Parallel Algorithms

Gianfranco Bilardi

Department of Information Engineering, University of Padova

Spring 2020

This chapter will focus on the design and analysis of parallel algorithms. Many thousands of parallel algorithms have been developed in the literature and many are currently in use in disparate domains. Here, we will consider only a handful of algorithms, which do solve important problems, but will also help us illustrate general ideas and methodologies[1].

1 Parallel Sorting

Sorting is a classical, basic computational problem. Many sequential as well as many parallel algorithms have been developed for it. We will study one particular algorithm, known as *bitonic sorting*, which is based on the merge-sort strategy and was proposed by Kenneth E. Batcher in 1964[2]. At that time, parallel computers were little more than an idea. Batcher's investigation was mostly motivated by the expectation that progress in electronic circuit technology would have eventually enabled the realization, as part of a computer arithmetic logic unit (ALU), of functional units for sorting sequences up to a certain size, at hardware speed, much faster than in software. The sorting of larger sequences would be done in software, but utilizing the hardware sorters on small subproblems, to accelerate the process. By and large, in spite of tremendous increases of the number of devices that can be placed on a processor chip, the idea of including functional units for sorting in hardware has not become mainstream in general purpose processors (although it has been used in some highly specialized machines). Instead, with the realization of ever larger multiprocessor machines, the bitonic sorting

[1] **Acknowledgment.** Many figures in this chapter have been taken from a report, by then students Ylenia Gravia and Andrea Mattiazzo, who transcribed the lectures of the course "Calcolo Parallelo" taught by G. Bilardi at the University of Padova during Spring 2018. Many thanks to Ylenia and Andrea!

[2] See K.E. Batcher, Sorting networks and their applications, Proceedings of the April 30–May 2, 1968, spring joint computer conference, April 1968, Pages 307314.

algorithm has been the basis for several software solutions[3].

Rather than directly presenting here Batcher's algorithm, we will try to illustrate the process by which one might discover it. A natural starting point is to examine the sequential sorting algorithms we know and evaluate their potential for parallelism. This would be a very instructive exercize for the reader to carry out, for algorithms such as quicksort, heapsort, and merge sort, which are all known to sort a sequence of size N in $O(N \log N)$ comparisons (on average, for quicksort, and in the worst case for the other two).

1.1 Parallelizing Merge Sort

Here, we will consider the merge sort strategy, which is based on the idea of dividing the N input sequence into two halves, recursively sort each half, and then merge the two resulting sorted sequences (see Figure 1).

The first question to ask, when interested in parallelizing a divide and conquer algorithm, is whether the subproblems can be solved concurrently. Here the answer is clearly positive, since sorting one half produces no results that are used in sorting the other half[4].

Let us then examine the impact on performance of executing the sorting of the two halves in parallel. We can easily write the following recurrence for the parallel computation time $T_{sort}(N)$:

$$\begin{aligned} T_{sort}(N) &= T_{sort}(N/2) + T_{merge}(N); \quad (1) \\ T_{sort}(2) &= 1, \quad (2) \end{aligned}$$

where $T_{merge}(N)$ is the time to merge two sorted sequences of $N/2$ elements each. The initial condition reflects the fact that one comparison is sufficient to order two elements. With respect to the sequential case, the improvement is that, in the right hand side, we have $T_{sort}(N/2)$ instead of $2T_{sort}(N/2)$. But what is the impact on the solution? If we use the well known algorithm for merging, with a very good approximation, $T_{merge}(N) = N$. Solving Equation 1 by standard techniques, we obtain $T_{sort} = 2N - 3 =\approx 2N$.

[3] This is just one, small example of a good idea that has proven useful in ways different from those which originally motivated its inventor. The history of mathematics, science, and technology is full of such examples. This is one reason why expanding the frontiers of knowledge is generally productive (beyond the intrinsic value of the intellectual enterprise), even in the absence of an immediate understanding of how that knowledge could be useful.

