARY: An algorithm written for COMMON can be run on ARBITRARY without any change. So, COMMON $\preceq$ ARBITRARY.
COLLISION on ARBITRARY: Each step of collision can be simulated by three steps of ARBITRARY. (1) Make every processor i, write the integer i in cell $M[i]$. (2) Make every processor i, if it had failed in step 1, write a special collision symbol in cell $M[i]$. (3) With every processor i that succeeded in step 1, if cell $M[i]$ did not change its contents during step 2, write $V[i]$ in cell $M[i]$. So, COLLISION $\preceq$ ARBITRARY.
TOLERANT on COLLISION: Assume that there is an auxilliary cell attached to each original cell. First let the processors attempt to increment their corresponding auxilliary cells. If they succeed they go on to write in the original cell. Otherwise, they should do nothing. So, TOLERANT $\preceq$ COLLISION.

All the above relations are in fact strict; $\preceq$ can be replaced by $\prec$ in each case. This is because, in each case, we can find a problem $\mathcal{P}$ such that $\mathcal{P}$ has a time lower bound of $\Omega(T)$, for some T, on the weaker model, but can be solved in $o(T)$ time on the stronger model. COMMON is incomparable with COLLISION or TOLERANT in that there are problems $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $\mathcal{P}_1$ can be solved faster on COMMON than on COLLISION, and $\mathcal{P}_2$ can be solved faster on TOLERANT than on COMMON.

CRCW Algorithms for finding the OR of n bits

(In the following pseudo-codes for algorithms I stands for the processor index. All the processors execute the same code, but interprets I as its own index. In other words, the algorithm is parameterised on I. In any step, if a processor finds that no work is specified for it, it remains idle. In steps where all processors must partake but their work has no dependence on I, the range of active processors will not be mentioned at all.)

OR-COMMON
Input: Array $A[1 \ldots n]$ of n bits
Output: A bit R that is the OR of the bits in A.

Step 1: $R = 0$; /* Concurrent write of 0 in R by all processors */

Step 2: pardo for $1 \leq I \leq n$ if $(A[I] == 1)$ $R = 1$;

Step 3: return R; /* CW of R at the return address by all processors */

OR-TOLERANT
Input: Array $A[1 \ldots n]$ of n bits.
Output: A bit R that is the OR of the bits in A.

Step 1: pardo for $I = 1$ $R = 1$;

Step 2: pardo for $1 \leq I \leq n$

if $(I == 1)$ $R = 0$;

else if $(A[I] == 1)$ $R = 0$;

Step 3: pardo for $I = 1$ if $(R == 0$ && $A[1] == 1)$ $R = 1$;

Step 4: return R;

WT Scheduling Principle

A PRAM model is said to be *self-simulating* , if for all $N \geq n \geq 1$, a PRAM of that model of size n can simulate a single step of another PRAM of the same model of size N in $O(\frac{N}{n})$ time. All CRCW PRAMs we have seen except TOLERANT are self-simulating. TOLERANT is not known to be self-simulating.

For example, consider the simulation of an N-processor PRAM on an n processor PRAM, both COMMON: Divide the N "virtual" processors into groups of size n each, and for each concurrent write, with the n "actual" processors activate the groups one by one. Since all the processors writing in the same memory location are guaranteed to write the same value, it is alright to stagger the writing into several phase. Thus, the simulation of a single step takes $O(\frac{N}{n})$ time.

The cost or work of an algorithm that runs in t time using p processors is the time-processor product pt.

If Seq(n) is the worst-case running time of the fastest known sequential algorithm for a problem of size n, an optimal parallel algorithm for the same problem runs in $O(\text{Seq}(n)/p)$ time using p processors. Optimality is many times a more important concern than fastness.

The degree of parallelism of a parallel step is the number of instructions in it; it is the same number of processors required to execute it in one clock unit.

Consider a parallel algorithm presented in T steps. Say the degree of parallelism of the i-th step is w_i. So, the total number of instructions in the algorithm is $W = \Sigma_{i=1}^{T} w_i$. If we use $P = \max(w_i)$ processors, the algorithm can be executed in exactly T steps. But the cost of this execution may be $\omega(W)$. For example, say $w_1 = n \log n$, while $w_i = n$, for $i > 1$. Say, $T = \log n$. The afore-mentioned execution takes $\log n$ time with $n \log n$ processors, which amounts to a cost of $O(n \log^2 n)$, whereas $W = O(n \log n)$.

Assume that our model is self-simulating, and that we have $p \leq P$ processors. Thus, the i-th step of the algorithm can be simulated in $\lceil w_i/p \rceil$ time. So, the total time taken

$$T_1 = \Sigma_{i=1}^{T} \lceil w_i/p \rceil \leq \Sigma_{i=1}^{T} (w_i/p + 1) = T + W/p$$

If we choose p as $\lceil W/T \rceil$, $T_1 = O(T)$ and the total cost of the execution would be $O(W)$.

Hence, we can describe our algorithms as having a number of parallel steps of varying degrees of parallelism, without bothering about the actual number of processors available, provided, our presentation makes clear the parallelism of each step. This principle is generally referred to as *Brent's Theorem* or *WT Scheduling Principle*.

Wherever possible, we will use this paradigm to represent our algorithms.

Some Standard Parallel Algorithmic Techniques

Balanced Trees

For some problems, constructing a balanced binary tree over the input, and traversing it back and forth level by level gives an efficient parallel algorithm.

OR-EREW

Input: Array $A[1 \ldots n]$ of n bits. For simplicity, assume that n is a power of 2.

Output: $R =$ OR of the bits in A. Model: EREW PRAM.

```
{
    pardo for 1 ≤ I ≤ n
        B[n − 1 + I] = A[I];
    for s = 1 to log n do
        pardo for n/2^s ≤ I ≤ n/2^(s−1) − 1
            B[I] = B[2I] ∨ B[2I + 1];
    pardo for 1 == I
        return B[1];
}
```

The copying of A into B can be done in $O(1)$ time and $O(n)$ operations. The s-th iteration of the for-loop can be executed in $O(1)$ time and $O(n/2^s)$ operations; hence the entire for-loop takes $O(\log n)$ time at $O(n)$ operations. Thus by Brent's Theorem, with $n/\log n$ processors, OR-EREW can be run in $O(\log n)$ time. It is optimal.

Incidentally, finding the OR of n bits on even a CREW PRAM with an infinite number of processors would take $\Omega(\log n)$ time.

PREFIX-SUMS-1

Input: Array $A[1 \ldots n]$ of integers. For simplicity, assume that n is a power of 2.

Output: An array $B[1 \ldots n]$ such that $B[i] = \Sigma_{j=1}^{i-1} A[j]$. Model: EREW PRAM.

```
{
    pardo for 1 ≤ I ≤ 2n − 1
        L[I] = M[I] = R[I] = 0;
    pardo for 1 ≤ i ≤ n
        M[n − 1 + I] = A[I];
    for s = 1 to log n do
        pardo for n/2^s ≤ I ≤ n/2^(s−1) − 1
            M[I] = (L[I] = M[2I]) + M[2I + 1];
    for s = log n to 1 do
        pardo for n/2^s ≤ I ≤ n/2^(s−1) − 1
        {
            R[2I] = R[I];
            R[2I + 1] = L[I] + R[I];
```

```
        }
    pardo for 1 ≤ I ≤ n
        B[I] = M[n − 1 + I] + R[n − 1 + I];
    return B;
}
```

PREFIX-SUMS-1, as presented above, has $O(\log n)$ steps for a total of $O(n)$ operations. Thus by Brent's Theorem, with $n/\log n$ processors, PREFIX-SUMS-1 can be run in $O(\log n)$ time.

Pointer Jumping

A technique called Pointer Jumping or Recursive Doubling is useful in algorithms on linked lists and rooted trees. The following algorithm illustrates the technique.

LIST-RANK

Input: A linked list represented as an array $s[1 \ldots n]$ of pointers; $s[i]$ is the successor of node i of the list; if node i is the last in the list, $s[i] = i$.

Output: An array $r[1 \ldots n]$ of "ranks", the number of pointers to be traversed in going from node i to the last node in the list. Model: EREW PRAM.

```
{
    pardo for 1 ≤ I ≤ n
        if (s[I] == I) r[I] = 0;
        else r[I] = 1;
    for s = 1 to log n do
        pardo for 1 ≤ I ≤ n
            if (s[I]! = s[s[I]])
            {
                r[I]+ = r[s[I]];
                s[I] = s[s[I]];
            }
    return r;
}
```

In each step of the for-loop, each node in the list "jumps" over the successor and makes the successor's successor the new successor. In $\log n$ iterations, thus every node is left pointing to the last node of the list. Note that the ranks are computed on the fly. In fact, not just ranking, but the prefix of any associative operation (like summation, multiplication) can be found this way.

In each iteration all the nodes participate; thus the cost of each iteration is $O(n)$, and the cost of the whole algorithm is $O(n \log n)$. This is not optimal, because sequentially lists can be ranked in $O(n)$ work.

Algorithm LIST-RANK was invented by J C Wyllie in his PhD thesis (Cornell 1979). Finding an optimal $O(\log n)$ time list-ranking algorithm then remained open for another seven years. We'll study that algorithm later.

Accelerated Crowding

Processor advantage of an algorithmic instance is the ratio of the number of processors available to the problem size.

Suppose for a problem $\mathcal{P}$ we have a super-fast but wasteful-in-processors algorithm. Say, we want to design a moderately fast, economic-in-processors algorithm for $\mathcal{P}$. Sometimes a solution can be found by iteratively reducing the problem size so that the processor advantage of the new instance increases exponentially through the iterations.

As an example, see the problem of finding the minimum of n numbers on a CRCW PRAM.

MIN1

Input: Array $A[1 \ldots n]$ of integers.

Output: The minimum integer R in A. Model: COMMON CRCW PRAM

Step 1: pardo for $1 \leq i \leq n$, $1 \leq j \leq n$ $\quad B[i][j] = (A[i] > A[j])$;

Step 2: pardo for $1 \leq i \leq n$ $\quad C[i] =$OR-COMMON$(B[i])$;

Step 3: pardo for $1 \leq i \leq n$ $\quad$ if $(C[i] == 0)$ $\; R = A[i]$;

Step 4: return R;

(Exercise: Rewrite the above algorithm so that the processor allocation is obvious. Hereafter, in algorithm descriptions the pardo variable will not always represent processor indices, but will still be indicative of the parallelism. In each case, figure out how the processor allocation should go.) Algorithm MIN1 runs in $O(1)$ time with n^2 processors. Now, suppose we want to design an algorithm that uses nr processors, for $1 < r < n$; that is, the initial processor advantage is r. We divide the array into segments of size r each, allocate r^2 processors for each, solve each in $O(1)$ time; so, we are left with an instance of size n/r—the processor advantage is now r^2.

MIN2

Input: Array $A[1 \ldots n]$ of integers; r the processor advantage.

Output: The minimum integer R in A. Model: COMMON CRCW PRAM

Step 0: if $(n == r)$ return MIN1$(A[1 \ldots n])$;

Step 1: pardo for $1 \leq i \leq n/r$ $B[i] =$MIN1$(A[(i-1)r + 1 \ldots ir])$;

Step 2: return MIN2$(B[1 \ldots n/r], r^2)$;

If $T(n, r)$ is the time taken by a call MIN2$(A[1 \ldots n], r)$, then for some constant c,

$$T(n,r) = T(n/r, r^2) + c = T(n/r^3, r^4) + 2c = T(n/r^7, r^8) + 3c = \cdots = T(n/r^{2^s - 1}, r^{2^s}) + sc$$

With $s = O(\log(\frac{\log n}{\log r}))$, thus $T(n, r) = O(s) = O(\log(\frac{\log n}{\log r}))$.

