# CS 467/567: Notes on Parallel Algorithms

Stefan D. Bruda

Winter 2023

## 1 Parallel Models

Recall that algorithms are developed on abstract models of computation in order to get rid of all the nitty and gritty details of programming for an actual machine. Most of the time in the course we used the Random Access Machine (RAM) for the purpose, which allows us to write algorithms in pseudocode.

There are at least two similar models for parallel computation. The simplest extension to the RAM for such a purpose is called unsurprisingly enough the *Parallel Random Access Machine*, or PRAM for short. In this model (see Figure 1) several RAM processors are all able to access common memory. Given that the processors are the same that we have seen earlier, the programming language for the PRAM continues to be the pseudocode. The difference is that we now have multiple processors and we need to describe computations happening simultaneously on all of them. We use for this purpose the following new instruction:

**for** $i = 1$ **to** $n$ **do in parallel** { statements parameterized on processor $p_i$ }

Another wrinkle is that shared memory is well, shared. We therefore need to know what happens on concurrent read operations (when two or more processors read from the same memory location) and also concurrent write operations (when two or more processors want to write into the same memory location). For each of the two operations (read and write) we can either allow it or not; in the latter case any attempt at concurrent access will result in an erroneous program termination. We therefore can have concurrent or exclusive read machines (CR or ER) as well as concurrent write or exclusive write machines (CW and EW). Given the fact that allowing concurrent write but disallowing concurrent read does not make a lot of sense we end up with three kinds of machines: EREW PRAM, CREW PRAM, and CRCW PRAM.

Once concurrent write is allowed another can of worms is opened: what should we do with the multiple values written by multiple processors into the same memory location. There are three major solutions to that: First, we can insist that all the values that are written concurrently are the same (Common CRCW PRAM); if this is not the case, then the algorithm is terminated. Secondly, we can assign priorities to the processors, such that $p_1$ has the highest, and $p_n$ (the last one) the lowest priority. Then, on a concurrent attempt to write the highest priority processors succeeds and the data written by all the others are discarded (Priority CRCW PRAM). Finally, an operation can be performed on all the values that are concurrently written into a single memory location and the result get written instead (Combining CRCW PRAM). The operation can be any arithmetic or logic operation that can be performed by a processor and is at the control of the algorithm. On the surface it may be argued that Combining operation is too powerful to be realistic, but in fact this is not as far fetched as it looks; see the textbook for implementation details.

The PRAM is a very convenient model, but in practice most contemporary massively parallel machines do not feature shared memory. Instead they consist of computation units which have their own local memory and processor (thus being equivalent to the RAM), and are connected together through some form of fast network. If we model the network by point-to-point connections (which is a realistic model) then we have our second parallel model called the *Interconnection Network*. The programming language of this model continues to be the pseudocode, which is used to describe independently all the computations taking place in all the RAM processing units. In practice there is some need to represent these computations

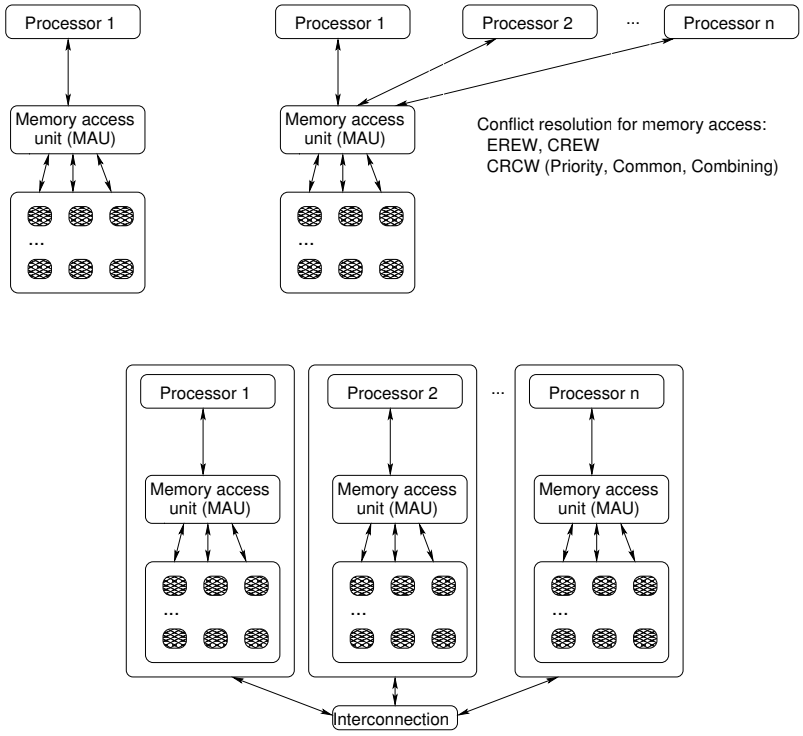


Figure 1: The RAM (top left), the PRAM (to right), and the interconnection network (bottom)

in a compact manner and so some indexing will appear in the pseudocode just like for the PRAM, but this is just syntactical sugar. Additionally, **send** and **receive** primitive operations are also needed. They only operate on point-to-point connections in the network.

## 2 Performance of parallel algorithms

Like in the sequential case we charge one unit of time for each sequential operations performed by one processor. Note that in the same time unit multiple processors will execute instructions in parallel, so that for an $n$-processor machine it is possible to execute $n$ instructions in one time units (one per processor). Executing instructions independently on processors is often referred to as *computation steps*.

In interconnection networks we also need to move data around. When this happens we often say that we have a *routing step*. We charge $O(1)$ time for each direct link that is traversed by the data (so that in the general case the cost of moving data depends on the distance between the source and the destination).

While we do not have routing in shared memory models, the memory access unit is substantially more complex than for sequential machines. Most algorithm analysis ignores this additional complexity and so charge a constant time for memory access (*uniform analysis*). This model works well in practice (that is, is realistic enough) and so it is widely adopted. It is also possible to charge : $O(\log M)$ time for accessing one word in memory of size $M$ (*discriminating analysis*).

Putting all these costs together we obtain the *running time* $t : \mathbb{N} \to \mathbb{N}$ of a parallel algorithm. Most commonly the running time is determined based on a worst case analysis, just like for sequential algorithms.

The running time is fine, but usually when analyzing a parallel algorithm we do so by comparing it with a sequential computation that accomplishes the same thing. Therefore three measures of performance

are used: *speedup* $S_p : \mathbb{N} \to \mathbb{N}$, *efficiency* $E_p : \mathbb{N} \to \mathbb{N}$, and *cost* $c_p : \mathbb{N} \to \mathbb{N}$:

$$S_p = \frac{t_1}{t_p} \qquad E_p = \frac{S_p}{p} \qquad c_p = p \times t_p$$

where $t_p : \mathbb{N} \to \mathbb{N}$ is the time taken by the $p$-processor algorithm being analyzed to solve the problem, and $t_1 : \mathbb{N} \to \mathbb{N}$ is the time taken by the *best known sequential algorithm* to solve the same problem. It is crucial to use the best known sequential algorithm, for else the performance of the parallel algorithm can be made arbitrarily better just by comparing it with an arbitrarily bad sequential algorithm.