[4] This (rather stringent) condition is sufficient, but not necessary to enable the concurrent processing of the two subproblems. If the subproblems exchange data, parallelism may still be possible, typically requiring careful synchronization.

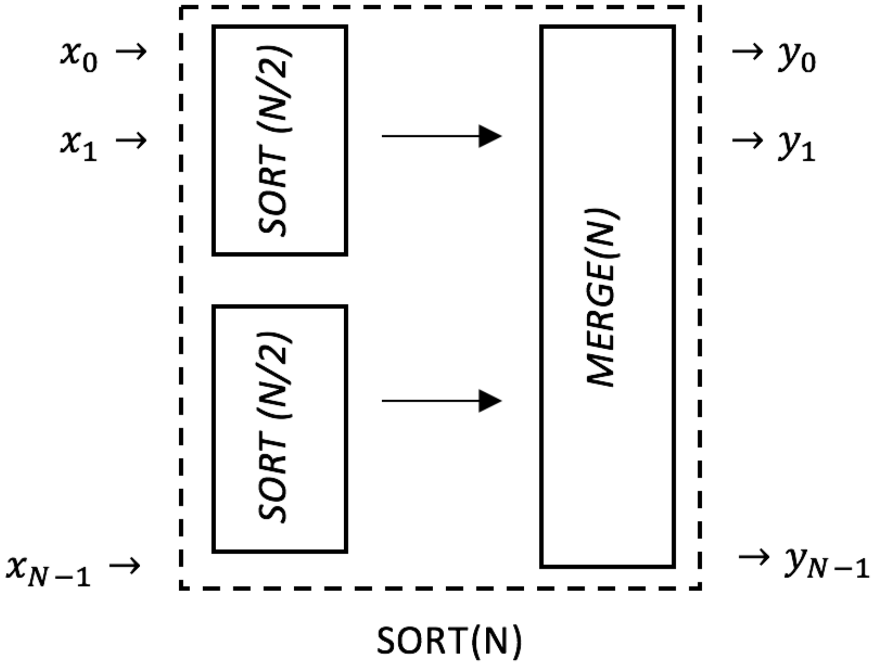
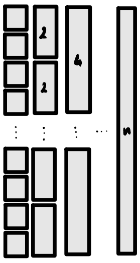


Figure 1: *Merge sort strategy.* An input sequence $\mathbf{x} = (x_0, \ldots, x_{N-1})$ is split into two halves, $(x_0, \ldots, x_{N/2-1})$ and $(x_{N/2}, \ldots, x_{N-1})$. Each half is separately sorted; the two resulting sequences are merged into one sorted sequence $\mathbf{y} = (y_0, \ldots, y_{N-1})$, such that $\mathbf{y} = \mathbf{sort}(\mathbf{x})$. ($N$ is assumed to be even, for simplicity. The broad term "strategy" is meant to suggest that different sorting algorithms can be derived from it, by adopting different merging procedures.)

We have gained a factor $\theta(\log N)$ with respect to the $O(N \log N)$ sequential merge sort. It is interesting to observe that this gain has not required the design of a new algorithm, but just the analysis of some parallelism already present in the standard merge sort. Could we find even more parallelism in the standard algorithm? Upon reflection, we cannot hope to do much better, if we consider that the final merging (associated with the root of the recursion tree of the algorithm), just by itself takes about time N. (Interestingly, the other recursive calls all together also take time N.) Unfortunately, standard merging is a highly sequential algorithm, since which two elements are compared at any given step depends on the relut of all the comparisons performed in the previous steps. The lesson we learn is that, if we want to get an efficient parallel mergesort, we have to find a way to efficiently parallelize merging. This will require the invention of a novel merging algorithm.