For some constant ϵ, $0 < \epsilon < 1$, let $r = n^\epsilon$. Then $T(n, n^\epsilon) = O(1/\epsilon) = O(1)$. That is, **the minimum of n numbers can be found in $O(1)$ time with $n^{1+\epsilon}$ processors on a COMMON CRCW PRAM.**

Instead let $r = 2$. Then $T(n, 2) = O(\log\log n)$. That is, **the minimum of n numbers can be found in $O(\log\log n)$ time with n processors on a COMMON CRCW PRAM.**

Divide and Conquer

The technique is this: Divide the input instance into a number of smaller instances. Solve each of the smaller instances independently and *in parallel*. Combine the solutions of the smaller instances into a solution for the original instance.

PREFIX-SUMS-2

Input: Array $A[1 \ldots n]$ of integers. For simplicity, assume that n is a power of 2.

Output: An array $B[1 \ldots n]$ such that $B[i] = \Sigma_{j=1}^{i-1} A[j]$. Model: CREW PRAM.

```
{
    if (n == 1) return A;
    pardo for 0 ≤ i ≤ 1
        B[in/2 + 1 ... (i + 1)n/2] =PREFIX-SUMS-2(A[in/2 + 1 ... (i + 1)n/2]);
    pardo for 1 ≤ i ≤ n/2
        B[n/2 + i]+ = B[n/2];
    return B;
}
```

Time : $T(n) = T(n/2) + c_1 = O(\log n)$. Cost: $C(n) = 2C(n/2) + c_2 n = O(n \log n)$.

Partitioning

Partitioning is similar to Divide and Conquer except in that here the main work is to be done in dividing the input instance into smaller instances. Combining the results would be comparatively easier.

MIN3

Input: Array $A[1 \ldots n]$ of integers; r the processor advantage.

Output: The minimum integer R in A. Model: COMMON CRCW PRAM

Step 1: /* Divide A into segments of size $2\log\log n$ each and find the minimum in each segment, with one processor per segment */

```
pardo for 1 ≤ i ≤ n/(2 log log n)
{
    B[i] = A[(i − 1).2 log log n + 1];
    for j = 2 to 2 log log n do
        if (b[i] > A[(i − 1)2. log log n + j]) B[i] = A[(i − 1).2 log log n + j];
```

}

Step 2: return MIN2($B[1 \ldots \frac{n}{2 \log \log n}], 2$);

Time taken by a call to MIN3 is clearly $O(\log \log n)$, whereas the cost is $O(n)$. So this is an optimal algorithm for MINIMUM.

Merging is another problem that can be solved using partitioning. But before the partition-based merge can be looked into, some simpler algorithms are to be studied.

Given a sorted array A of items from a linearly ordered set $\mathcal{S}$ and an item x in $\mathcal{S}$, the *rank* of x in A is the number of items less than or equal to x in A.

SEARCH-1

Input: A sorted array A of size n and an item x.

Output: The rank R of x in A. Model: CREW PRAM

{

$R = 0$;

pardo for $1 \leq i \leq n-1$

if $(A[i] \leq x \;\; \&\& \;\; A[i+1] > x) \;\; R = i$;

return R;

}

With $n-1$ processors SEARCH-1(A, x) runs in $O(1)$ time.

SEARCH-2

Input: A sorted array A, an item x, and p the number of processors.

Output: The rank R of x in A. Model: CREW PRAM

{

if $(n \leq p+1)$ return SEARCH-1(A, x);

pardo for $1 \leq i \leq p$

if $(x \geq A[i\lceil \frac{n}{p+1} \rceil]) \;\; B[i] = 0$;

else $B[i] = 1$;

$B[0] = 0$; $B[p+1] = 1$;

pardo for $0 \leq i \leq p$

if $(B[i] == 0 \;\; \&\& \;\; B[i+1] == 1) \;\; R = i+1$;

return $(R-1)\lceil \frac{n}{p+1} \rceil +$ SEARCH-2$(A[(R-1)\lceil \frac{n}{p+1} \rceil + 1 \ldots R\lceil \frac{n}{p+1} \rceil], x)$;

}

SEARCH-2 is a straight generalisation of the ordinary binary search. In the latter we have one processor, and in each step we reduce the search space to half by comparing the item with the mid-point. In the former we have p processors, and in each step we reduce the search space to

$1/(p+1)$ by dividing the array into $p+1$ equal sized segments and comparing the item with every point of division in parallel. Thus, with p processors SEARCH-2 takes $O(\frac{\log n}{\log(p+1)})$ time.

CROSS-RANK-1

Input: Two sorted arrays $A[1 \ldots n]$ and $B[1 \ldots n]$; $n \geq m$.

Output: Arrays $\alpha[1 \ldots n]$ and $\beta[1 \ldots n]$ such that the rank of $A[i]$ in B is $\alpha[i]$ and the rank of $B[i]$ in A is $\beta[i]$. Model: CREW PRAM

{

pardo for $1 \leq i \leq n$

$\alpha[i] =$ SEARCH-2$(B[1 \ldots m], A[i], 1)$;

pardo for $1 \leq i \leq m$

$\beta[i] =$ SEARCH-2$(A[1 \ldots n], B[i], 1)$;

}

With one processor per item, the calls to SEARCH-2 take $O(\log n)$ time. Thus, CROSS-RANK-1 runs in $O(\log n)$ time with $n+m$ processors.

MERGE-1

Input: Two sorted arrays $A[1 \ldots n]$ and $B[1 \ldots m]$; $n \geq m$.

Output: Array $C[1 \ldots m+n]$, the merge of A and B. Model: CREW PRAM

{

CROSS-RANK-1$(A[1 \ldots n], B[1 \ldots m], \alpha[1 \ldots n], \beta[1 \ldots m])$;

pardo for $1 \leq i \leq n$

$C[i + \alpha[i]] = A[i]$;

pardo for $1 \leq i \leq m$

$C[i + \beta[i]] = B[i]$;

}

The pardo-steps each take $O(1)$ time with a linear number of processors. So, MERGE-1 runs in $O(\log n)$ time CROSS-RANK-1 runs in $O(\log n)$ time with $n+m$ processors.

Let us now use partitioning to design an optimal algorithm out of MERGE-1.

MERGE-2

Input: Two sorted arrays $A[1 \ldots n]$ and $B[1 \ldots m]$; $n \geq m$.

Output: Array $C[1 \ldots m+n]$, the merge of A and B. Model: CREW PRAM

{

$L = \lceil \log mn \rceil$; $\quad nBL = \lceil n/L \rceil$; $\quad mBL = \lceil m/L \rceil$;

pardo for $1 \leq i \leq nBL \quad Al[i] = A[(i-1) * L + 1]$;

pardo for $1 \leq i \leq mBL \quad Bl[i] = B[(i-1) * L + 1]$;

CROSS-RANK-1($Al[1 \ldots nBl], Bl[1 \ldots mBl], \alpha l[1 \ldots nBl], \beta l[1 \ldots mBl]$);
pardo for $1 \leq i \leq nBL$
    $\alpha\alpha l[i] = (\alpha[i] - 1) * L +$ SEARCH-2($B[(\alpha[i] - 1) * L + 1 \ldots \alpha[i] * L], Al[i], 1$);
pardo for $1 \leq i \leq mBL$
    $\beta\beta l[i] = (\beta[i] - 1) * L +$ SEARCH-2($A[(\beta[i] - 1) * L + 1 \ldots \beta[i] * L], Bl[i], 1$);
pardo for $1 \leq i \leq nBL$
{
    $AAleft[i] = (i-1) * L + 1; \quad ABleft[i] = \alpha\alpha l[i] + 1$;
    if ($\alpha l[i] == \alpha l[i+1]$) { $AAright[i] = i * L; \quad ABright[i] = \alpha\alpha l[i+1]$; }
    else { $ABright[i] = \alpha[i] * L; \quad AAright[i] = \beta\beta l[\alpha[i] + 1]$; }
}
pardo for $1 \leq i \leq mAL$
{
    $BBleft[i] = (i-1) * L + 1; \quad BAleft[i] = \beta\beta l[i] + 1$;
    if ($\beta l[i] == \beta l[i+1]$) { $BBright[i] = i * L; \quad BAright[i] = \beta\beta l[i+1]$; }
    else { $BAright[i] = \beta[i] * L; \quad BBright[i] = \alpha\alpha l[\beta[i] + 1]$; }
}
pardo for $1 \leq i \leq nBL$
    MERGE-SEQ($A[AAleft[i] \ldots AAright[i]], B[ABleft[i] \ldots ABright[i]],$
        $C[AAleft[i] + ABleft[i] - 1, \ldots AAright[i] + ABright[i]]$);
pardo for $1 \leq i \leq nBL$
    MERGE-SEQ($A[BAleft[i] \ldots BAright[i]], B[BBleft[i] \ldots BBright[i]],$
        $C[BAleft[i] + BBleft[i] - 1, \ldots BAright[i] + BBright[i]]$);
}

In the above, MERGE-SEQ is the sequential merge algorithm.

Intuitively, MERGE-2 works as follows. Arrays A and B are divided into segments of size L each, and from each segment the first element is chosen as a *leader*. Al and Bl are arrays of leaders from A and B respectively. Al and Bl are then cross-ranked. Now each leader knows two leaders on the other side that straddle him— in other words, knows the segment on the other side in which he falls. Each leader then searches (binary or linear) this segment to find his rank in the other side; $\alpha\alpha l[i]$ is the rank of $Al[i]$ in B, and $\beta\beta l[i]$ is the rank of $Bl[i]$ in A.

Let us say, figuratively, the reflection of $Al[i]$ falls between the $\alpha\alpha l[i]$-th and the subsequent items of B. Each leader $Al[i]$ is entrusted the items consecutive in A from $Al[i]$ himself to a leader or a reflection of a leader whichever appears first. Each leader $Al[i]$ also entrusted the items consecutive in B from the reflection of $Al[i]$ to a leader or a reflection of a leader whichever happens first. Similarly for each leader from B also.

Each item entrusted to a leader is larger than every item entrusted to a smaller leader, and each leader is entrusted a maximum of $2L - 1$ items, at most L from own array and at most $L - 1$ from the other. (Prove.) Now the problem is partitioned. Each leader now merges the two pieces entrusted to him sequentially. So, with one processor per leader MERGE-2 runs in $O(L + \log n)$

time; that is, in $O(\log n)$ time with $\frac{n+m}{\log n}$ processors.

Partitioning can give a still faster algorithm for merging.

MERGE-3

Input: Two sorted arrays $A[1 \ldots n]$ and $B[1 \ldots m]$; $n \geq m$.

Output: Array $C[1 \ldots m+n]$, the merge of A and B. Model: CREW PRAM. Suppose $n+m$ processors are available.