**Theorem 1** ***(Speedup theorem)*** *In the classical theory of parallel algorithms $S_p \leqslant p$ and so $E_p \leqslant 1$.*

The proof of the speedup theorem is not difficult, see the textbook for details. The same goes for the slowdown theorem below.

Speedup and efficiency are usually (but not always) invariable with the input size. A parallel algorithm with $S_p = p$ (or $E_p = 1$, or $c_p = t_1$) is called *optimal*. At the other end of the spectrum, if $S_p = O(1)$ then the running time of the parallel algorithm is just as bad as the running time of a sequential algorithm. Problems for which $S_p = O(1)$ for any parallel algorithm are called *inherently sequential*. This is believed to happen to a whole class of problems called *P-complete problems*.

Another important measure is the *slowdown*, which is the effect on running time of reducing the number of processors

**Theorem 2** ***(Slowdown theorem)*** *In the classical theory of parallel algorithms if a certain computation can be performed with $p$ processors in time $t_p$ and with $q < p$ processors in time $t_q$ then $t_p \leqslant t_q \leqslant t_p + pt_p/q$.*

We will not insist on slowdown for now, but we need to note that the actual *number of processors* also important. It is possible that an analysis of the algorithm reveals that a number of processors are idle most of the time and so can be discarded without affecting the performance. Sometimes the optimal running time can only be achieved with a certain number of processors. Sometimes reducing the number of processors below a certain threshold results in an unacceptable slowdown.

### 2.1 Combinational circuits

The combinational circuit models an "unfolded" parallel computation and so it is a useful tool in the analysis of parallel algorithms. We will use in this course more traditional tools for algorithm analysis, so this section is for your information only.

In a combinational circuit processors capable of performing the usual logic and arithmetic operations on $O(\log n)$-sized words but having only a constant number of internal registers. These processors are connected to each other as vertices in a directed acyclic graph, with vertices with no incoming edges called *input processors*, and vertices with no outgoing edges called *output processors*. The processors can be viewed as aligned into columns, one column per distance from the input nodes. It is convenient (though not strictly necessary) to have all the output vertices in the rightmost column.

The *depth* of the circuit (or number of columns) and the *width* of the circuit (the number of processors in the largest column) are both performance measures for a combinational circuit. The reason is that the combinational circuit is used to represent the unfolded computation of an "usual" parallel machine. In this case the depth of the circuit is equivalent to running time of the parallel machine and the width is equivalent to the number of processors. The cost of the computation is then the depth times the width of the circuit.

## 3 Parallel prefix computations on the PRAM

Prefix computation is a problem that is a very useful building block for more complex parallel algorithms, replacing other techniques that are not useful in a parallel algorithm because they are inherently sequential.

In solving it, we will also encounter a technique which will turn out to be useful in many other circumstances.

First, the prefix computation problem is stated as follows: *Given an array x with n values, find all the prefix sums* $s_i = \sum_{k=0}^{i} x_i$, $0 \leqslant i < n$, *where the summation is done according to an associative binary operation* $\circ$. The following sequential algorithm will provide an optimal solution for the problem:

**Algorithm** RAM-PREFIX $(x_{0\ldots n-1})$ **returns** $s_{0\ldots n-1}$:

1. $s_0 \leftarrow x_0$
2. **for** $i = 1$ **to** $n-1$ **do**:
   (a) $s_i \leftarrow s_{i-1} \circ x_i$

Indeed, the lower sequential bound for the problem is linear (all the values need to be considered), and the simple algorithm above meets this lower bound. We conclude that prefix sum is not a very interesting problem sequentially.

Here is now an $n$-processor PRAM algorithm for the problem;

**Algorithm** PRAM-PREFIX $(x_{0\ldots n-1})$ **returns** $s_{0\ldots n-1}$:

1. **for** $i = 0$ **to** $n-1$ **do in parallel**:
   (a) $s_i \leftarrow x_i$
2. **for** $j = 0$ **to** $\log n - 1$ **do**:
   (a) **for** $i = 2^j$ **to** $n-1$ **do in parallel**:
      i. $s_i \leftarrow s_{i-2^j} \circ s_i$

Try to trace the algorithm to convince yourself that it is correct. Obviously the running time of the algorithm is $t_n(n) = O(\log n)$. Comparing this with the sequential time $t_1(n) = O(n)$ we obtain a cost $c_n(n) = O(n \log n)$. Therefore PRAM-PREFIX is not optimal. However, we can use a *divide and conquer* technique that will reduce the number of processors required while maintaining the running time, thus yielding an optimal parallel algorithm.

The general idea of the technique is to divide the input data into $m$ groups (form some $m < n$), solve the problem sequentially for each group (meaning that one processor will be used for each group, with the $m$ processors solving the problem for the $m$ groups in parallel), and then use a true parallel algorithm to combine the results.

In the particular case of prefix computations, we exploit for this purpose the associativity of $\circ$. Let $k = \log n$ and $m = n/k$ (rounded). We then use an $m$-processor algorithm:

**Algorithm** OPTIMAL-PRAM-PREFIX $(x_{0\ldots n-1})$ **returns** $s_{0\ldots n-1}$:

1. All the processors $P_i$, $0 \leqslant i < m$ use in parallel RAM-PREFIX to compute he prefix sums $s_{ik}$, $s_{ik+1}, \ldots, s_{(i+1)(k-1)}$, where $s_{ik+j} = x_{ik} \circ x_{ik+1} \circ \cdots \circ x_{ik+j}$
   - $O(k) = O(\log n)$ time
2. Now PRAM-PREFIX is used on all the processors to compute the prefix sum of the sequence $\langle s_{k-1}, s_{2k-1}, \ldots, s_{n-1} \rangle$; the result is put back into $\langle s_{k-1}, s_{2k-1}, \ldots, s_{n-1} \rangle$
   - At the end of this step $s_{ik-1}$ will be replaced with $s_{k-1} \circ s_{2k-1} \circ \cdots \circ s_{ik-1}$
   - $O(\log m) = O(\log(n/\log n))$ time
3. Fnally, all processors $P_i$, $1 \leqslant i < m$ perform sequentially $s_{ik+j} \leftarrow s_{ik-1} \circ s_{ik+j}$ for all $0 \leqslant j \leqslant k-2$
   - Executed sequentially by all processors (except $P_0$)
   - $O(k) = O(\log n)$ time

The overall running time is $t_{n/\log n}(n) = O(\log n) + O(\log(n/\log n)) + O(\log n) = O(\log n)$ for an optimal cost $c(n) = O(n)$.

The algorithm also illustrated how an $m$-processor PRAM can be made to run an algorithm designed to run on $n$ processors, $n > m$. As already mentioned, this "self-simulation" is extremely useful in practice, as it shows how to solve a problem with less that the number of processors required theoretically.

The optimal implementation is not without drawbacks. Indeed, a certain storage overhead is necessary for this algorithm as opposed to the previous, non-optimal one. So if optimality is not a concern (e.g., we have $n$ processors anyway) then the original algorithm is preferable.