1.2 Processor Requirements

Before seaching for a parallel way to merge, let us analize how many processors are needed to execute standard merge sort in time $O(N)$. It is insightful to take a close look at the recursion tree. This tree has $\log_2 N$ levels, which we number $\ell = 0, \ldots, \log_2 N - 1$, from the root ($\ell = 0$) to the leaves ($\ell = \log_2 N - 1$). We can schedule all the 2^ℓ merging operations of size $N/2^\ell$ at level ℓ to be executed concurrently, with the levels executed one at the time, from the leaves to the root. We can recompute the parallel time by summing the time over the levels:

$$\begin{aligned} T_{sort}(N) &= T_{sort}(2) + \sum_{\ell=\log_2 N-2}^{0} T_{merge}(N/2^\ell) \\ &= 1 + \sum_{\ell=\log_2 N-2}^{0} N/2^\ell = 1 + 4 + 8 + \ldots + N = 2N - 3. \end{aligned}$$

We do obtain the same result as above; we just understand it from another angle. Clearly, the schedule we have considered requires 2^ℓ processors at level ℓ, one for each (sequential) merging operation taking place at that level. The processor requirement is maximum at the leaves, where $P = N/2$, to simultaneoulsy compare $N/2$ pairs of elements. However, as we move toward the root, the processor requirement is halved at each level. Intuitively, we are making very poor use of the $N/2$ processors, since the are really all active just in one step and then quickly go out of action. The underlying reason is that the algorithm is very parallel at the leaves, but very sequential at the root.

Then, how parallel standard merge sort really is? From a methodological perspective, we observe that while this question seems to make intuitive sense, it is elusive at this stage, since we have not yet given a specific, let alone quantitative meaning to the notion "degree of parallelism of a given algorithm."

Should we consider the question as ill posed, for lack of a precise definition? Absolutely not, at least if we are interested in making progress on the problem. Some of the most significant advances in science have been made possible by the discovery of a rigorous definition of an hitherto intuitive notion[5]. We will introduce a precise definition of the degree of parallelism of an

[5] Excellent examples are (i) the definition of computability by Alan Turing (1912-1954) in terms of "his" machines; (ii) the quantitative definition of the amount of information by Claude Shannon (1916-2001) in terms of probability; (iii) the definition of π by Archimedes (288 BC-212 BC) in terms of polygons inscribed and circumscribed in a circle; (iv) the

algorithm in a subsequent section. However, we will take advantage of the algorithm at hand to explore the concept.

What we want to capture by the notion of "degree of parallelism" of an algorithm is the possibility to effectively use a certain number of processors to speed up the execution of the algorithm. With P processors, we can hope to speed up by a factor of P. More specifically, if we denote by $T(N,P)$ the time to execute the algorithm on an input of size N using P processors, ideally we would like to obtain $T(N,P) = T(N,1)/P$, that is, to reduce the sequential time by a factor P. Since it may be difficult to keep all the P processors busy all the time, pragmatically we may want to settle for $T(N,P) = O(T(N,1)/P)$; in other words, we could say that the degree of parallelism is at least P if the algorithm can be scheduled in such a way that, on average, each processor executes an operation at least for a fixed percentage of the time.

To apply these considerations to the traditional merge sort, we observe that, in this case (dropping the sort subscript, for readability), we have: $T(N,1) = N\log_2 N$ and $T(N,N/2) = 2N$. Since $P = N/2$ yields a speed up $T(N,1)/T(N,N/2) = (\log_2 N)/2$, we are not willing to say that the degree of parallelism is $N/2$.

Could we achieve a speed up $\theta(\log_2 N)$ by using only $P = \log_2 N$ processors? The answer is positive, as outlined next. Let $\ell_0 = \lfloor \log_2 \log_2 N \rfloor$ so that $(\log_2 N)/2 < 2^{\ell_0} \leq (\log_2 N)$. We can use $P_0 = 2^{\ell_0} \leq \log_2 N$ processors to concurrently execute the subtrees of the recursion tree rooted at level ℓ_0, with each subtree sequentially processed by one processor. Since the size of the subproblems is $N/2^{\ell_0}$, they can be processed in time

$$T_1 = T(N/2^{\ell_0}) = (N/2^{\ell_0})\log_2(N/2^{\ell_0}) \leq 2N/(\log_2 N) * \log_2 N = 2N.$$