Divide A and B into segments of size $\sqrt{n}$ $\sqrt{m}$ each, respectively. Elect the first element from each segment as a leader. Cross rank the leaders in $O(1)$ time; this requires $\sqrt{n}\sqrt{m} < n+m$ processors only. For each leader in A (resp. B), search the segment in B (resp. A) using $\sqrt{m}$ (resp. $\sqrt{n}$) processors. Now the problem is partitioned. Solve each partition (of a size at most $\sqrt{n}+\sqrt{m}$) recursively.

The time taken here is $T(n) = T(\sqrt{n}) + c$ for some constant c. So, $T(n) = O(\log \log n)$. The number of processors is linear.

Using MERGE-3, an optimal $O(\log \log n)$ time CREW PRAM algorithm for merging can be designed; DIY.

Symmetry Breaking

A set of similar active entities conflicting over a set of resources can be modelled as follows. Corresponding to each entity, let there be a vertex, and between two vertices place an edge iff their coresponding entities are in conflict over some resource. We want to allocate the vertices the resources they are seeking in as few phases as possible. A solution is to "colour" the vertices so that no two adjacent vertices get the same colour. Then the colouring would represent a scheduling for resource allocation.

Consider the simplest case: the graph is a linked list. A list can be 2-coloured in $O(\log n)$ time: rank the list and then use the last bit of the rank as colour. Something faster than this seems difficult, if at all possible. But surprisingly, 3-colouring of lists turns out to be much easier.

LIST-3-COLOUR

Input: A linked list represented as an array $s[1 \ldots n]$ of pointers; $s[i]$ is the successor of node i of the list; if nodes i and j are respectively the first and last in the list, then $s[j] = i$.

Output: An array $c[1 \ldots n]$ of "colours"; each $c[i]$ is 0, 1 or 2; if nodes i and j are adjacent in the list, then $c[i] \neq c[j]$. Model: EREW PRAM.

```
{
    pardo for 1 ≤ v ≤ n
        (c[v] = v);
    for k = 1 to log* n + 1 do
        pardo for 1 ≤ v ≤ n
            if (the lsb at which c[v] and c[s[v]] differ is the i-th)
                c[v] = ⟨i, c[v]_i⟩ where c[v]_i is the i-th lsb of c[v];
```

```
    /* Now the list is 6 coloured */
    for k = 3 to 5 do
        pardo for 1 ≤ v ≤ n
            if (c[v] == k)
                c[v] = the least colour that is not present in v's neighbourhood;
    return c;
}
```

Here, $\log^{(1)} n = \log n$; for $k > 1$, $\log^{(k)} n = \log\log^{(k-1)} n$; $\log^* n = \min\{k \mid \log^{(k)} n \leq 1\}$.

Correctness: First let us prove that the second for-loop 6-colours the list. Assume that the list is validly coloured at the beginning of an iteration. Suppose v and w are two nodes such that $w = s[v]$. The new colours for v and w are, respectively, $\langle i, c[v]_i\rangle$ and $\langle j, c[w]_j\rangle$, where $c[v]$ and $c[w]$ differ in the i-th bit, and $c[w]$ and $c[s[w]]$ differ in the j-th bit. If $i \neq j$, then clearly the new colours of v and w are different. If $i = j$, then again the colours are different because $c[v]_i \neq c[w]_i$. Since the loop begins with a valid colouring, it ends with a valid colouring too.

At the beginning of the first iteration, each colour is an $L_0 = \lceil \log n \rceil$ bit number. So, at the end of the first iteration, each colour is an $L_1 = \lceil \log L_0 \rceil + 1 = \lceil \log \lceil \log n \rceil \rceil + 1 = \lceil \log\log n \rceil + 1$ bit number. Note that $\lceil \log \lceil X \rceil \rceil = \lceil \log X \rceil$. (Prove.) Thus, if $n > 2$, $L_1 \leq 2\lceil \log\log n \rceil$. At the end of the second iteration, each colour is an $L_2 = \lceil \log L_1 \rceil + 1 = \lceil \log 2\lceil \log\log n \rceil \rceil + 1 = \lceil \log\log\log n \rceil + 2$ bit number. If $2 \leq \lceil \log\log\log n \rceil$, $L_2 \leq 2\lceil \log\log\log n \rceil$. Continuing like this, at the end of the k-th iteration, each colour is an $L_k \leq 2\lceil \log^{(k+1)} n \rceil$ bit number, provided, $2 \leq \lceil \log^{(k+1)} n \rceil$. So, after $\log^* n - 1$ iterations each colour has at most 4 bits. One further iteration reduces the bits to at most $2 + 1 = 3$. Now the list is 8 coloured. The next iteration gives a 6-colouring, because the binary combinations 110 and 111 cannot arise, the positions being counted 0, 1 and 2.

The 6-colouring is reduced to a 3-colouring by doing away with the excess colours one by one. For example, consider a vertex v of colour 4. Since none of its neighbours are coloured 4 (the colouring is valid), it can find one of the three colours 0, 1 and 2 to colour itself with. None of its neighbours will choose the same colour for itself, because it won't recolour simultaneously with v.

Thus, a linked list of n vertices can be 3-coloured in $O(\log^* n)$ time with n processors on an EREW PRAM.

Pipelining

Pipelining is the following technique: Divide the given task T into a number of subtasks t_1, t_2, $\ldots$ t_k to be solved one after the other. Assign one processor (or a group of processors) for each task. As soon as an instanace of T is processed for t_i, it is passed on to the processor(s) on t_{i+1}, and the processor(s) on t_i take up a new instance that is ready for t_i. Thus, if all t_i's are of equal size, in k units of time the pipeline fills up, and after that one instance is output in every time unit. Thus n tasks can be solved in $n - 1 + k$ units of time; the speed up is $\frac{nk}{n-1+k}$; when $n >> k$, this is almost k, the best possible.

We will see examples of pipelining later.

Sorting

Sorting algorithms fall in two category: comparison based sorting and integer sorting. In the former the only allowed operation on keys is comparison, whereas in the latter it is allowed to analyse the keys. Naturally, the former is more restrictive than the latter. Comaparison based sorting has a sequential lower bound of $\Omega(n \log n)$, while radix sorting of n integers takes only linear (in the total length of the keys) amount of time.

Let us study the parallel complexity of comparison based sorting.

Comparator Networks for Sorting

A comparator module is a device with two inputs and two outputs. The first output of a module is the minimum of its two inputs, and the second the maximum. A network made up of comparator modules such that *(1) an input of a module in the network is connected either to the output of exactly one module or to an input of the network, (2) an output of a module in the network is connected either to the input of exactly one module or to an output of the network, and (3) there is no cyclic path in the network* is called a comparator network. If we assume that a module takes unit time to switch its input to output, the time complexity of a network is clearly its depth, the longest path from an input to an output. The cost of the network is the number of modules in it.

An EREW PRAM can simulate a comparator network in time proportional to its depth, and operations proportional to its cost. (*How?*)

Odd-Even Merge Sorting

First let us consider an algorithm for merging.

ODD-EVEN-MERGE

Input: Two sorted arrays $A[1 \ldots n]$ and $B[1 \ldots n]$. Assume that n is a power of 2.

Output: The merge $C[1 \ldots 2n]$ of the two arrays

Step 0: If $n = 1$, in one comparison merge the arrays. Otherwise, proceed.

Step 1: Form two pairs of arrays:

$(A[1, 3, 5, 7, \ldots, n-1], B[1, 3, 5, 7, \ldots, n-1])$ and $(A[2, 4, 6, 8, \ldots, n], B[2, 4, 6, 8, \ldots, n])$.

That is, take the odd items to one side, and the even to the other. Recursively merge each pair—independently and in parallel. Let $O[1 \ldots n]$ and $E[1 \ldots n]$ be the merges, respectively.

Step 2: $C[1] = O[1]$; $C[2n] = E[n]$;

pardo for $1 \leq i \leq n-1$

$C[2i] = \min(O[i+1], E[i])$;

$C[2i+1] = \max(O[i+1], E[i])$;

Step 3: return C;

As Figure 1 shows, the above algorithm can be easily converted into a comparator network. If $T_M(n,n)$ is the depth and $C_M(n,n)$ is the cost of the network for an $n \times n$ input, then $T_M(n,n) = T_M(n/2,n/2)+1 = T_M(n/4,n/4)+2 = \cdots = T_M(n/2^s,n/2^s)+s = T_M(1,1)+\log n = \log n+1$ and $C_M(n,n) = 2C_M(n/2,n/2)+(n-1) = 4C_M(n/4,n/4)+2n-(1+2) = \cdots = 2^sC_M(n/2^s,n/2^s)+sn-(1+2+4+\cdots+2^{s-1}) = 2^sC_M(n/2^s,n/2^s)+sn-(2^s-1) = nC_M(1,1)+n\log n-(n-1) = n\log n+1$.

ODD-EVEN-MERGE-SORT

Input: An array $A[1\ldots n]$ of items to be sorted. Assume that n is a power of 2.

Output: The sorted sequence

Step 0: If $n = 2$, in one comparison sort the array. Otherwise, proceed.

Step 1: Divide the array into two: ($A[1,\ldots,n/2]$ and $A[n/2+1,\ldots,n]$). Recursively sort each half—independently and in parallel.

Step 2: ODD-EVEN-MERGE the sorted halves into the output.

Step 3: return C;

Figure 2 shows the network design scheme. If $T_S(n)$ is the depth and $C_S(n)$ is the cost of the network for an $n \times n$ input, then

$T_S(n) = T_S(n/2)+T_M(n/2,n/2) = T_S(n/2)+\log n = T_S(n/4)+2\log n-1 = T_S(n/8)+3\log n-(1+2) = \cdots = T_S(n/2^s)+s\log n-(1+2+3+\cdots+(s-1)) = T_S(2)+(\log n)(\log n-1)-(\log -1)(\log n-2)/2 = (\log^2 n+\log n)/2$.

$C_S(n) = 2C_S(n/2)+C_M(n/2,n/2) = 2C_S(n/2)+\frac{n}{2}\log\frac{n}{2}+1 = 4C_S(n/4)+\frac{n}{2}[\log\frac{n}{2}+\log\frac{n}{4}]+(2+1) = 8C_S(n/8)+\frac{n}{2}[\log\frac{n}{2}+\log\frac{n}{4}+\log\frac{n}{8}]+(4+2+1) = 2^sC_S(n/2^s)+\frac{n}{2}[s\log n-(1+2+3+\cdots+s)]+(2^s-1) = (n/2)C_S(2)+(n/2)[(\log n)(\log n-1)/2]+(n/2)-1 = \frac{n\log^2 n-n\log n}{4}+n-1$.

Zero-one principle A comparison based sorting algorithm that works correctly on all sequences of 0's and 1's will work correctly on any sequence of items drawn from a linearly ordered set.

This can be proved as follows: Suppose this is false for some algorithm A. Then A must correctly sort all sequences of 0's and 1's, and yet must fail on some sequence of items drawn from some linearly ordered set. Consider the output of A on such a sequence s. Clearly it will not be sorted. Find the smallest item that is out of place in the output. Tag this item and all smaller items with 0. Tag all larger items with 1. Now run the algorithm again with the items replaced by their tags in s. The input is now a sequence of 0's and 1's. But the output will not be sorted. (*Why ?*) Contradiction!

Correctness of ODD-EVEN-MERGE-SORT If ODD-EVEN-MERGE correctly merges two sorted sequences of 0's and 1's, then ODD-EVEN-MERGE-SORT must correctly sort any sequence of 0's and 1's, and hence by zero-one principle, any sequence of items drawn from a linearly ordered set.