## 3.1 The importance of prefix computations

Sequentially the prefix computation performs a "sweep" of the input sequence; such a sweep can be accomplished in many other ways (some times more efficient!). A parallel algorithm however performs the "sweep" in an optimal amount of time using prefix computations! It is often the case that such a sweep cannot be done more efficiently in parallel.

As a fist example consider the maximum sum subsequence probem formulated as follows: *Given a sequence of (not necessarily positive) integers $\langle x_0, x_1, \ldots, x_{n-1}\rangle$ find two indices $u$ and $v$ such that $x_u + \cdots + x_v$ is maximal.* The problem is solved optimally by the following well-known algorithm. You probably already encountered it in CS 327 or equivalent:

**Algorithm** RAM-MAX-SUM $(x_{0\ldots n-1})$ **returns** $u, v$:

1. $Maxseen \leftarrow x_0$; $u \leftarrow 0$; $v \leftarrow 0$; $Maxhere \leftarrow x_0$; $q \leftarrow 0$
2. **for** $i = 0$ **to** $n$ **do**:
   (a) **if** $Maxhere \geqslant 0$ **then** $Maxhere \leftarrow Maxhere + x_i$
   **else** $Maxhere \leftarrow x_i$; $q \leftarrow i$
   (b) **if** $Maxseen < Maxhere$ **then** $Maxseen \leftarrow Maxhere$; $u \leftarrow q$; $v \leftarrow i$

A parallel algorithms solving the maximum sum subsequence cannot do this kind of traversal efficiently; such a traversal is *inherently sequential*. We therefore retort to prefix computations, as follows:

**Algorithm** PRAM-MAX-SUM $(x_{0\ldots n-1})$ **returns** $u, v$:

1. Perform a prefix computation over $(x_{0\ldots n-1})$ storing the result in $(s_{0\ldots n-1})$
2. Perform a modified prefix computation over $(s_{n-1\ldots 0})$ using max for $\circ$, storing the maximum in $(m_{n-1\ldots 0})$ and the index where the maximum occurs in $(a_{n-1\ldots 0})$
   - The modification is that in addition to the actual value the index at which the value appears is also stored
3. Perform in parallel $b_i \leftarrow m_i - s_i + x_i$
4. Use a modified prefix computation with max for $\circ$ to find $L \leftarrow \max_{0\leqslant i<m} b_i$ together with the index $u$ of $L$
5. Let $v \leftarrow a_u$
6. **return** $u, v$

Here is a sample run of the algorithm:

| Input | $x_i$ | -4 | 2 | 6 | -1 | -7 | 4 | 2 | -1 |
|---|---|---|---|---|---|---|---|---|---|
| Prefix sum | $s_i$ | -4 | -2 | 4 | 3 | -4 | 0 | 2 | 1 |
| Modified prefix sum | $m_i$ | 4 | 4 | 4 | 3 | 2 | 2 | 2 | 1 |
| with max as $\circ$ | $a_i$ | 2 | 2 | 2 | 3 | 6 | 6 | 6 | 7 |
| $b_i \leftarrow m_i - s_i + x_i$ | $b_i$ | 4 | 8 | 6 | -1 | -1 | 6 | 2 | -1 |

We then have $L = 8$, $u = 1$, and $v = 2$.

The algorithm above is optimal for $n/\log n$ processors. The optimality is based on the optimality consideration for the prefix computation algorithm; indeed, that is basically the only thing that we use in our algorithm.

As another example, consider the polynomial interpolation problem, stated as follows: *Given $n+1$ pairs of numbers $(x_i, y_i)$, $0 \leqslant i \leqslant n$ such that $x_0 < x_1 < \cdots < x_n$, find a polynomial $h(x)$ such that $h(x_i) = y_i$, $0 \leqslant i \leqslant n$.*

To solve this problem we can use Newton's interpolation method:

$$h(x) = y_0 + Y_{01}(x - x_0) + Y_{02}(x - x_0)(x - x_1) + \cdots + Y_{0n}(x - x_0)(x - x_1)\cdots(x - x_n)$$

$$\text{where } Y_{ii} = y_i \text{ and } Y_{i(i+j)} = \frac{Y_{i(i+j-1)} - Y_{(i+1)(i+j)}}{x_i - x_{i+j}}$$

Solving the recursion for $Y_{0i}$, $0 \leqslant i \leqslant n$ yields

$$Y_{0i} = \frac{y_0}{X_{01}X_{02}\cdots X_{0i}} + \frac{y_1}{X_{10}X_{12}\cdots X_{1i}} + \cdots + \frac{y_i}{X_{i0}X_{i1}\cdots X_{i(j-1)}}$$

where $X_{ij} = x_i - x_j$ for all $i \neq j$. The denominators can be computed using prefix computations with the scalar multiplication operation. In fact, one prefix computation computes all the denominators for numerator $y_j$, thus yielding an optimal parallel algorithm.

We end our incursion into prefix computation by consider the array packing problem: *Given an array $X_{1\ldots n}$ with some values therein labeled, bring all the labeled values into contiguous positions.*

A sequential algorithm running in the optimal $O(n)$ time maintains two pointers in the array $q$ and $r$ with initial values $q = 1$ and $r = n$. Then $q$ advances to the right if $X_q$ is labeled, $r$ advances to the left if $X_r$ is unlabeled, and $X_q$ and $X_r$ are swapped whenever $X_q$ is unlabeled and $X_r$ is labeled. The labeled elements are all in adjacent positions in the first part of the array as soon as $q \geqslant r$

An optimal $n/\log n$-processor parallel algorithm running in $O(\log n)$ time proceeds like this:

1. Create a secondary array $S$ of size $n$ such that $S_i = 1$ if $X_i$ is labeled and $s_i = 0$ otherwise
2. Compute a prefix sum over $S$
3. Move each labeled value $X_i$ to index $S_i$

Array packing is another of those problems that have significant practical applications in parallel algorithms.

# 4 Divide and Conquer on the PRAM

Awhile ago I promised that we will come back to divide and conquer techniques, so now is the time to keep my promise. We will continue to use the PRAM as our model of parallel computation.

## 4.1 Binary search

Let's start things off by consider the classical and relatively simple problem of binary search: *Given a sequence $S_{1..n}$ sorted in increasing order and a value $x$, find an index $k$ such that $S_i = x$.* If $n$ processors are available then the problem can be solved in constant time on the CRCW PRAM as follows:

1. All processors read $x$
2. Each processor $P_i$ compares $x$ with $S_i$

3. All processors $P_i$ (if any) that found $x = S_i$ write $j$ into $k$ using min as combining operator if such an operator necessary

The algorithm runs on any kind of CRCW PRAM other than Common as long as we do not care which index $k$ is returned. If we can guarantee that $x$ appears at most one time in the array then the algorithm also runs on the CREW PRAM.

The algorithm has a great running time (one cannot do better than $O(1)$!), but a horrible $O(n)$ cost. The simple approach of dividing the work equally between processors worked well for prefix computations, so let's try to improve our solution using the same simple divide and conquer approach. Using $N < n$ processors,

1. Divide the sequence $S$ into $N$ roughly equally sized subsequences (of length $O(n/N)$ each)
2. Each processor performs a sequential binary search to search for $x$ in one subsequence
3. Those procesors (if any) that found $x$ write the respective index into $k$ using min as combining operator

Did we do better? Well, some but not a lot. Indeed, we obtain a running time $O(\log(n/N))$ which is faster than the sequential solution but not by a lot given the $N$ processor expense.