The remaining levels of the tree can now be executed one at the time, for $\ell = \ell_0 - 1, \ldots, 1, 0$, using time

$$T_2 = \sum_{\ell=\ell_0-1}^{0} T_{merge}(N/2^{\ell}) \leq 2N.$$

Finally, we have:

epsilon-delta definition of limit of a function by Augustine Cauchy (1789-1857); (v) the definition of infinite sets, by George Cantor (1845-1918), as those that admit a one-to-one correspondence with a proper subset; (vi) the definition of entropy, by Rudoplh Clausius (1822-1888), in terms of heat and temperature. In most of these cases, an informal version of the concept was in use for centuries if not millenia (as for the infinite), before a rigorous counterpart was developed and opened completely new vistas.

$$T(N, P_0) = T_1 + T_2 \leq 4N,$$

corresponding to a speed up

$$T(N,1)/T(N,P_0) \geq (N \log_2 N)/(4N) = (\log_2 N)/4 \geq P_0/4.$$

Intuitively, we have scheduled standard mergesort on P_0 processors which, on average, perform useful operation at least 25% of the time. We conclude that standard merge sort has a degree of parallelism at least $P_0 = 2^{\lfloor \log_2 \log_2 N \rfloor}$, whence $(\log_2 N)/2 \leq P_0 \leq (\log_2 N)$.

1.3 Bitonic Merging

The bitonic merging algorithm is based on the concept of bitonic sequence, which we introduce next.

Definition 1 *Let* $\mathbf{x} = (x_0, \ldots, x_{N-1})$ *be a sequence of elements from a totally ordered universe*[6]*. We say that* $\mathbf{x}$ *is an* up-down *sequence if it consists of a non decreasing prefix followed by a non increasing suffix. We say that* $\mathbf{x}$ *is a* bitonic *sequence if it is the cyclic shift*[7] *of some up-down sequence* $\mathbf{y}$.

A few examples and observations will help us familiarize with the concepts just introduced. Sequence $(0, 3, 4, 7, 6, 5, 2, 1)$ is up-down; its 3-rd left cyclic shift, $(7, 6, 5, 2, 1, 0, 3, 4)$, is bitonic. A non decreasing sequence is special case of up-down sequence (empty suffix); similarly for a non increasing sequence (empty prefix). An up-down sequence is trivially bitonic (0-th shift). See also Problems 1.2.1, 1.2.2, and 1.2.3, to gain further understanding.

What is the relevance of bitonic sequences in the context of merging? Batcher's bitonic merging algorithm is an algorithm that actually sorts all bitonic sequences. Given two sorted sequences $\mathbf{z_1}$ and $\mathbf{z_2}$, their concatenation $\mathbf{x} = \mathbf{z_1}\mathbf{z_2}^R$, where the superscript R denotes reversal,[8] is bitonic (in fact, up-down); therefore, sorting $\mathbf{x}$ is equivalent to merging $\mathbf{z_1}$ and $\mathbf{z_2}$. Notice that the two sequences can be of different length.

We will focus on bitonic sequences whose length is a power of 2, $N = 2^d$. Bitonic merging is a divide and conquer algorithm. The divide step produces two bitonic subsequences of length $N/2$, using the operation defined next.

[6] A universe is *totally ordered* if it is endowed with a binary relation "<" which is asymmetric and transitive and such that, whenever $a \neq b$, either $a < b$ or $b < a$ holds.

[7] We say that $\mathbf{x}$ is the k-th *left cyclic shift* of $\mathbf{y}$ if $x_i = y_{(i+k) \mod N}$, for $i = 0, \ldots, N-1$. The k-th *right cyclic shift* can by defined symmetrically, and equals the $(N-k)$-th left cyclic shift.