Consider two sorted sequences of 0's and 1's: all 0's before all 1's. Say the first sequence has i 0's, and the second j 0's. Then the number of zeroes on the odd and even sides respectively will be $\lceil i/2 \rceil + \lceil j/2 \rceil$ and $\lfloor i/2 \rfloor + \lfloor j/2 \rfloor$. So, the number of extra zeroes on the odd side can be 0, 1, or 2. After removing the first zero on the odd side (it goes directly to the first output), the number of extra zeroes on the odd side can be -1, 0, or 1. That is, in the pairing off in the last step, atmost one zero will be paired with a one; all the earlier pairs will be $(0, 0)$ and all the later ones $(1, 1)$. The output is indeed sorted, and hence the merge is correct.

Bitonic-Sort-Merge-Sort

A sequence $x_0, \ldots, x_{n-1}$ (of distinct items) is bitonic, if for $0 \leq M < m \leq n-1$, $x_0 < \ldots < x_M > \ldots > x_m < \ldots < x_{n-1} < x_0$. That is, traversing the sequence cyclically (clockwise or anti-clockwise) we find exactly two "tones"—one incresing, and one decreasing. Accordingly, there is one peak (M) and one trough (m). Any sequence obtained by cyclically shifting a bitonic sequence is also bitonic.

Claim: Say, the two sequences $a_0, \ldots, a_{n/2-1}$ and $b_0, \ldots, b_{n/2-1}$ are defined as follows: for $0 \leq i \leq n/2 - 1$, $a_i = \min(x_i, x_{i+1})$ and $b_i = \max(x_i, x_{i+1})$. If $x_0, \ldots, x_{n-1}$ is bitonic, then $a_0, \ldots, a_{n/2-1}$ and $b_0, \ldots, b_{n/2-1}$ are also bitonic. Moreover, all a's are smaller than all b's.

This can be proved as follows. Assume that the trough is at 0; that is $m = 0$. Hence, $x_0 < \ldots < x_M > \ldots > x_{n-1} > x_0$. Wlg, assume that $M \geq n/2$. Otherwise, invert the $>$ relation and cyclically shift the items until x_M becomes the 0th item. Now, consider the subsequences $a_0, \ldots, a_{M-n/2}$ and $b_0, \ldots, b_{M-n/2}$. Clearly, $a_0 = x_0$; $b_0 = x_{n/2}$; and $a_{M-n/2} = x_{M-n/2}$; $b_{M-n/2} = x_M$; So, $a_0 < \ldots < a_{M-n/2}$ and $b_0 < \ldots < b_{M-n/2}$. Next, compare the sequences $x_{M-n/2}, \ldots, x_{n/2-1}, x_{n/2}$ and $x_M, \ldots, x_{n-1}, x_0$. The former is an increasing sequence and the latter a decreasing one. Moreover, $x_{M-n/2} < x_M$ and $x_{n/2} > x_0$. So, the two sequences must cross over at some point. That is, there exists a k, $M - n/2 \leq k \leq n/2 - 1$, such that in the two sequences

$$x_{M-n/2} < \ldots < x_k | < x_{k+1} < \ldots < x_{n/2-1} < x_{n/2}$$
$$\wedge \qquad\qquad\qquad \vee$$
$$x_M > \ldots > x_{k+n/2} | > x_{k+n/2+1} > \ldots > x_{n-1} > x_0$$

where the cross over point is marked with a $|$, the left side has the top sequence elements smaller than the corresponding bottom sequence ones, and the right side has the top sequence elements larger than the corresponding bottom sequence ones. That is, the top sequence elements on the left and the bottom sequence elements on the right are all a's. The others are all b's. Thus we have, $a_0 < \ldots < a_{M-n/2} < \ldots < a_k \Box a_{k+1} > \ldots > a_{n/2-1} > a_{n/2} a_0$, and $b_0 > \ldots > b_{M-n/2} > \ldots > b_k \bullet b_{k+1} < \ldots < b_{n/2-1} < b_{n/2} b_0$. Here $\Box$ and $\bullet$ each may be $>$ or $<$, depending on the input. But whatever they are, as the above inequality shows, $a_0, \ldots, a_{n/2-1}$ and $b_0, \ldots, b_{n/2-1}$ will be bitonic. Also, the peak among a's (either x_k or $x_{k+n/2+1}$) is smaller than the trough among b's (either x_{k+1} or $x_{k+n/2}$).

What we have proved so far is that if we arrange the items of a bitonic sequence along the edges of a circle (which is divided into two semicircles by the trough and its diametrically opposite item), compare every item with its diametrically opposite one, and if necessary swap them so that all losers fall in the same semicircle and the all winners in the other, then both semicircles are bitonic; moreover all the items on the loser side are smaller than all the items on the winner side.

Now assume that the dividing diameter is not along the trough. As can be readily seen, this only causes two opposite sectors (defined by the old and new diamters) to swap their items. Bitonicity is still maintained.

BITONIC-SORT

Input: A bitonic sequence $A[1 \ldots n]$. Assume that n is a power of 2.

Output: The items of A in sorted order.

Step 0: If $n = 1$, in one comparison sort the array. Otherwise, proceed.

Step 1: Compare every item in the first half with the item $n/2$ positions away from it. Move the losers to the first half and the winners to the second.

Step 2: Sort each half independently and in parallel.

Step 3: return A;

As Figure 3 shows, the above algorithm can be easily converted into a comparator network.

A merge algorithm BITONIC-SORT-MERGE is now easy to conceive: Concatenate the first array with the second array inverted. The resultant sequence is bitonic. Sort it.

Using this merging algorithm, in the same way as in ODD-EVEN-MERGE-SORT, a general sorting algorithm BITONIC-SORT-MERGE-SORT can be designed.

If $T_M(n,n)$ is the depth and $C_M(n,n)$ is the cost of the merge network for an $n \times n$ input, then $T_M(n,n) = T_M(n/2,n/2) + 1 = \log n + 1$ and $C_M(n,n) = 2C_M(n/2,n/2) + n = 4C_M(n/4,n/4) + 2n = \cdots = 2^s C_M(n/2^s, n/2^s) + sn = nC_M(1,1) + n\log n = n\log n + n$. If $T_S(n)$ is the depth and $C_S(n)$ is the cost of the sort network for an $n \times n$ input, then

$T_S(n) = T_S(n/2) + T_M(n/2,n/2) = T_S(n/2) + \log n = (\log^2 n + \log n)/2$.

$C_S(n) = 2C_S(n/2) + C_M(n/2,n/2) = 2C_S(n/2) + \frac{n}{2}\log\frac{n}{2} + \frac{n}{2} = 4C_S(n/4) + \frac{n}{2}[\log\frac{n}{2} + \log\frac{n}{4}] + 2n/2 = 8C_S(n/8) + \frac{n}{2}[\log\frac{n}{2} + \log\frac{n}{4} + \log\frac{n}{8}] + 3n/2 = 2^s C_S(n/2^s) + \frac{n}{2}[s\log n - (1+2+3+\cdots+s)] + sn/2 = (n/2)C_S(2) + (n/2)[(\log n)(\log n - 1)/2] + (\log n - 1)(n/2) = \frac{n\log^2 n + n\log n}{4}$.

Since a comparator network can be simulated on an EREW PRAM without any time or cost penalty, we have: n items can be sorted on an EREW PRAM in $O(\log^n)$ time with n processors.

After Batcher invented Odd-Even-Merge sort and Bitonic-Sort-Merge sort networks in 1956, for nearly 25 years it remained an open question whether there are $O(\log n)$ depth optimal comparator networks for sorting, until Ajtai, Komlos and Szemeredi demonstrated one—known as the AKS sorting n/w. The AKS sorting n/w is of scant practical value, because the constant factor involved is, after the latest update, 1600. But it is the only comparator n/w known to match the asymptotic lower bound: recall, finding even the OR of n bits on an EREW PRAM takes $\Omega(\log n)$ time with any number of processors.

A lower bound for sorting

For the sake of studying the comparison complexity of problems we use the parallel comparison model(PCM henceforth): this model is similar to PRAM except in that comparisons are the the

only operations that incur time or cost. A computation on PCM can be visualised as a series of comparison steps; in each step, a set of comparisons are peformed in parallel; based on the results of these comparisons, algorithm decides, free of cost, which are the comparisons to be performed in the next step. PCM is, as a model, stronger than even PRIORITY CRCW PRAM. This is because a concurrent write can be simulated on PCM at no cost: radix sort the processors on the addresses they want to write; all the conflicting processors come together; pick the smallest indexed one from each group of conflicting processors and let it write; this does not involve any comparison between the actual keys, and so incurs no time or cost. A lower bound proved on PCM will be applicable to PRAMs.

Claim: If $c(t,n)$ is the number of comparisons that any algorithms that sorts n items in t comparison steps on PCM must necessarily perform, then $c(t,n) > \frac{tn^{1+1/t}}{e} - tn$.

A proof by induction is as follows: Basis: For t=1, $c(1,n) = n(n-1)/2 > n^2/e - n$. For $t \geq 1$, $c(t,1) = 0 > (t/e - t)$, and for $t \geq 1$, $c(t,2) = 1 > (t2^{1+i/t}/e - 2t)$. Hypothesis: for $t < T$ and $n \leq N$, or $t \leq T$ and $n < N$, $c(t,n) > \frac{tn^{1+1/t}}{e} - tn$.

Let A be some algorithm that sorts N items in T comparison steps on PCM. Consider the first step of A; it should specify a set (independent of the input) of m comaprisons. (Sequential analogy: Bubble sort always begins by comparing the last two items; Insertion sort and Merge sort, the first two items; Quick sort, the pivot and the leftmost non-pivot.) Form a graph G in which the N input positions form the vertices; there is an edge between two vertices iff the items at the corresponding positions are compared in the first step of A. Consider a maximal independent set (MIS) M of G. (An independent set (IS) of a graph is a set of mutually non-adjacent vetices. An MIS M is an IS that cannot be enlarged by adding more vertices. So every vertex not in M must have a neighbour that is in M.) If $\mid M \mid = x$, then there must be at least $N - x$ edges between M and $V - M$. Now, take N distinct items from a linearly ordered set, and fabricate an input for A as follows: place the largest items on the MIS vertices in some random order, and distribute the remaining items over the remaining vertices, again in some random order. When A is run on this input, the first step fails to make a comparison between any of the largest x item. The $N - x$ comparisons between M and $V - M$ performed in this step does not contribute to the sorting of either M or $V - M$ If $c(T,N)$ is the number of comparisons that any algorithms that sorts N items in T comparison steps on PCM must necessarily perform, then we have the following recurrence relation:

$$c(T,N) \geq c(T, N-x) + (N-x) + c(T-1, x)$$

Applying the induction hypothesis,

$$c(T,N) > T\left[\frac{(N-x)^{1+1/T}}{e} - (N-x)\right] + (N-x) + (T-1)\left[\frac{x^{1+1/(T-1)}}{e} - x\right]$$

$$= \frac{T}{e}N^{1+1/k}\left[(1-x/N)^{1+1/T} + (1-1/T)\frac{x^{1+1/(T-1)}}{N^{1+1/T}} + \frac{e}{TN^{1/T}}\right] - TN$$

So, all it remains to show is that the term in square brackets is ≥ 1. Using the Geometric Arithmetic Mean Inequality ($\alpha a + \beta b \geq a^\alpha b^\beta$, where $\alpha + \beta = 1$, $\alpha, \beta, a, b \geq 0$),

$$\left[(1-x/N)^{1+1/T} + (1-1/T)\frac{x^{1+1/(T-1)}}{N^{1+1/T}} + \frac{e}{TN^{1/T}}\right] \geq \left[(1-x/N)^{1+1/T} + \frac{x}{N^{1-1/T^2}}\frac{e^{1/T}}{N^{1/T^2}}\right]$$

Since the increasing sequence $(1+1/t)^t$ converges to e, this is

$$\geq \left[(1-x/N)^{1+1/T} + \frac{x}{N}(1+1/T)\right]$$

By Bernoulli's Inequality ($(1-\alpha)^t \geq 1-\alpha t$ for $t \geq 1$, $\alpha \leq 1$), this is ≥ 1. Hence the claim.