Next we try a more complex divide and conquer approach: Since we have $N$ processors, we might as well perform an $N+1$-way (rather than binary) search. At each stage the sequence under consideration is split into $N+1$ subsequences. We will operate in a sequence of stages. In the first stage the whole sequence is under consideration, while in subsequent stages we will consider only a subsequence. At each stage we perform the following algorithm:

1. Each processor $P_i$ compares $x$ with the elements $s$ at the right boundary of the $i$-th subsequence
2. If $x < s$ then all the elements in the $i+1$-st and higher subsequences can be discarded
3. If $x > s$ then all the elements in the $i$-th and lower subsequences can be discarded
4. If $x = s$ then the index has been found

This is a trivial extension of the binary search concept: the process reduces the sequence under consideration $N$ times rather than just halving it (like in the sequential case). The overall running time is thus $O(\log_N n)$.

## 4.2 Merging

This is again a familiar problem: *Given two sequences of numbers (or more generally comparable values)* $A = \langle a_1, a_2, \ldots, a_r \rangle$ *and* $B = \langle b_1, b_2, \ldots, b_s \rangle$ *sorted in nondecreasing order, compute the sequence* $C = \langle c_1, c_2, \ldots, c_{r+s} \rangle$ *such that each* $c_i$ *belongs to either* $A$ *or* $B$*, ech* $a_i$ *and* $b_i$ *appear exactly once in* $C$*, and the sequence* $C$ *is sorted in increasing order*.

I am sure that you are also familiar with the following classical, sequential algorithm with optimal $O(n)$ running time:

**Algorithm** RAM-MERGE$(a_{1\ldots r}, b_{1\ldots s})$ **returns** $c_{1\ldots r+s}$:

1. $i \leftarrow 1$, $j \leftarrow 1$
2. **for** $k = 1$ **to** $r + s$ **do**
   (a) **if** $a_i < b_j$ **then** $c_k \leftarrow a_i$, $i \leftarrow i + 1$
   (b) **else** $c_k \leftarrow b_j$, $j \leftarrow j + 1$

We want to develop a parallel algorithm running in sublinear time (we do want to do better that the sequential solution) using an adaptive number of processors. Furthermore we are looking for an optimal algorithm. Assume that $r \leqslant s$; this is without loss of generality, since if this is not the case then we can just flip the two sequences. We then have the following algorithm:

**Algorithm** PRAM-MERGE$(a_{1\ldots r}, b_{1\ldots s})$ **returns** $c_{1\ldots r+s}$:

1. Select $N-1$ elements from $A$ that divide $A$ into $N$ sequences of approximately equal size; call this sequence $A' = \langle a'_1, a'_2, \ldots \rangle$. Similarly find the sequence $B' = \langle b'_1, b'_2, \ldots \rangle$ that divide $B$ into $N$ sequences of roughly the same size (constant time):
   (a) **for** $i = 1$ **do in parallel** $a'_i \leftarrow a_{i\lceil r/N \rceil}$, $b'_i \leftarrow b_{i\lceil s/N \rceil}$
2. Merge $A'$ and $B'$ into a sequence of triples $V = \langle v_1, v_2, \ldots v_{2N-2} \rangle$, where each triple consists of an element of $A'$ or $B'$, its position in $A'$ or $B'$, and the name of the sequence of origin ($A$ or $B$) ($O(\log N)$ time):
   (a) **for** $i = 1$ **to** $N$ **do in parallel**
      i. Processor $P_i$ uses binary search on $B'$ to find the smallest $j$ such that $a'_i < b'_j$
      ii. **if** $j$ exists **then** $v_{i+j-1} \leftarrow (a'_i, i, A)$ **else** $v_{i+N-1} \leftarrow (a'_i, i, A)$
   (b) **for** $i = 1$ **to** $N$ **do in parallel**
      i. Processor $P_i$ uses binary search on $A'$ to find the smallest $j$ such that $b'_i < a'_j$
      ii. **if** $j$ exists **then** $v_{i+j-1} \leftarrow (b'_i, i, B)$ **else** $v_{i+N-1} \leftarrow (b'_i, i, B)$
3. $Q_1 \leftarrow (1, 1)$
4. **for** $i = 2$ **to** $N$ **do in parallel**
   (a) **if** $v_{2i-2} = (a'_k, k, A)$ **then**
      i. processor $P_i$ uses binary search on $B$ to find the smallest $j$ such that $b_j > a'_k$
      ii. $Q_i \leftarrow (k\lceil r/N \rceil, j)$
   (b) **else**
      i. processor $P_i$ uses binary search on $A$ to find the smallest $j$ such that $a_j > b'_k$
      ii. $Q_i \leftarrow (j, k\lceil s/N \rceil)$
5. **for** $i = 1$ **to** $N$ **do in parallel**
   (a) Processor $P_i$ uses RAM-MERGE and $Q_i = (x, y)$ to merge two subsequences beginning at $a_x$ and $b_y$ and places the result in $C$ beginning at index $x + y - 1$. The merge continues until either
   ($a$) an element larger than or equal to the firt component of $v_{2i}$ in each of $A$ and $B$ (when $i \leqslant N-1$), or
   ($b$) no elements are left in either $A$ or $B$ (when $i = N$)

In steps are 3 to 5 each processor merges and inserts into $c$ the elements of two subsequences, one from $a$ and one from $b$. The indices of the two elements (the one from $a$ and the other from $b$) at which each processor begins merging are first computed and stored in an array $Q$ of pairs. This step takes $O(r + s/N)$ time. The overall running time is therefore $O(n/N + \log n)$, which is optimal for for $N \leqslant n/\log n$ processors, as desired.

This is the most complex algorithm so far (and also the most complex that you will encounter in the course), so let's see how it works by running it on the following sequences, with $r = s = 12$ and $N = 3$:

$$\begin{aligned} a &= \underbrace{2,3,4}, \underbrace{6,11,12}, \underbrace{13,15,16}, \underbrace{20,22,24} \\ b &= \underbrace{1,5,7}, \underbrace{8,9,10}, \underbrace{14,17,18}, \underbrace{19,21,23} \end{aligned}$$

The split performed in Step 1 is already outlined above. Accordingly, we have

$$\begin{aligned} A' &= 4, 12, 16 \\ B' &= 7, 10, 18 \end{aligned}$$

Step 2$(a)$ then proceeds as follows:

- $P_1$: $i = 1 \wedge j = 1 \Rightarrow v_1 = (4, 1, A)$
- $P_2$: $i = 2 \wedge j = 3 \Rightarrow v_4 = (12, 2, A)$
- $P_3$: $i = 3 \wedge j = 3 \Rightarrow v_5 = (16, 3, A)$

and then Step 2$(b)$ goes like this:

- $P_1$: $i = 1 \wedge j = 2 \Rightarrow v_2 = (7, 1, B)$
- $P_2$: $i = 2 \wedge j = 2 \Rightarrow v_3 = (10, 2, B)$
- $P_3$: $i = 3 \wedge j = N/A \Rightarrow v_6 = (18, 3, B)$

We thus have $v = (4, 1, A), (7, 1, B), (10, 2, B), (12, 2, A), (16, 3, A), (18, 3, B)$.