[8] The *reversal* $\mathbf{y}$ of a sequence $\mathbf{y} = (y_0, \ldots, y_{N-1})$ is defined as $\mathbf{y}^R = (y_{N-1}, \ldots, y_0)$.

Definition 2 *Let $\mathbf{x} = (x_0, \ldots, x_{N-1})$ be a sequence of elements from a totally ordered universe, with N even. We introduce the operators L and U, each producing a sequence of length $N/2$, as follows:*

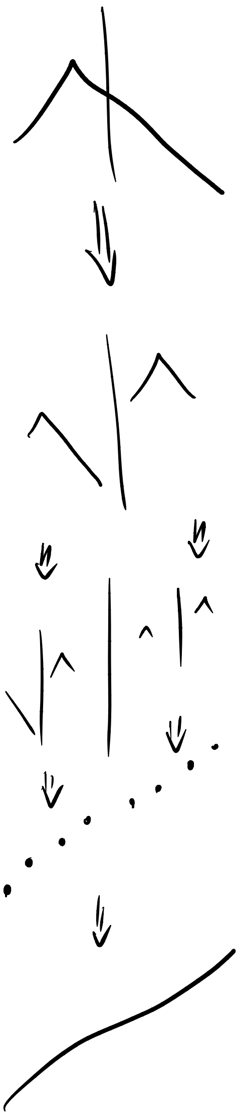
$$L\mathbf{x} = (\min(x_0, x_{N/2}), \ldots, \min(x_{N/2-1}, x_{N-1})); \tag{3}$$
$$U\mathbf{x} = (\max(x_0, x_{N/2}), \ldots, \max(x_{N/2-1}, x_{N-1})). \tag{4}$$

The seed idea of bitonic merging sparks from an acute observation on what happens when the L/U operation is applied to a bitonic sequence. We first illustrate the idea by an example. Cosider the following bitonic sequence of length $N = 16$:

$$\mathbf{x} = (10, 13, 14, 18, 19, 20, 21, 24, 25, 23, 22, 17, 16, 15, 12, 11).$$

We now obtain $L\mathbf{x}$ and $U\mathbf{x}$, writing the second half of $\mathbf{x}$ below the first half, and then computing the minimum and the maximum of each pair aligned vertically:

$$(x_0, \ldots, x_7) = (10, 13, 14, 18, 19, 20, 21, 24) \tag{5}$$
$$(x_8, \ldots, x_{15}) = (25, 23, 22, 17, 16, 15, 12, 11) \tag{6}$$
$$L\mathbf{x} = (10, 13, 14, 17, 16, 15, 12, 11) \tag{7}$$
$$U\mathbf{x} = (25, 23, 22, 18, 19, 20, 21, 24). \tag{8}$$