Suppose an algorithm sorts n items in T comparison steps using p processors, where each processor performs at most one comparison in one step. Clearly, $pT \geq c(T,n) > \frac{Tn^{1+1/T}}{e} - Tn$. Hence, $p > \frac{n^{1+1/T}}{e} - n$, and $(p/n+1) > \frac{n^{1/T}}{e}$, and That is, $T = \Omega(\frac{\log n}{\log(p/n+1)})$.

A Simple CREW Merge Sort

Given n item to sort, divide them into two groups of size $n/2$ each. Sort each group in parallel, recursively. Merge the sorted arrays: Use Valiant's merge.

Time: $T(n) = T(n/2) + c\log\log n = T(n/4) + c\log\log(n/2) + c\log\log n = T(n/2^r) + c\log\log(n/2^{r-1}) + \ldots + c\log\log n = T(1) + c\log((\log n)!) = O(\log n \log\log n)$.

Cost: $C(n) = 2T(n/2) + dn = O(n\log n)$.

Here c and d are some constants.

Thus, n items can be sorted on a CREW PRAM in $O(\log n \log\log n)$ time with $\frac{n}{\log\log n}$ processors.

A Pipelined CREW Merge Sort

Let us now study an optimal $O(\log n)$ time CREW sorting algorithm.

Given n items to sort, construct a binary tree such that the items are at the the leaves. For simplicity of dsicussion, assume that n is a power of 2. Suppose each node in the tree has two arrays: a Cache and a Sample. In the beginning the input items are placed in the Cache arrays of the leaves, one item per leaf and hence also per Cache array; all other Cache and Sample arrays are empty. The algorithm proceeds in a number of stages. $3\log n$ stages to be exact. A stage is described below. In the following, we assume that u is an arbitrary node in the tree at some level $k > 0$, and v and w are its children. The leaves are at level 0 and the root at level $\log n$. By $C_t(u)$ (resp. $S_t(u)$) we mean the Cache (resp. Sample) of u after the t-th stage. A stage is performed in parallel for each node u in the tree.

A High Level Description of Stage t of PIPELINED-MERGE-SORT

Step 1: Draw samples from $C_{t-1}(u)$ to form $S_t(u)$, as follows:

If $2k+1 \leq t \leq 3k+1$, pick every 4th item from the right end of $C_{t-1}(u)$ as a sample.

If $t = 3k+2$, pick every 2nd item from the right end of $C_{t-1}(u)$ as a sample.

If $t = 3k+3$, pick every item of $C_{t-1}(u)$ as a sample.

Step 2: Merge $S_t(v)$ and $S_t(w)$ to form $C_t(u)$.

In the third stage of the algorithm, each leaf selects its Cache item into its Sample, and hands it over to the parent; the Cache of the parent will now contain two items in sorted order.

Inductively assume that Caches of v and w (which are at level $k-1$) contain all the items in their respective subtrees in sorted order, at the end of step $3k-3$. Clearly, at the end of the $3k$-th step, v and w will select all their items into their respective samples, which will then be merged into the Cache of u. That is, the Cache of u contains all the items in the subtree rooted aty u in sorted order, at the end of step $3k$. Hence the algorithm will correctly sort the items in $3 \log n$ stages.

Now it remains to show that each stage can be executed in $O(1)$ time with n processors.

Merging with covers In a sorted array $a_1, \ldots, a_n$ of n items, two adjacent items a_i and a_{i+1} are said to define the i-th interval $[a_i, a_{i+1})$; the 0-th and the n-th intervals are $[-\infty, a_1)$ and $[a_n, +\infty]$.

A sorted array C is said to be a c-cover of another sorted array A if at most c items in A fall in any interval of C. Say, A and B are two sorted arrays. Let C be a c-cover of both A and B. In addition suppose C is ranked into both A and B; let r_A,r_B $:\rightarrow \{0, 1, 2, 3, \ldots\}$ be the rank functions.

Suppose there is one processor per item of C. Let x and y be some pair of adjacent items of C. Consider the interval $[x, y)$. Let the processor at x go to A and inform all items with rank $\geq r_A(x)$ and $< r_A(y)$ that they are under the command of x. Treat B similarly.

Now suppose there is one processor per item of A. Let z be some item of A; z knows the C member who is in charge of him; say x. Let the processor at z go to B and sequentially search among the items that are under the command of x; there are at most c of them; so in $O(c)$ time z can find its interval in B. Now A is ranked into B. Similarly, with one processor per item of B, rank B into A. Now A and B are cross ranked, and hence merged.

That is, A and B can be merged in $O(c)$ time with $O(|A| + |B| + |C|)$ processors.

Claim 1 For all $t > 0$, $S_{t-1}(u)$ is a 3-cover of $S_t(u)$.

We will prove a more general statement: For all $t > 0$, h consecutive intervals in $S_{t-1}(u)$ can contain at most $2h + 1$ items of $S_t(u)$.

It is clear that this statement holds if u became full in the $(t-2)$th stage or earlier; that is $3k \leq t - 2$; k is the level of u. (Why ?) So assume that u becomes full at the $(t-1)$-th step or later. Hence $S_t(u)$ contains every fourth item of $C_{t-1}(u)$

Consider h consecutive intervals of $S_{t-1}(u)$. They must correspond to $4h$ consecutive intervals of $C_{t-2}(u)$. But, $C_{t-2}(u)$ is the merge of $S_{t-2}(v)$ and $S_{t-2}(w)$. Say i intervals from $S_{t-2}(v)$ and j intervals from $S_{t-2}(w)$ overlap the $4h$ concerned intervals of $C_{t-2}(u)$. Two cases are possible.
(1) Of the $4h + 1$ consecutive items in $C_{t-2}(u)$ that define the concerned $4h$ intervals the first and the last both come from the same sample array; say, from $S_{t-2}(v)$. Then the contribution of $S_{t-2}(v)$ in the $4h+1$ items is $i+1$. The remaining $(4h+1)-(i+1)$ items come from $S_{t-2}(w)$. The intervals beginning at each of them, and in addition the interval closing before the first of them must all overlap the $4h$ concerned intervals of $C_{t-2}(u)$. Hence, $j = 4h - i + 1$.
(2) Of the $4h + 1$ consecutive items in $C_{t-2}(u)$ that define the concerned $4h$ intervals the first and the last both come from different sample arrays; wlg say, the first from $S_{t-2}(v)$ and the last from $S_{t-2}(w)$. Then the contribution of $S_{t-2}(v)$ in the $4h + 1$ items is i. The remaining $(4h + 1) - i)$ items, of which the last is also part, come from $S_{t-2}(w)$. The intervals closing before each of them must all overlap the $4h$ concerned intervals of $C_{t-2}(u)$. Hence, $j = 4h - i + 1$. That is, $i + j = 4h + 1$.

So, $i + j = 4h + 1$ either way. Inductively assume that the i intervals from $S_{t-2}(v)$ and j intervals from $S_{t-2}(w)$ contain at most $2i + 1$ and $2j + 1$ items resp. of $S_{t-1}(v)$ and $S_{t-1}(w)$, and hence together at most $2(i+j)+2 = 8h+4$ items of $C_{t-1}(u)$. That is, the concerned $4h$ consecutive intervals of $C_{t-2}(u)$, contain at most $8h + 4$ items of $C_{t-1}(u)$. But $S_t(u)$ contains every fourth item of $C_{t-1}(u)$. So the concerned $4h$ consecutive intervals of $C_{t-2}(u)$, contain at most $2h + 1$ items of $S_t(u)$. Or, the h consecutive intervals of $S_{t-1}(u)$ that we began with contain at most $2h + 1$ items of $S_t(u)$.

Now let $h = 1$, and we have the claim.

Claim 2 For all $t > 0$, $C_{t-1}(u)$ is a 3-cover of both $S_t(v)$ and $S_t(w)$.

$C_{t-1}(u)$ is the merge of $S_{t-1}(v)$ and $S_{t-1}(w)$, which are resp. 3-covers of $S_t(v)$ and $S_t(w)$. Consider any interval of $C_{t-1}(u)$; it must be a subinterval (proper or otherwise) of some interval each in $S_{t-1}(v)$ and $S_{t-1}(w)$; since those super-intervals themselves contain at most 3 items of the resp. sample arrays at the next stage, so must the subinterval being considered. Hence the claim.

Now we can discuss the stage in detail.

Stage t of PIPELINED-MERGE-SORT

Step 1.1: /* Assume that $C_{t-2}(u) \rightarrow C_{t-1}(u)$. That is, $C_{t-2}(u)$ is ranked into $C_{t-1}(u)$. */

Draw samples from $C_{t-1}(u)$ to form $S_t(u)$, as follows:

If $2k + 1 \leq t \leq 3k + 1$, pick every 4th item from the right end of $C_{t-1}(u)$ as a sample.

If $t = 3k + 2$, pick every 2nd item from the right end of $C_{t-1}(u)$ as a sample.

If $t = 3k + 3$, pick every item of $C_{t-1}(u)$ as a sample.

With one processor per Cache item per node, drawing of samples can be done in $O(1)$ time.

Step 1.2: Rank $S_{t-1}(u)$ into $S_t(u)$: Every x in $S_{t-1}(u)$ knows its rank in $C_{t-2}(u)$ because that is where it was drawn out as a sample. But $C_{t-2}(u) \rightarrow C_{t-1}(u)$. Hence, $x \rightarrow C_{t-1}(u)$. Again, $S_t(u)$ is drawn from $C_{t-1}(u)$; so, $x \rightarrow S_t(u)$.

With one processor per Sample item per node, the old Sample arrays can be ranked into the new ones.

Step 1.2: Find $C_{t-1}(u) \rightarrow S_t(v)$ and $C_{t-1}(u) \rightarrow S_t(w)$: $C_{t-1}(u)$ is the merge of $S_{t-1}(v)$ and $S_{t-1}(w)$. So, every item x in $C_{t-1}(u)$ must know its rank in both $S_{t-1}(v)$ and $S_{t-1}(w)$. Since the old Sample array is ranked into, and is a 3-covers of, the new sample array x now knows its rank in both $S_t(v)$ and $S_t(w)$ upto an error of 3.

With one processor per Cache item per node, the $(t-1)$-Caches of all nodes can be ranked into the t-Samples of the respective children in at most $3 + 3 = 6$ steps.