Note now that $\lceil r/N \rceil = \lceil s/N \rceil = 3$. Then according to Steps 3 and 4 $Q = (1, 1), (5, 3), (6, 7), (10, 9)$. Therefore the following sequences are merged sequentially in Step 5:

- $P_1$: $2, 3, 4, 6$ and $1, 5$
- $P_2$: $11$ and $7, 8, 9, 10$
- $P_3$: $12, 13, 15$ and $14, 17$
- $P_4$: $20, 22, 24$ and $19, 21, 23$

## 4.3 Sorting

No course on algorithms can be complete without a discussion on sorting. We begin such a discussion by performing sorting in constant time on the Combining CRCW PRAM as follows:

**Algorithm** CRCW-SORT$(S_{1..n})$ **returns** $S'_{1..n}$:

1. **for** $i = 1$ **to** $n$ **do in parallel**
   **for** $j = 1$ **to** $n$ **do in parallel**
   (a) **if** $s_i > s_j \vee s_i = s_j \wedge i > j$
   (b) **then** $P_{ij}$ writes 1 in $c_i$ using $+$ as combining operation
   (c) **else** $P_{ij}$ writes 0 in $c_i$ using $+$ as combining operation
2. **for** $i = 1$ **to** $n$ **do in parallel**
   (a) $P_{i1}$ stores $S_i$ into $S'_{1+c_i}$

The method used here is *enumeration* or *rank sorting*. We have an $O(n^2)$-processor algorithm with constant running time for an $O(n^2)$ cost. This is obviously not optimal, but still attractive given the speed. However, the algorithm likely not of a great practical value since the number of processors is very high.

What about the CREW PRAM though? This is often regarded as more practically meaningful than the CRCW variant. We can still compare one pair of values $(s_i, s_j)$ in each processor, but we cannot write all the results $c_i$ in a single memory location. We therefore need to perform the summation explicitly:

**Algorithm** CREW-SORT$(S_{1..n})$ **returns** $S'_{1..n}$:

1. If $s_i > s_j \vee s_i = s_j \wedge i > j$ then processor $P_{ij}$ writes 1 into $c_{ij}$; otherwise $P_{ij}$ writes 0 into $c_{ij}$
2. Set $c_i$ to $\sum_{j=1}^{n} c_{ij}$ then continue as in the CRCW algorithm

The extra step $c_i \leftarrow \sum_{j=1}^{n} c_{ij}$ can be performed as follows: we keep adding (in parallel) pairs of values until a single value remains. This takes $O(\log n)$ time using $n$ processors. Overall we use $O(n^2)$ processors and end up with an $O(\log n)$ running time.

We note an $O(\log n)$ slowdown of the CREW algorithm compares with the Combining CRCW variant. This is a price to be paid for not being allowed to charge one time unit for the Combining operations and so happens to most algorithms.

Divide and conquer can be used to obtain an optimal sorting algorithm for the CREW PRAM by transforming the good old merge sort into a parallel implementation:

**Algorithm** CREW-SORT($S_{1..n}$) **returns** $S_{1..n}$:

1. Distribute equal size subsequences of $S$ to the $N$ processors. Each processor will then sort its subsequence sequentially using an optimal algorithm.
2. Keep merging pairwise adjacent subsequences (in parallel) until one sequence (of length $n$) is obtained (using PRAM-MERGE)

Step 1 takes $O((n/N)\log(n/N))$ time. Step 2 proceeds iteratively, with iteration $k$ featuring $N/k$ subsequences (of length $kn/N$ each) to merge. We then allocate $O(k)$ processor for each merge, which results in $O(n/N + \log(kn/N)) = O(n/N + \log n)$ time per iteration. Since we need $O(\log N)$ iterations the overall running time is $O((n/N)\log N + \log n \log N)$.

Note in passing that it is also possible to sort faster ($O(\log n)$ time with $O(n)$ processors, which is still optimal). However, such an algorithm does not scale well.

## 4.4 Convex hull

Last but not least, we return as promised to a problem that we already discussed in details namely, the convex hull. The divide and conquer algorithm discussed earlier can be adapted in a relatively straightforward manner to a parallel implementation as follows:

**Algorithm** PRAM-CONVEX-HULL($n, Q$):

1. Sort the points in $Q$ according to their $x$ coordinate
2. Partition $Q$ into $n^{1/2}$ sets $Q_1, Q_2, \ldots, Q_{n^{1/2}}$ of $n^{1/2}$ points each such that the sets are separated by vertical lines and $Q_i$ is to the left of $Q_j$ iff $i < j$
3. **for** $i = 1$ to $n^{1/2}$ **do in parallel**
   (a) **if** $|Q_i| < 3$ **then** $CH(Q_i) \leftarrow Q_i$
   (b) **else** $CH(Q_i) \leftarrow$ PRAM-CONVEX-HULL($n^{1/2}, Q_i$)
4. **return** PRAM-MERGE-CH($CH(Q_1), CH(Q_2), \ldots, CH(Q_{n^{1/2}})$)

If we allocate $O(n)$ processors then we can do Step 1 in $O(\log n)$ time. Step 2 takes constant time since the sets $Q_i$ are all subsequences of $Q$. We anticipate a bit and claim that Step 4 takes $O(\log n)$ time. If so, the overall running time is $t(n) = t(n^{1/2}) + c \log n$ and so $t(n) = O(\log n)$ for a cost of $O(n \log n)$. Therefore in the classical sense (non-output sensitive complexity) the algorithm is optimal.

Merging the convex hulls can be accomplished in parallel as follows:

**Algorithm** PRAM-MERGE-CH($CH(Q_1), CH(Q_2), \ldots, CH(Q_{n^{1/2}})$):

1. Let $u$ be the leftmost point of $CH(Q_1)$ and $v$ the rightmost point of $CH(Q_{n^{1/2}})$

2. Identify the upper hull:
   (a) Assign $O(n^{1/2})$ processors to each $CH(Q_i)$
   (b) Each processor assigned to $CH(Q_i)$ finds the upper tangent common between $CH(Q_i)$ and $CH(Q_j)$ for some $i \neq j$
   (c) Between all common tangents between $CH(Q_i)$ and $CH(Q_j)$, $j < i$ let $L_i$ (tangent with $CH(Q_i)$ at point $l_i$) be the tangent with the smallest slope
   (d) Between all common tangents between $CH(Q_i)$ and $CH(Q_j)$, $j > i$ let $R_i$ (tangent with $CH(Q_i)$ at point $r_i$) be the tangent with the smallest slope
   (e) If the angle formed by $L_i$ and $R_i$ is smaller than 180 degrees then no points from $CH(Q_i)$ are in the upper hull; otherwise include in the upper hull all the points between $l_i$ and $r_i$ (inclusive)
   (f) Identify the upper hull as all the points from $u$ to $r_1$, then all the points identified above, then all the points from $r_{n^{1/2}}$ to $v$ (inclusive)
3. Identify the lower hull (similar to the upper hull)
   - The lower hull is identified as above but this time $u$ and $v$ are excluded
4. Return the union of the upper and lower hulls (using *array packing*)

We compute the upper tangent of $CH(Q_i)$ and $CH(Q_j)$ in $O(\log n)$ time as follows: Let $s$ and $w$ be the middle points in the (sorted) upper hulls from $CH(Q_i)$ and $CH(Q_j)$. If $\overline{sw}$ is the upper tangent of $CH(Q_i)$ and $CH(Q_j)$ then we are done (Case a in Figure 2). Otherwise repeat from Step 1 but excluding at least half the points of at least one upper hull (Cases b–h in Figure 2).

