Terminology

- **SIMD** – single instruction, multiple data stream.
  - Each processor must perform exactly the same operation at each time step, only the data differs.

- **MIMD** – multiple instruction multiple data stream
  - Each processor can perform a different operation
Shared Address Space Architectures

- **UMA** – uniform memory access
  - Each processor has its own memory. Each can access a common shared memory.

- **NUMA** – non-uniform memory access
  - Each processor has its own memory, which can be accessed by other processors.
  - Faster to access your own memory than that of another processor.
Message Passing Architectures

- Processors communicate by sending messages to each other, instead of through memory.

- Static interconnection networks:
  - Many possible topologies:
  - Fixed degree
  - Hypercube
  - In graph terms, the goals are usually:
    - keep the maximum degree small.
    - keep the diameter small.

- Dynamic interconnection networks:
  - crossbar switching network
  - bus based networks, (e.g. Ethernet)
PRAM

- Parallel random access machine
- A straightforward generalization of standard serial computers.
- \( p \) processors that each have local memory, and symmetrical access to a large shared memory.
- MIMD UMA
Parallel Max

- Sequential Max algorithm required $n-1$ comparisons.
- Parallel Max also requires that many comparisons, but many of them can take place at the same time.
- We no longer count total operations, we count the maximum number of operations performed by any processor.
- The tournament algorithm for Max has an easy parallel implementation.
keytype parlargest(int n, keytype S[]) {
    index step, size;
    local index p;
    local keytype first, second;

    p = index of this processor;
    size = 1;
    for (step = 1; step <= lg n; step++) {
        first = S[2*p - 1];
        second = S[2*p - 1 + size];
        S[2*p - 1]= max(first,second);
        size = 2 * size;
    }
    return S[1]
}
Parallel Max Analysis

- Algorithm is assuming input size is a power of 2.
- Main loop executes $\log n$ times.
Parallel Binomial Coefficient

- Recall the recursive approach for computing the binomial coefficient:

\[
\binom{n}{k} = \begin{cases} 
\binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\
1 & k = 0 \text{ or } k = n 
\end{cases}
\]

- This can be calculated via dynamic programming:
  - \( B[i][j] = B[i-1][j-1] + B[i-1][j] \)

- Since entries in a particular row do not depend on each other, every entry in a row can be computed simultaneously.
int parbin(int n, int k)
{
    B[0..n][0..k];
    int i;
    local int j;
    j = index of this processor;
    for (i = 0; i <= n; i++) {
        if (j <= min(i,k)) {
            if (j == 0 || j == i)
                B[i][k] = 1;
            else
                B[i][k] = B[i-1][j-i] + B[i-1][j];
        }
    }
    return B[n][k]
}
Parallel Binomial Coefficient Analysis

- Run time of sequential algorithm $\Theta(nk)$.
- Run time of parallel algorithm $\Theta(n)$.
Dynamic Programming in General

● Is it always possible to speed up dynamic programming algorithms with more processors?
● How about computing the $n$th Fibonacci term?
Parallel Sorting

- With $n^2$ processors it is possible to sort $n$ items in $\lg n$ time.
- Unknown whether there is a $\lg n$ time algorithm that uses only $n$ processors.
- Let's look at a linear time sorting algorithm...

Parallel Merge Sort

• Recall the non-recursive merge-sort implementation:
  - divide the unsorted list into pairs, sequentially merge pairs
  - sequentially merge sorted sets of two.
  - merge sets of four. etc.
• In a parallel implementation all merges of the same size can occur simultaneously.
• The individual merges are performed sequentially.
• Most comparisons done by any one processor:
  \[ W(n) = W(n/2) + n - 1 \]
  \[ \Theta(n) \]
Odd Even Merge Sort

- The previous algorithm would be much improved if we could parallelize the merge operation - It can be done!
- In order to merge to sorted lists A and B:
  - First partition A and B into odd and even indexed sublists:
    - even(A) = \(a_0, a_2, a_4, \ldots\) odd(A) = \(a_1, a_3, a_5, \ldots\)
  - Recursively merge even(A) and odd(B) to get a new list C.
  - merge odd(A) with even(B) to get D.
  - Now merge C and D:
    - interleave them: \(L' = c_0, d_0, c_1, d_1, \ldots\)
    - swap any neighbors that are out of order (just once) to get L.
    - Magically, L is sorted.

Correctness

- We haven't spent much time proving algorithm correctness – usually it has been obvious.
- This is a non-obvious case.
- The proof (which we won't do in detail) uses the 0-1 sorting lemma:
  - Any oblivious comparison exchange sort that correctly sorts any list of 0's and 1's, correctly sorts arbitrary lists.
- The gist is:
  - $C$ and $D$ each have about the same number of 0's, they each have half the 0's from $A$ and half from $B$.
  - So when $C$ and $D$ are interleaved, there won't be many 0's and 1's out of order.
Running Time

- $O(\lg^2 n)$