Step 2.1: Merge $S_t(v)$ and $S_t(w)$ to form $C_t(u)$: The old Cache of the parent is a 3-cover of, and is ranked into, the new Samples of the children. Hence, with one processor per Cache item per node, and one processor per Sample item per node, the new Samples of all the children can be merged into the Cache of the respective parents in $O(1)$ time.

Step 2.2: Find $C_{t-1}(u) \rightarrow C_t(u)$: Already we know $C_{t-1}(u) \rightarrow S_t(v)$ and $C_{t-1}(u) \rightarrow S_t(w)$. $C_t(u)$ is the merge of $S_t(v)$ and $S_t(w)$. For each x in $C_t(u)$ add its rank in $S_t(v)$ and $S_t(w)$ to get its rank in $C_t(u)$.

With one processor per Cache item per node, this will take $O(1)$ time.

It is now clear that the algorithm runs in $O(1)$ time, if we have one processor per Cache/Sample item per node. But in stage t a node u at level k need to get processors only if $2k+1 \leq t \leq 3k+3$.

Consider the following table. The k-th row (from bottom), t-th column (from left) entry in it gives it the sizes of Sample/Cache arrays at nodes of level k at the end of t-th stage.

										0/2	0/4	1/8	2/16	4/32
							0/2	0/4	1/8	2/16	4/16	8/16	16/16	
						0/2	0/4	1/8	2/8	4/8	8/8	8/8	8/8	8/8
				0/2	0/4	1/4	2/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
		0/2	0/2	1/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
0/1	0/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1

Try out more rows at the top. The following can be readily seen.

At stages numbered $3r$, for some integer r, the Cache size at a (live) parent is 1/4 the Cache size at one of its children, and hence 1/8 the Cache size at both the children put together. Hence the the total number of Cache item processors needed in this stage is at most $n + n/8 + n/64 + \ldots = 8n/7 = O(n)$

At stages numbered $3r+1$, for some integer r, the Cache size at a (live) parent is 1/4 the Cache size at one of its children if the children are not dead, 1/2 the Cache size at one of its children otherwise. Hence the the total number of Cache item processors needed in this stage is at most $n + n/4 + n/32 + n/256 + \ldots = 9n/7 = O(n)$

At stages numbered $3r+2$, for some integer r, the Cache size at a (live) parent is 1/4 the Cache size at one of its children if the children are not dead, the same as the Cache size at one of its children otherwise. Hence the the total number of Cache item processors needed in this stage is at most $n + n/2 + n/16 + n/128 + \ldots = 11n/7 = O(n)$

Similarly, the total number of Sample item processors also can be shown to be $O(n)$ in any stage. (DIY)

(Exercise: Formally prove the above.)

So, n items can be sorted in $O(\log n)$ time with n processors on a CREW PRAM.

Graph Algorithms

A graph $G = (V, E)$ consists of a finite nonempty set V of vertices and a set $E \subseteq V \times V$ of unordered pairs of distinct vertices. Each $e \in E$ is an edge of G. We shall denote the number of edges $|E|$ by m, the number of vertices $|V|$ by n. Two vertices $u, v \in V$ are said to be adjacent iff $\{u, v\} \in E$; in this case, we also say that u and v are neighbours of each other. We denote the set of neighbours of v by $N(v)$. When $e = \{u, v\}$, we say that e is incident with u and v. The number of neighbours of a vertex is its degree. The maximum vertex degree of a graph G is denoted by $\Delta(G)$; whenever there is only one graph G under consideration we shall use Δ for $\Delta(G)$. A graph in which every vertex is of degree d, is a d-regular graph. A 3-regular graph is also called a cubic graph. A graph that has a maximum vertex degree of 3 is called a subcubic graph.

In a directed graph every edge is an ordered pair; if $(u, v) \in E$ we say that v is an out-neighbour of u and u is an in-neighbour of v. The number of out-neighbours of a vertex is its out-degree, and the number of in-neighbours is its in-degree.

$G' = (V', E')$ is said to be a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$. If $E' = \{\{u, v\} \mid \{u, v\} \in E \text{ and } u, v \in V'\}$ then, $G' = G[V']$ is the subgraph induced by V'. We use $G - U$ to denote $G[V - U]$ for $U \subseteq V$, $G - v$ to denote $G - \{v\}$ for $v \in V$, and $G - e$ to denote $G' = (V, E - \{e\})$ for $e \in E$.

A graph is called complete if for every pair (u, v) of distinct vertices, u and v are adjacent. A complete graph on n vertices is called an n-clique. A maximum clique of G is a largest cardinality complete subgraph of G; its cardinality is the clique number $\omega(G)$ of G.

An independent set I of G is a subset of V such that for every $u, v \in I$, $\{u, v\}$ is not in E. In a maximal independent set (MIS for short) I, in addition the following condition holds: for each $v \in V$ either $v \in I$ or v has a neighbour in I.

Vertex colouring

In the vertex colouring problem on graphs, we seek to assign colours to the vertices so that no two adjacent vertices get the same colour. The minimum number of colours needed to colour G is the chromatic number of G and is denoted by $\chi(G)$. Clearly, $\chi(G)$ is not smaller than $\omega(G)$, the clique number. If, for every $V' \subseteq V$, $\chi(G[V']) = \omega(G[V'])$, then $G = (V, E)$ is called a perfect graph.

Finding the chromatic number of a 4-degree planar graph, let alone an arbitrary graph, is NP complete. Minimally vertex colouring an arbitrary graph is, thus, computationally difficult.

Hence, to colour graphs efficiently we should either restrict our attention to certain types of graphs that are easy to colour minimally, or settle for a possibly non-minimal colouring.

In the sequential setting, both these approaches have given efficient algorithms. A simple $O(m)$ time search can 2-colour a bipartite graph. The proof of 4-colour theorem gives a polynomial time algorithm for 4-colouring a planar graph; this algorithm has large constant factors. But planar graphs can be 5-coloured in $O(n)$ time. Several special classes of perfect graphs have efficient algorithms for minimal colouring, though no polynomial time algorithm for recognising general perfect graphs is known. For example an interval graph can be recognised and minimally coloured in $O(n)$ time. A graph can be easily $(\Delta + 1)$-coloured in $O(m)$ time. Brooks showed that any

connected graph is Δ-colourable if it is neither a complete graph on $\Delta + 1$ vertices nor a circuit of odd length. A Δ-colouring has often been called a *Brooks' colouring* and a Δ-colourable graph a *Brooks' graph*. A Brooks' graph can be Δ-coloured in linear time.

An Optimal Algorithm for 3-Colouring of Linked Lists

The symmetry breaking algorithm we saw earlier 3-colours a linked list in $O(\log^* n)$ time with n processors on an EREW PRAM. Clearly, this is not optimal.

The following is an algorithm that 3-colours a list in $O(k \log^{(k)} n)$ time with $O(n/\log^{(k)} n)$ processors on an EREW PRAM.

OPTIMAL-3-COLOUR-LINKED-LISTS

Input: A linked list of n nodes in an array

Output: A 3-colouring σ of L.

Step 1: Find a $log^{(k)}n$ colouring of L using the first k steps of the symmetry breaking algorithm. Let these colours be in the range $[4 \ldots \log^{(k)} n + 3]$. With n processors this would take $O(k)$ time. With $n/\log^{(k)} n$ processors this would take $O(k \log^{(k)} n)$ time.

Step 2: Arrange the node in a 2-D array of $n/\log^{(k)} n$ columns and $\log^{(k)} n$ rows. (Rows and columns numbered starting at 0.) Assign one processor per column. Sort the vertices in each column. Each processor has to sort $\log^{(k)} n$ items in the range $[4 \ldots \log^{(k)} n + 3]$. Use radix sort; takes only linear time. Note that the sorting does not disrupt the adjacency relations in the list.

Step 3: The nodes in each row can be classified into two: the *intra-row* nodes and the *inter-row* nodes. A node is inter-row if either its incoming edge is from another row or its outgoing edge is to another row. A node is intra-row iff it is not inter-row. Process the inter-row nodes of the 0-th row like this:

(1) Consider the inter-row nodes that have incoming edges from another row; they form an independent set (Why?); give each of them the smallest colour absent in their neighbourhood. (2) Now consider the inter-row nodes that have outgoing edges to another row and were not considered in (1) above; they also form an independent set; give each of them the smallest colour absent in their neighbourhood.

Each inter-row node of row-0 is now coloured 1, 2 or 3. Repeat the same for each of the rows. Total time: $O(\log^{(k)} n)$.

Step 4: Now to colour the intra-row nodes. Get all the processors back to the 0-th row. For each of the processors, do the following:

```
for 1 ≤ i ≤ 2 log^(k) n − 1 do
    if (the node is intra-row and row + colour - 3 == i)
        recolour the node with the smallest colour absent in its neighbourhood
        and step on to the next row in the same column
```

In the above, *row* is the present row the processor is in, and *colour* is the colour of the node on which the processor presently sits. Since the last row number is $\log^{(k)} n - 1$ and the last

colour is $\log^{(k)} n + 3$, all processors will fall through their columns in $2\log^{(k)} n - 1$ steps.

Note that two intra-row nodes from different rows cannot be adjacent. Also, in each step i, in each row only nodes of the same colour (i-*row*+3) recolour themselves. Hence the no two adjacent nodes recolour themselves simultaneously. Every node gets a chance to recolour itself. So the algorithm works correctly.

(Δ+1)-Colouring of Bounded Degree Graphs

FAST-(Δ+1)-COLOURING

Input: A graph $G = (V, E)$ in adjacency list representation. $\Delta = O(1)$. Model CREW PRAM. Assume that $n + m$ processors are available.

Output: A (Δ+1)-colouring $\sigma : V \to \{1, \ldots, \Delta + 1\}$ of G.

Step 1: Let each vertex $v \in V$ scan its adjacency list, and find its largest larger neighbour; let this vertex be designated $p(v)$. If a vertex $v \in V$ does not have a larger neighbour, then $p(v)$ is undefined. But the subgraph defined by the function $p : V \to V$ is a tree; because, $p(v) > v$.

Step 2: If $p(v)$ is undefined, and v is not $p(w)$ for any w then v is isolated in the subgraph defined by p. For all isolated vertices v, let $p(v)$ be the first of v's neighbours. The subgraph defined by p is still a tree.

Step 3: Remove the subgraph defined by p (let me call it L_p) from G; note that we are removing a set of edges from G; the vertex set of L_p as well as $G - L_p$ is V; the maximum vertex degree of $G - L_p$ is at most $\Delta - 1$. Recursively Δ-colour $G - L_p$. 3-colour L_p using the symmetry breaking algorithm. Each vertex of G now acquires two colours, one from the Δ-colouring of $G - L_p$, and another from the 3-colouring of L_p. So G is now 3Δ coloured.

Step 4: Consider the colours $\delta + 2 \ldots 3\Delta$ one by one; for each colour, recolour each vertex of that colour with the smallest colour absent in its neighbourhood.

When $\Delta = O(1)$ the above algorithm runs in $O(\log^* n)$ time with n processors on a CREW PRAM.

OPTIMAL-(Δ+1)-COLOURING

Input: A graph $G = (V, E)$ in adjacency list representation. $\Delta = O(1)$. Model EREW PRAM. Assume that $n/\log^{(k)} n$ processors are available.

Output: A (Δ+1)-colouring $\sigma : V \to \{1, \ldots, \Delta + 1\}$ of G.