# 5 Interconnection networks

The PRAM is a very powerful model, rarely realizable in practice. It is however important for the theory of algorithms; in particular lower bounds are strong on the PRAM. The PRAM is also surprisingly equivalent to other, realistic models. On the other hand, most massively parallel machines are laid out as interconnection networks.

From the point of view of the theory of algorithms interconnection networks typically have fixed topology. An interconnection network is therefore a *family of graphs* with RAM processors (including storage) as nodes and data links as edges. The number of processors (nodes) may vary, but the topology remains the same. Possible topologies include linear array, mesh, tree, hypercube, and fully connected; the latter is not very realistic though. Note in passing that models with variable topology also exist (but we will not cover them in this course).

A useful measure of communication complexity on a given network topology is the *diameter* of the network, defined as the maximum number of links on the shortest path between two processors. This is now much time it takes for a piece of data to be communicated from one processor to another in the worst case.

Any PRAM algorithm can be simulated on an interconnection network with a slowdown that is in the worst case equal to the diameter of the network. Indeed, communication is free on the PRAM, as all the processors can communicate in constant time with any other processor using a shared memory location. In an interconnection network on the other hand arbitrary inter-processor communication takes a time equal to the diameter of the network.

Note however that using PRAM algorithms for interconnection networks will usually result in poor performance compared to algorithms specifically tailored for the particular topology of the network, since the latter algorithms may take advantage of the particularities of the respective network topology. *Data distribution* becomes an important problem in networks.

In what follows we will consider two topologies. One of them (the linear array) is the simplest (and also the poorest choice), whereas the second (the hypercube) offers one of the best compromises between

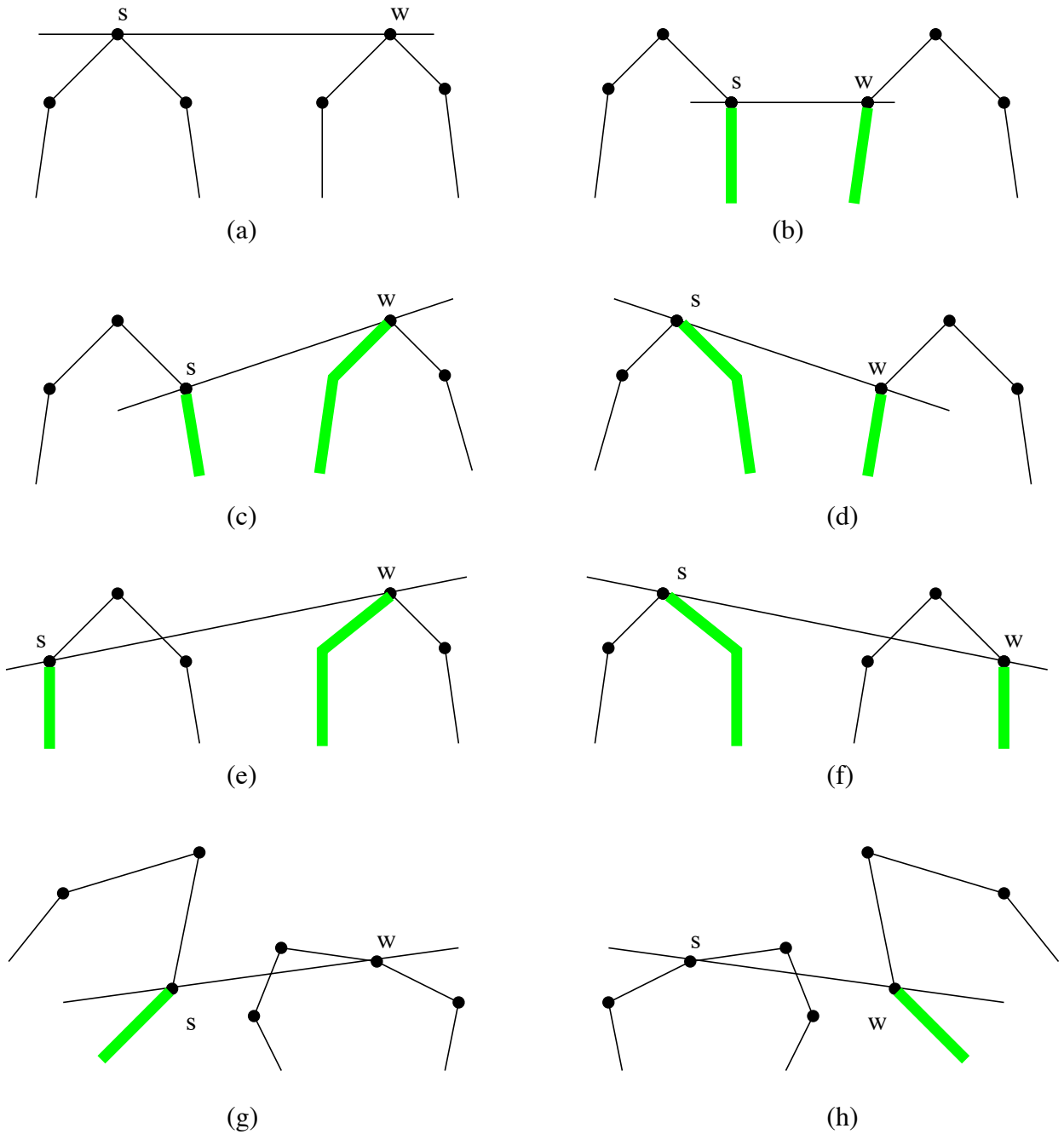


Figure 2: Computing the upper tangent.

connectivity and number of point-to-point connections. In fact many interconnection networks in real-world cluster supercomputers have a hypercube topology.

## 5.1 Linear arrays and sorting

In a *linear array* with $n$ procesors, procesor $P_i$ is (bidirectionally) connected to processor $P_{i+1}$ for all $1 \leqslant i < n$. This is the simplest network topology and also the weakest model. Nonetheless, it performs quite well in certain (kind of limited) situations.

Consider for instance the problem of sorting in increasing order a sequence $S = \langle S_1, S_2, \ldots, S_n \rangle$ which is available *on-line*, meaning that each $S_i$ becomes available at time $i$, $1 \leqslant i \leqslant n$. The lower bound for this problem is $\Omega(n)$ no matter how many processors are available. Indeed, this is how much time it takes for all the data to arrive.

Let us discuss an on-line sorting algorithm on the linear array. We assume that $P_1$ is the "input processor" where the input data becomes available.