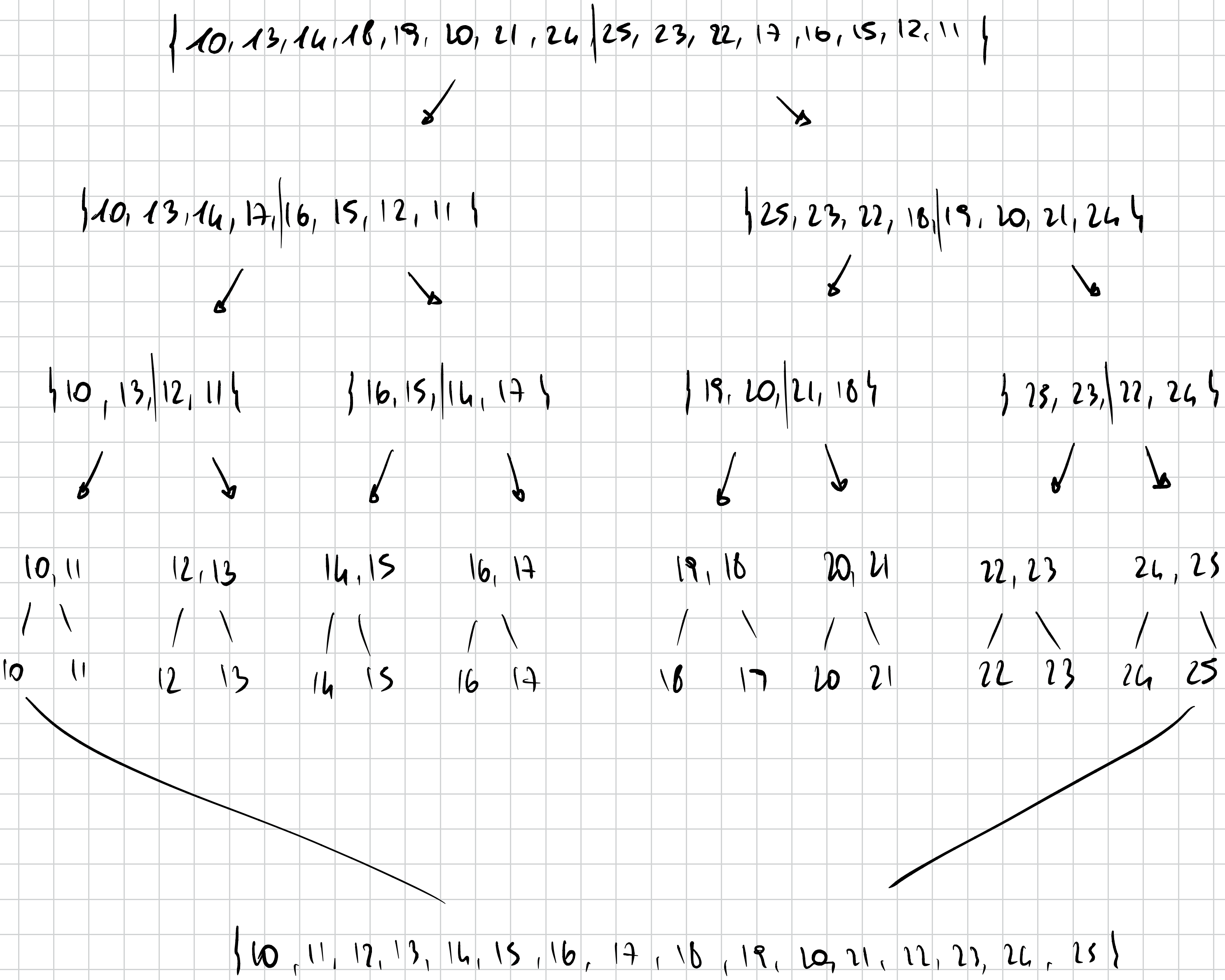
First, observe that each elements of $L\mathbf{x}$ is not larger than each element of $U\mathbf{x}$, that is, $\max(L\mathbf{x}) \leq \min(U\mathbf{x})$. Specifically, $\max(L\mathbf{x}) = 17$ and $\min(U\mathbf{x}) = 18$. In other words, the elements have been partitioned into two halves separated by value (like in quicksort, except that here the result does not depend on comparing every element to a pivot, but on properties of bitonic sequences).

Second, $L\mathbf{x}$ and $U\mathbf{x}$ are both bitonic sequences. In the example, $L\mathbf{x}$ is up-down and $U\mathbf{x}$ an 8-shift of an up-down sequence. This property is very useful, as it enables to recursively refine the partition, until it blocks have size 1, yielding a sorted sequence.

The properties illustrated in the above example hold in general, as stated by the following theorem.