Step 1: for $v \in V$ in parallel do

for $1 \leq i \leq \Delta$ do

(1) if $p(v)$ is undefined, make a proposal to the i-th neighbour w_i of v

(2) if $s(v)$ is undefined, check all the proposals v has received and choose one of them as $s(v)$

(3) if v's proposal to w_i has been accepted (if it made any), set $p(v) = w_i$

p (predecessor) and s (successor) together define a 2-degree graph: a collection of chains and cycles.

Step 2: Remove the subgraph defined by p (let me call it L_p) from G. Recursively $(\Delta+1)$-colour $G - L_p$. 3-colour L_p using OPTIMAL-3-COLOUR-LINKED-LISTS. Each vertex of G now acquires two colours, one from the Δ-colouring of $G - L_p$, and another from the 3-colouring of L_p. So G is now $3(\Delta + 1)$ coloured.

Step 3: Consider the colours $\delta + 2 \ldots 3(\Delta + 1)$ one by one; for each colour, recolour each vertex of that colour with the smallest colour absent in its neighbourhood.

A vertex fails to find a predecessor only if its proposal is rejected by all it neighbours. Then each of them must have chosen some other vertex as its predecessor. Hence in $G - L_p$ all of them have a degree of at most $\Delta - 1$. So a vertex can go without both predecessor and successor for at most $\Delta - 1$ levels of recursion; after that it is the sole neighbour of all its neighbours; they must necessarily take its proposals after that; so all the edges in the graph will be removed in or before the $(2\Delta - 1)$-th level of recursion. That is, when $\Delta = O(1)$ the above algorithm runs in $O(\log^{(k)} n)$ time with $n/\log^{(k)} n$ processors on an EREW PRAM. (How does the time complexity of this algorithm depend on Δ ?)

Brooks' Colouring of Bounded Degree Graphs

Let us study an an $O(\Delta^2 \log \Delta \log n)$ time algorithm for Δ-colouring a Brooks' graph, with $n/\log n$ processors, on an EREW PRAM. In particular, this algorithm 3-colours a 3-degree Brooks' graph in $O(\log n)$ time, with $n/\log n$ processors, on an EREW PRAM.

For the problem of Δ-colouring, a colour c is said to be *feasible* at v if v does not have a neighbour coloured c. An uncoloured vertex in a partially coloured graph is said to be *at impasse* if it has no feasible colour in $\{1, \ldots, \Delta\}$. If v is a vertex at impasse, then its neighbour of colour i will be denoted by $v_i (i = 1, \ldots, \Delta)$.

An α-β component in G is a component of the subgraph induced by vertices coloured α or β. Note that interchanging colours α and β in an α-β component does not affect the validity of the colouring. So, for a vertex v at impasse if v_α and v_β belong to different α-β components interchanging colours in one of them will resolve the impasse.

We define a construct called *fork* as follows: given a partially coloured Brooks' graph $G = (V, E)$, a fork at $v \in V$ is a minimal simple path F of at least two vertices with v as an endpoint, so that the other endpoint $e(F)$ of F either (i) has a degree of at most $\Delta - 1$ in G, or (ii) has a degree of Δ in G, and has two neighbours, both of which are not in F, and are coloured the same. If v is at impasse, and the neighbour of v in F is the α-coloured neighbour v_α of v in G, then we call F an α-fork.

For each vertex w in F, $w \neq e(F)$, let the successor of w, be the farther from v of its neighbours in F.

If v is at impasse, then using F the impasse can be resolved as follows: Recolour each $x \in F$, $x \neq e(F)$, with the colour of its successor in F. Uncolour $e(F)$. Give $e(F)$ one of the colours

missing in its neighbourhood; note that, by definition, such a colour should exist.

Similarly, if v is coloured, then using F we can recolour v: Recolour each $x \in F$, $x \neq e(F)$, with the colour of its successor in F. Uncolour $e(F)$. Give $e(F)$ one of the colours missing in its neighbourhood.

First, we present a procedure that will be subsequently used in the algorithm.

Procedure MAXIMAL_SET(Δ, α, β)

Input: A Δ-regular graph $G = (V, E)$, in which every α-β component is a simple path and for every vertex v at impasse v_α and v_β are end points of α-β chains.

Output: A maximal set of α-β components in G, such that no two members in the set *touch* the same impasse vertex; a component Γ touches a vertex v if at least one vertex in Γ is adjacent to v.

Step 1: Form a graph H in which each vertex corresponds to an α-β component of G. Place an edge between two vertices of H if and only if the α-β components corresponding to them touch the same impasse vertex. Since an α-β component can touch at most two impasse vertices, one at each end, the maximum vertex degree of H is 2. Remove all isolated vertices from H.

Step 2: Find the connected components of H and identify each as a chain or a cycle. From each cycle remove a vertex. Let the resulting graph be H'. List rank each component of H'. For every odd ranked vertex of H' interchange colours α and β in the corresponding α-β component of G.

An algorithm for 3-colouring of 3-degree graphs is now presented.

Procedure 3_Colour_Cubic_Graphs

Input: A cubic graph $G = (V, E)$ in adjacency list representation.

Output: A 3-colouring $\sigma : V \to \{1, 2, 3\}$ of G.

Step 1: Find an MIS M of G, and for each $v \in M$ let $\sigma(v) = 3$. The graph $G - M$ has a maximum degree of 2; that is, $G - M$ consists of disjoint chains and cycles. From every odd cycle of $G - M$ elect a representative into a set I. Clearly, $G - M - I$ is 2-colourable; 2-colour it. Each vertex of I is at impasse, and is left uncoloured.

REMARK: For each vertex $v \in V$, we keep two flags $P(v)$ and $Q(v)$. Each flag can take on values "up" and "down". Initially all flags are down. Every flag that is up, will be associated with an impasse vertex; that is, every flag that is up will hold a pointer to an impasse vertex. As soon as the impasse is resolved at a vertex, all its associated flags will be brought down. We shall use the following procedure extensively.

Procedure RESOLVE

```
begin
    For v ∈ I do in parallel
        if v has at least one colour missing in its neighbourhood then
            give v the minimum feasible colour, remove it from I,
            and bring down every flag associated with v
end.
```

In the sequel, a vertex with its P-flag up will be called a P-vertex, and a component in the subgraph induced by P-vertices a P-component; and, similarly for flag Q.

Step 2: For each odd cycle C of $G - M$, associate the P-flags of all coloured vertices in C with the impasse vertex in C, and put up all of them.
REMARK: Every odd cycle of $G - M$ provides a path between v_1 and v_2, if v is the impasse vertex contained in it. Moreover, each vertex on this path is coloured 1 or 2 and its neighbour outside the path is coloured 3. Thus, v_1 and v_2 belong to the same $1-2$ component, which is a simple path with v_1 and v_2 as its end points.

Step 3: Recolour all colour-1 vertices with colour 2 where feasible. Recolour all colour-3 vertices with colour 2 where feasible. Call RESOLVE.
REMARK: Every $1-3$ component of G is a now simple path. Thus, each of v_3 and v_1 is an end point of a $1-3$ chain. That is every impasse vertex has exactly two (not necessarily distinct) $1-3$ chains going out of it.

Step 4: Call MAXIMAL_SET$(3, 1, 3)$. Call RESOLVE.
REMARK: Impasse is resolved for a vertex, if colour was changed in any one of the two $1-3$ components emanating out of it. For each $v \in I$, now we have a simple $1-3$ path in G with v_1 and v_3 as its end points. But, note that now it is not necessary for v_1 and v_2 to be in the same $1-2$ component, let alone a $1-2$ path.

Step 5: For each impasse vertex v, associate all Q-flags in the $1-3$ path from v_1 to v_3 with v, and put up all of them.

Step 6: Recolour all colour-3 vertices with colour 1 where feasible. Recolour all colour-2 vertices with colour 1 where feasible. Call RESOLVE. Call MAXIMAL_SET$(3, 3, 2)$. Call RESOLVE.
REMARK: For each $v \in I$, now we have a simple $2-3$ path in G with v_2 and v_3 as its end points. But, v_1 and v_2 may not be in the same $1-2$ component. And v_1 and v_3 may not be in the same $1-3$ component. All P-components and Q-components of G are simple chains, by construction. Also, no vertex can have both its flags up, unless it is a common endpoint of a P-chain and a Q-chain. So, every impasse vertex v has a P-path $L_P(v)$ and a Q-path $L_Q(v)$ of its own.

Step 7: For each impasse vertex v, if $L_P(v)$ has at least one vertex with a non-P-neighbour of colour 1 or 2, find either a 1-fork or a 2-fork at v that involves only vertices from $L_P(v)$. Use this fork to resolve the impasse at v. Call RESOLVE.
REMARK: When $L_P(v)$ has at least one vertex with a non-P-neighbour of colour 1 or 2, the two endpoints of $L_P(v)$ being coloured 1 and 2, a fork at v can be found from $L_P(v)$ in at least one direction. If v is still at impasse after this step, $L_P(v)$ is a maximal $1-2$ component of G.

Step 8: For each impasse vertex v, if $Q_P(v)$ has at least one vertex with a non-Q-neighbour of colour 1 or 3, find either a 1-fork or a 3-fork at v that involves only vertices from $L_Q(v)$. Use this fork to resolve the impasse at v.
REMARK: So, we are left with only the case where v_i and v_j are the end points of a simple i-j path for $1 \leq i < j \leq 3$.

Step 9: For $v \in I$, let $v_1 = x$ $v_2 = y$ and $v_3 = z$. If y is adjacent to both x and z, then x and z are not adjacent because G has no 4-cliques, and y is the only neighbour of colour 2 for both x and z; let $\sigma(x) = \sigma(z) = 2$, $\sigma(y) := 1$, and $\sigma(v) := 3$. If y is adjacent to at most one of x and z, then interchange colours 1 and 3 in $L_Q(v)$. Now, clearly, $L_P(v)$ can provide a fork at v in at least one direction. Use this fork to resolve the impasse at v. Call RESOLVE.

The procedure 3-Colour-Cubic-Graphs 3-colours a 4-clique free cubic graph in $O(\log n)$ time and $O(n)$ operations on an EREW PRAM.

Proof: Correctness of the procedure is obvious from the remarks following individual steps.

In Step 1 of the procedure, we have to find an MIS of G. We do this by first finding a $(\Delta+1)$-colouring $C : V \to \{1, \ldots, \Delta + 1\}$ of G using the optimal algorithm for $(\Delta+1)$-colouring a bounded degree graph, and then, for $i := 1$ to $\Delta + 1$, in turn adding $w \in V$ to M, if $C(w) = i$ and no neighbour of w is already in M. The time taken is $O(\Delta^2 \log \Delta \log^{(k)} n)$ (for any fixed $k \geq 1$) with $\frac{n}{\log^{(k)} n}$ processors on an EREW PRAM. That is, with $\frac{n}{\log n}$ processors ($k = 1$), the time taken is $O(\Delta^2 \log \Delta \log n)$. Besides, there are $O(1)$ instances of, all vertices having to scan their respective neighbourhoods in parallel; these can be solved in $O(\Delta)$ time and $O(n\Delta)$ operations on a CREW PRAM. The rest of the procedure is dominated by a constant number of invocations to the list ranking algorithm, and hence can be solved optimally in $O(\log n)$ time on an EREW PRAM. Hence the claim on resource requirements holds for a CREW PRAM.