An useful basic operation is COMPARE-EXCHANGE$(P_i, P_{i+1})$, which compares the designated values held by $P_i$ and $P_{i+1}$ and possibly exchanges them, so that the smaller value is placed in $P_i$ and the largest in $P_{i+1}$. This operation takes $O(1)$ computation and communication steps and is the basis of sorting by comparison-exchange:

**Algorithm** SORT-COMPARISON-EXCHANGE:

1. $P_1$ reads $S_1$
2. **for** $j = 2$ **to** $n$ **do**
   (a) **for** $i = 1$ **to** $j - 1$ **do in parallel** $P_i$ sends its designated value to $P_{i+1}$
   (b) $P_1$ reads $S_j$
   (c) **for all** odd $i < j$ **do in parallel** COMPARE-EXCHANGE$(P_i, P_{i+1})$
3. **for** $j = 1$ **to** $n$ **do in parallel**
   (a) $P_1$ produces its datum as output
   (b) **for** $i = 2$ **to** $n - j + 1$ **do in parallel** $P_i$ sends its datum to $P_{i-1}$
   (c) **for all** odd $i < n - j$ **do in parallel** COMPARE-EXCHANGE$(P_i, P_{i+1})$

The algorithm takes linear time, which is optimal. However, the cost is $O(n^2)$. You might be thinking that this algorithm bears a striking resemblance with the classical sequential bubble sort algorithm, and if so you would be quite right. The algorithm therefore has all the advantages and disadvantages of bubble sorting.

We can do better by adapting another well known algorithm namely, merge sorting. We maintain the PRAM idea of several merges overlapping with each other. This time the merges are arranged in a real pipeline, which allows us to overlap computation and communication. In fact we need two pipelines, so conceptually we consider that there are *two* links (top and bottom) between processors. The following algorithm performs merge sorting on an $r + 1$ linear array for $r = \log n$:

- Processor $P_1$:
  1. Reads $s_1$ from the input sequence; $i \leftarrow 1$
  2. **for** $i = 2$ **to** $n$ **do**
     (a) **if** $j$ is odd **then** place $s_{i-1}$ on the top link
     (b) **else** place $s_{i-1}$ on the bottom link
     (c) Reads $s_i$ from the input sequence; $i \leftarrow j + 1$
  3. Place $s_n$ on the bottom link
- Processor $P_i$, $2 \leqslant i \leqslant r$:

1. $j \leftarrow 1, k \leftarrow 1$
2. **while** $k < n$ **do**
   **if** the top input buffer contains $2^{i-2}$ values **and** the bottom input buffer contains one value **then**
   (a) **for** $m = 1$ **to** $2^{i-1}$ **do**
      i. Let $x$ be the largest of the first elements from the top and bottom buffers
      ii. Remove $x$ from its buffer
      iii. **if** $j$ is odd **then** place $x$ on the top link
      iv. **else** place $x$ on the bottom link
   (b) $j \leftarrow j + 1, k \leftarrow k + 2^{i-1}$

- Processor $P_{r+1}$:

1. **if** the top input buffer contains $2^{r-1}$ values **and** the bottom input buffer contains one value **then**
   (a) Let $x$ be the largest of the first elements from the top and bottom buffers
   (b) Remove $x$ from its buffer and produce it as output

The processor $P_i$ needs $2^{i-2} + 1$ values so it starts at time $2^{i-2} + 1$ after $P_{i-1}$. $P_i$ produces its first output at time $1 + (2^0 + 1) + (s^1 + 1) + \cdots + (2^{i-2} + 1) = 2^{i-1} + i - 1$ and its last output $n - 1$ time units later. Therefore the running time is $2^r + r + (n - 1) = 2n + \log n - 1 = O(n)$ and the cost is $O(n \log n)$. This is the best known algorithm for on-line sorting on the linear array, but it also uses a sizable amount of space (the implicit buffers on the up and down links). It is questionable if we can consider this much space available.

Note that sorting (on- or off-line) on the linear array with $N$ processors has a lower bound of $\Omega(N)$ time (and so $\Omega(N^2)$ cost) since in the worst case a datum must traverse the diameter of the network (which in this case in $N$) and data has to be distributed to all processors for comparison purposes. It follows that no sorting algorithm can be optimal; that is, the sequential algorithm has the best cost. However, the improved running time is still an advantage in the right circumstance.

Based on the lower bounds explained in the previous paragraph a bubble sort variant is optimal for off-line sorting on the linear array. Here is such an algorithm:

**Algorithm** TRANSPOSITION-SORT:

1. **for** $j = 0$ **to** $N - 1$ **do**
   **for** $i = 0$ **to** $N - 1$ **do in parallel**
   (a) **if** $i \bmod 2 = j \bmod 2$ **then** COMPARE-EXCHANGE$(P_i, P_{i+1})$

## 5.2 The hypercube

The biggest disadvantage of the linear array is that it has the largest possible diameter. The two-dimensional array (or *mesh*) provides a considerably smaller diameter while maintaining many of the nice properties of the linear array: it is simple theoretically and appealing in practice, it features a and small maximum degree for nodes (4), and its topology is regular and modular. The $2N^{1/2} - 2$ diameter, considerably smaller than for the linear array but still quite large.

A great compromise between vertex degree and network diameter is the *hypercube*. This topology is defined as follows: For some integers $i$ and $b$, let $i^{(b)}$ be the number such that the binary representations of $i$ and $i^{(b)}$ differ only in the $b$ position. The processors $P_1, P_2, \ldots, P_N$ for $N = 2^g$, $g \geqslant 1$ are arranged in a *g-dimensional hypercube* whenever each processor $P_i$ is connected to exactly all the processors $P_{i^{(b)}}$, $0 \leqslant b < g$. Both the degree and the diameter of a hypercube with $N$ nodes are $O(\log N)$.

To illustrate the characteristics of hypercube algorithms we consider the problem of matrix multiplication: we are required to compute $c_{jk} = \sum_{i=0}^{n-1} a_{ji} \times b_{ik}$ for $0 \leqslant j, k < n$. The straightforward sequential algorithm runs in $O(n^3)$ running time, while the best known sequential algorithm runs in $O(n^{2+\varepsilon})$ time, $0 < \varepsilon < 0.38$.

For input size $n = 2^g$ we use a hypercube with $N = n^3 = 2^{3g}$ processors. Let the processors be conceptually arranged in an $n \times n \times n$ array such that $P_r$ (or $P_{(i,j,k)}$) occupies position $(i, j, k)$ with $r = in^2 + jn + k$

$$r = \underbrace{r_{3g-1}r_{3g-2}\dots r_{2q}}_{i}\underbrace{r_{2q-1}r_{2q-2}\dots r_{q}}_{j}\underbrace{r_{q-1}r_{q-2}\dots r_{0}}_{k}$$

Each set of processors agrees with each other on one coordinate form a hypercube with $n^2$ processors and on two coordinates form a hypercube with $n$ processors. In oter words, the processors $P_{(i,j,k)}$, $0 \leqslant j, k < n$ form a "layer", with $n$ such layers overall. We use the following designated registers $P_r$ (that is, $P_{(i,j,k)}$): $A_r$, $B_r$, and $C_r$ (or in the alternate notation $A_{(i,j,k)}$, $B_{(i,j,k)}$, and $C_{(i,j,k)}$). Let the input be available $A_{(0,j,k)}$ ($A_{(0,j,k)} = a_{jk}$) and $B_{(0,j,k)}$ ($B_{(0,j,k)} = b_{jk}$). Output will be produced in $C_{(0,j,k)}$ ($C_{(0,j,k)} = c_{jk}$).

**Algorithm** MATRIX-MULT($A = (a_{ij})_{0\leqslant i,j\leqslant n}, B = (b_{ij})_{0\leqslant i,j\leqslant n}$) **returns** $C = (c_{ij})_{0\leqslant i,j\leqslant n}$:

1. *Data distribution*: $A$ and $B$ (layer 0) are distributed to the other processors so that $P_{(i,j,k)}$ stores $a_{ji}$ and $b_{ik}$
   (a) **for** $m = 3g - 1$ **down to** $2g$ **do**
   **for all** $0 \leqslant r < N \wedge r_m = 0$ **do in parallel** $A_{r^{(m)}} \leftarrow A_r$; $B_{r^{(m)}} \leftarrow B_r$
   // result: $A_{(i,j,k)} = a_{jk}$ and $B_{(i,j,k)} = b_{jk}$, $0 \leqslant i < n$
   (b) **for** $m = g - 1$ **down to** $0$ **do**
   **for all** $0 \leqslant r < N \wedge r_m = r_{2g+m}$ **do in parallel** $A_{r^{(m)}} \leftarrow A_r$
   // $A_{(i,j,i)} \rightarrow A_{(i,j,k)}$; result: $A_{(i,j,k)} = a_{ji}$, $0 \leqslant k < n$
   (c) **for** $m = 2g - 1$ **down to** $g$ **do**
   **for ann** $0 \leqslant r < N \wedge r_m = r_{g+m}$ **do in parallel** $B_{r^{(m)}} \leftarrow B_r$
   // $B_{(i,i,k)} \rightarrow B_{(i,j,k)}$; result: $B_{(i,j,k)} = a_{jk}$, $0 \leqslant i < n$
2. *Term computation*: Each $P_{(i,j,k)}$ computes $C_{(i,j,k)} \leftarrow A_{(i,j,k)} \times B_{(i,j,k)}$
   // result: $C_{(i,j,k)} = a_{ji} \times b_{ik}$
3. *Summation*: For $0 \leqslant j, k < n$ compute $C_{(0,j,k)} \leftarrow \sum_{i=0}^{n-1} C_{(i,j,k)}$

The algorithm performs all the arithmetic calculations in constant time, but still need $O(\log n)$ time for data distribution, which is not optimal. However, given the communication overhead a slowdown of $O(\log n)$ is to be expected, and so from this point of view we have an "optimal on the hypercube" algorithm.

Matrix multiplication has numerous applications to graph problems. You probably remember that graphs can be represented by their adjacency matrix: An adjacency matrix $A = (a_{ij})_{0\leqslant i,j<n}$ defines a graph $G = (\{0, 1, \dots, n\}, E)$ whenever $a_{ij} = 1$ if $(i, j) \in E$ and 0 otherwise. This representation is not particularly good for sequential algorithms and the RAM or indeed the PRAM for the simple reason that $A$ is usually sparse in practice and handling sparse matrices is most of the time more trouble than it is worth it. However, the adjacency matrix representation is particularly well suited for being manipulated n the hypercube.

The connectivity matrix $C = (c_{ij})_{0\leqslant i,j<n}$ is defined such that $c_{ij} = 1$ if there exists a path from $i$ to $j$ and 0 otherwise. It can be computed as $C = A'^n$, where $A' = (a'_{ij})_{0\leqslant i,j<n}$ with $a'_{ii} = 1$ and $a'_{ij} = a_{ij}$ for all $i \neq j$. C-style booleans can use plain matrix multiplication, while true booleans require multiplication with $\wedge$ instead of $\times$ and $\vee$ instead of $+$. One way or another, the connectivity matrix can be calculated using matrix multiplication. Algorithmically speaking we can compute $C$ using $O(\log n)$ matrix multiplications; indeed, the graph $C$ is the reflexive and transitive closure of the graph $A$ and so $A'^p = A'^n$ for any $p \geqslant n$.

This kind of repeat multiplication is particularly advantageous for the hypercube since it does not necessitate data redistribution. Indeed, the result of the previous multiplication is in the right place for the next multiplication. It follows that $C$ can be computed on the hypercube with $n^3$ processors and in $O(\log^2 n)$ time.

The connectivity matrix contains several kinds of useful information about the original graph, such as information about the connected components of that graph.

In the same spirit we can compute the all-pairs shortest paths in a graph by computing the matrix $D$ such that $d_{ij}$ is the cost of the shortest path between $i$ and $j$.

We assume as usual no cycles or negative weights so there is no advantage to visit any vertex more than once; also recall that all algorithms for computing shortest paths use the property that any shortest path between two vertices contain shortest paths between the intermediate vertices. Given these properties the shortest paths $d_{ij}^k$ containing at most $k+1$ vertices can be computed inductively as follows:

- $d_{ij}^1 = w_{ij}$ whenever there exists a vertex between $i$ and $j$ and $\infty$ otherwise
- $d_{ij}^k = \min_{0 \leqslant p < n}(d_{ip}^{k/2} + d_{pj}^{k/2})$

$D^k = (d_{ij}^k)_{0 \leqslant i,j < n}$ therefore computable starting from $D^1$ (the matrix of the weights in the original graph) using a special form of matrix multiplication with $+$ instead of $\times$ and min instead of $+$, which can be accomplished on the hypercube with $n^3$ processors in $O(\log^2 n)$ time.

We can go like this all the way to say, minimum-weight spanning trees. In conclusion, matrix representation for graphs is more advantageous on the hypercube than other representations.

# 6 Other interesting network topologies

To finish this section of the course we note in passing that other network topologies also exist. Somehow more exotic ones include binary trees (with degree 3 and diameter $O(\log n)$), which can then be extended to a *mesh of trees*. A mesh of trees features $n^{1/2}$ identical binary trees of $n^{1/2}$ processors. Each set of $n^{1/2}$ "equivalent" processors (in the sense of a preorder traversal) are further linked to form a binary tree. This results in a network with degree 6 and diameter $O(\log n)$.

One may consider the tree topology superior to the hypercube since it has the same diameter but a fixed degree. However, the first degree implies fewer direct links and so data communication is very likely to encounter congestion issues, meaning that most of the data will have to be simultaneously forwarded by a single processor, which cannot realistically happen without considerable delays.

In the *star* topology each processor is labeled with a permutation of $\{1, 2, \ldots, m\}$ (resulting in $m!$ processors for a given $m$). Two processors $P_u$ and $P_v$ are connected with each other whenever the index $v$ can be obtained from the index $u$ by exchanging the first symbol with the $i$-th symbol for some $2 \leqslant i \leqslant m$. The network has an $m-1$ degree $m-1$ and an $O(m)$ diameter.

There is no need to remember this kind of topologies, I only wanted to mention them in case you find the information interesting.