For two adjacent vertices u and w, let $[u, w]$ be the entry for the edge (u, w) in u's edge list. We assume that $[u, w]$ and $[w, u]$ have a pointer to each other. These pointers, if not given as a part of the input, can be easily created in $O(1)$ time using $O(n^2)$ space and n processors on an EREW PRAM—form an adjacency matrix in which each "non-zero" item is a pointer to an edge list entry; no initialisation is needed as we will never look at a "zero". It is easy to see that with this structure available, all vertices can scan their neighbourhoods in parallel, in $O(\Delta)$ time, without read conflicts. Hence, the algorithm can be run on an EREW PRAM also, with the same resource bounds.

Note that Procedure RESOLVE can be implemented on an EREW PRAM in $O(\log n)$ time. Since, the P-vertices and the Q-vertices associated with an impasse vertex form a list each, an invocation to list ranking is sufficient to bring down the flags in all of them. □

Since for any subcubic graph on n vertices, a cubic graph on $O(n)$ vertices of which the former is a subgraph, can be constructed in constant time with $O(n)$ processors, a 4-clique free 3-degree graph can be 3-coloured in $O(\log n)$ time and $O(n)$ operations on an EREW PRAM.
Now, an algorithm for Brooks' colouring a general graph is presented.

Procedure Δ-Colour-Regular-Graphs

Input: A $(\Delta+1)$-clique-free regular graph $G = (V, E)$ in adjacency list representation.

Output: A Δ-colouring $\sigma : V \to \{1, \ldots, \Delta\}$ of G.

Step 0: If $\Delta \leq 3$ use the procedure 3-Colour-Cubic-Graphs.

Step 1: Find an MIS M of G, and for each $v \in M$ let $\sigma(v) = \Delta$. The graph $G - M$ has a maximum degree of $(\Delta - 1)$. From every Δ-clique of $G - M$ elect a representative into a set I.

As $G - M - I$ does not contain any Δ-clique, it is $(\Delta-1)$-colourable. Recursively $(\Delta-1)$-colour $G - M - I$. Each vertex of I is at impasse, and is left uncoloured. For each $v \in I$, select the minimum $\beta \in \{1, \ldots, \Delta - 1\}$ such that v_Δ and v_β are not adjacent. Swap colours between v_1 and v_β.

REMARK: For each $v \in I$, $G[v, v_1, ..., v_{\Delta-1}]$ is a Δ-clique of G. Also, v_1 and v_Δ are not adjacent. Observe that, v_Δ is not adjacent to all of $\{v_1, \ldots, v_{\Delta-1}\}$ because, v_Δ is adjacent to v and G does not contain a $(\Delta+1)$-clique.

Step 2: Recolour, first all colour-1 vertices, and then all colour-Δ vertices, with a colour other than 1 and Δ where feasible. Call RESOLVE.

REMARK: Now, every $1-\Delta$ component of G is a chain or a cycle. Also, for each $v \in I$, v_1 and v_Δ are end points of $1-\Delta$ chains.

Step 3: Call MAXIMAL_SET$(\Delta, 1, \Delta)$. Call RESOLVE.

Step 4: For each impasse vertex v, associate all P-flags in the $1-\Delta$ path from v_1 to v_Δ with v, and put up all of them.

Step 5: Repeat steps 2 and 3, with colour 2 substituted for colour 1.

REMARK: Now v_2 and v_Δ are the end points of the same 2-Δ path, for every $v \in I$. But v_1 and v_Δ need not even be in the same $1-\Delta$ component. All P-components of G are simple chains, by construction. So, every impasse vertex v has an exclusive P-path $L_P(v)$.

Step 6: For each impasse vertex v, if possible, find a fork at v that involves only vertices from $L_P(v)$, and use this fork to diffuse the impasse at v. Call RESOLVE.

REMARK: If $L_P(v)$ does not provide a fork as required, then no vertex w of $L_P(v)$ has two non-P-neighbours of the same colour. Also, both P-neighbours of w, when there are two, must be coloured the same. Since v_1 and v_Δ are the end points of $L_P(v)$, this would mean that $L_P(v)$ is a maximal $1-\Delta$ component of G. That is, v_1 and v_Δ are the end points of the same $1-\Delta$ path $S_1(v) = L_P(v)$, and v_2 and v_Δ are the end points of the same $2-\Delta$ path $S_2(v)$. The only vertex common to $S_1(v)$ and $S_2(v)$ is v_Δ. Since v_1 and v_Δ are not adjacent, $S_1(v)$ does not degenerate into a single edge. But, $S_2(v)$ may be a single edge.

Step 7: For each impasse vertex v, interchange colours 2 and Δ in $S_2(v)$. No neighbour of v_1 is now of colour 2, because, prior to this step, v_1 and v_Δ were not adjacent. Recolour v_1 with 2, and the impasse at v is resolved.

The procedure Δ-Colour-Regular-Graphs Δ-colours a regular Brooks' graph in $O(\Delta^2 \log \Delta \log n)$ time with $\frac{n}{\log n}$ processors on an EREW PRAM.

Proof: With $\frac{n}{\log n}$ processors on an EREW PRAM, all steps of Procedure Δ-Colour-Regular-Graphs, except the $(\Delta+1)$-colouring and the recursive call in Step 1, can be implemented in $O(\Delta \log n)$ time, and that the $(\Delta+1)$-colouring can be done in $O(\Delta^2 \log \Delta \log n)$ time. Note that $G - M$ is properly coloured and does not have a vertex of colour $\Delta + 1$. That is, the restriction of the colouring C to $V - M$ is a valid Δ-colouring of $G - M$. Hence, $G - M - I$ is already Δ-coloured

when it is passed on to the next lower level of recursion and need not be coloured again. In other words, we need only one invocation of the colouring algorithm, at the beginning of the procedure; the lower levels of recursion can make use of the same colouring. □

A Brooks' graph can be Δ-coloured in $O(\Delta^2 \log \Delta \log n)$ time with $\frac{n}{\log n}$ processors on an EREW PRAM.

Finding Connected Components in undirected graphs

Let us study two algorithms for finding connected components in undirected graphs, running in $O(\log^{1+1} n)$ and $O(\log^{1+1/2} n)$ time respectively with $n + m$ processors on CREW PRAM.

CONNECTED-COMPONENTS-1

Input: A graph $G = (V, E)$ on n vertices and m edges given in adjacency list representation. Assume that twin pointers are available; that is, the two entries corresponding to an edge $\{u, v\}$, one in u's adj-list and the other in v's adj-list, ponit to each other. A total of $n + 2m$ processors are available; one each for vertices and adj-list entries. Model: CREW

Output: A function $p : V \to V$ so that, for all $u, v \in V$, $p(u) = p(v)$ iff u and v belong to the same connected component (CC) of G.

Algorithm:

Let $G_0 = (V_0, E_0)$ be G.

for $i = 0$ to $\log n - 1$ do

execute PHASE(i) with $G_i = (V_i, E_i)$ as the input.

PHASE(i)

Step 0: For all $v \in V_i$ let $p(v) = v$; p for parent.

Step 1: (Hook: attempt 1) For each vertex $v \in V_i$, if v has a larger neighbour, set $p(v)$ to be the largest larger neighbour of v; this involves just finding the maximum in v's adj-list and comparing it with v; since we have one processor per adj-list entry this takes only $O(\log n)$ steps of recursive doubling.

REMARK: The function $p : V_i \to V_i$ now defines a rooted forest that may have single-vertex trees; every root in the forest points to itself. The roots failed to hook because they do not have larger neighbours.

Step 2: (Check whether successful) For every root r find out whether it has any children; that is, whether r is $p(w)$ for some w. This is done as follows: for each non-root $v \in V_i$, let the processor at v make a mark on the entry for v in $p(v)$'s adj-list; now, for each root r, there is a mark in r's adj-list iff r has a child; find out whether that is so, using $O(\log n)$ steps of recursive doubling over r's adj-list.

Step 3: (Hook: attempt 2) For each root r, if r does not have a child, let $p(r)$ be the first vertex in r's adj-list; r is no longer a root after this.

REMARK: The function $p : V_i \to V_i$ now defines a rooted forest that cannot have single-vertex

trees; we have just hooked up all the single-vertex trees there were on to some other trees; again, every root in the forest points to itself. Note that the equivalence relation defined by the CCs in $G_P = (V_i, \{\{x, p(x)\} | x \in V_i\})$ is a refinement of the equivalence relation defined by the CCs in G. Also from a CC of G_i containing N vertices G_P gets at most $N/2$ CCs.

Step 4: (Contract) For all $v \in V_i$ let $q(v) = p(v)$. Perform $\log n$ steps of pointer jumping on the p-pointers. At the end-of this, each tree in the p-forest is a star; that is, a depth one tree.
REMARK: Now, for each star, we are going to fuse all the vertices belonging to it, into a single (super)vertex.

Step 5: (Find the internal edges) Rename each $\{v, w\} \in E_i$ with $\{p(v), p(w)\}$. If we assume that every adj-list entry in v's list has a pointer to v then this can be done in $O(1)$ time. Clearly, any edge that is now named $\{r, r\}$ for some r is between two vertices belonging to the same star, and hence is "internal".

Step 6: (Edge Plugging) For each non-root $v \in V_i$, insert the adj-list of v into the adj-list of $q(v)$ immediately after the entry corresponding to v in it. (Remember, $q(v)$ is the original $p(v)$; so must be adjacent to v in G_i.) If we assume that every adjacency list has a dummy node at the end of it, this would mean inserting v's list between two nodes in $p(v)$'s list; so the plugging can be done in $O(1)$ time by the processor at v, provided it has both the end-points of v's adj-list; that is easily achieved: assume that an adj-list is a circular doubly-linked list.
REMARK: Now, every root has a list of all the neighbours of all its descendants. But each adj-list can have several entries with same name. For example, suppose $\{u, v\}$ and $\{x, y\}$ are edges in G_i. It may be that $p(u) = p(x)$ and $p(v) = p(y)$. Then both the edges are now named $\{p(u), p(x)\}$.

Step 7: (Find the redundant edges) Sort the adjacency list of each root; using Cole's merge sort this will take $O(\log n)$ time. Now all the edges with the same name have come together. All except the first from each bunch can be declared "redundant".

Step 8: (Remove the waste) All redundant edges and internal edges and dummy entries occuring in the middle of adj-lists are waste. Perform $O(\log n)$ steps of pointer jumping over the adj-lists, with the restriction that pointers must jump over waste entries only.
REMARK: Between any two supervertices, now, there can be at most one edge; no duplicates.

Step 9: (Get ready for the next PHASE) Let G_{i+1} be the graph defined by the roots and their adjacency lists.

Step 10: (Update the earlier drops-out) For all $v \in V_0$ let $p(v) = p(p(v))$.

Clearly, the equivalence relation defined by the CCs in G_i is a refinement of the equivalence relation defined by the CCs in G_{i+1}. Also the number of vertices in each CC of G_{i+1} is at most half the number of vertices in the corresponding CC in G_i. That is, from one PHASE to the next, each CC shrinks by a factor of at least 2. So after $\log n$ PHASEs there will be exactly as many supervertices as there are CCs in the original graph; once the whole of a CC has shrunk to a supervertex it cannot hook–so there will be no further coarsening of that equivalence class.

(Exercise: show that the function $p : V \to V$ will exhibit the requisite properties at the end of the algorithm.)