Theorem 1 (L/U operation on bitonic sequences.) *Let $\mathbf{x} = (x_0, \ldots, x_{N-1})$ be a bitonic sequence of elements from a totally ordered universe, with N even. Then,*

- $\max(L\mathbf{x}) \leq \min(U\mathbf{x})$, *and*
- *$L\mathbf{x}$ and $U\mathbf{x}$ are bitonic.*


{10, 13, 14, 18, 19, 20, 21, 24 | 25, 23, 22, 17, 16, 15, 12, 11}
{10, 13, 14, 17, | 16, 15, 12, 11}
{25, 23, 22, 18, | 19, 20, 21, 24}
{10, 13, | 12, 11}
{16, 15, | 14, 17}
{19, 20, | 21, 18}
{25, 23, | 22, 24}
10, 11
12, 13
14, 15
16, 17
19, 18
20, 21
22, 23
24, 25
10
11
12
13
14
15
16
17
18
17
20
21
22
23
24
25
{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25}

IT'S SORTED!

Based on this theorem, and given the preceding considerations, we obtain the following algorithm to sort any bitonic sequence, hence to merge.

BM - Bitonic Merging Algorithm

```
Input: $\mathbf{x} = (x_0, \ldots, x_{N-1})$, with $N = 2^d$.
Output: $BM(\mathbf{x})$.
If $N = 1$, then $BM(\mathbf{x}) = \mathbf{x}$;
else $BM(\mathbf{x}) = BM(L\mathbf{x})BM(U\mathbf{x})$.
```

The following result is a straightforward corollary of Theorem 1.

Proposition 1 (BM algorithm - sorting properties.) *If $\mathbf{x}$ is a bitonic sequence of $N = 2^d$ elements, the the output $BM(\mathbf{x})$ of the bitonic merging algorithm is* $\mathbf{sort}(\mathbf{x})$*, the sorted version of* $\mathbf{x}$[9] .

Intuitively, algorithm BM is highly parallel, since all the $N/2$ comparisons involved in the L/U computation can be executed concurrently, in just one parallel step, if $N/2$ comparators are available. Moreover, the two recursive calls, $BM(L\mathbf{x})$ and $BM(U\mathbf{x})$, do not exchange data and can proceed concurrently as well. See Figure 2, for a schematic overview.

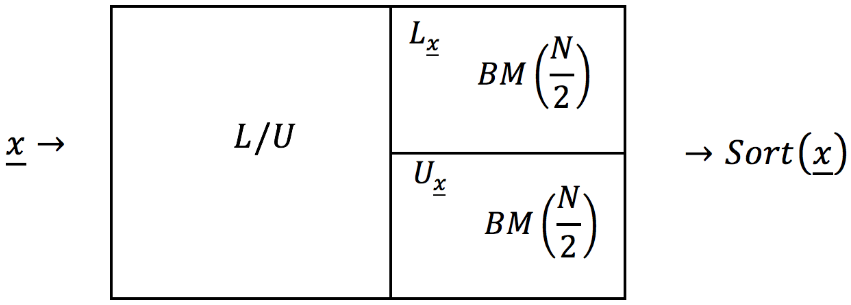


Figure 2: *Recursive view of BM algorithm.* From the input sequence $\mathbf{x} = (x_0, \ldots, x_{N-1})$, the BM algorithm produces two subsequences, $L\mathbf{x}$ and $U\mathbf{x}$, of length $N/2$, which are recursively processed by the same algorithm. When $\mathbf{x}$ is bitonic, the concatenation of the outputs of the two subsequences is $\mathbf{sort}(\mathbf{x})$, the sorted version of $\mathbf{x}$.

[9] That $\mathbf{x}$ is a bitonic sequence is a sufficient condition for $BM(\mathbf{x})$ to be sorted, but it is not necessary. In fact, it can be shown that the permutations of order N sorted by BM are $\sqrt{N^N}$, considerably fewer than $N!$, the number of of all permutations, but considerably more than the $B(N) = N2^{N-2}$ bitonic permutations (see Problem 1.3.2.)

By an argument of the same type than the one leading to Equation 1, we can write the recurrence relation for the parallel execution time of algorithm BM:

$$
\begin{aligned}
T_{BM}(N) &= \overset{\text{BM on } L_x, U_x}{\uparrow}\!\!\! T_{BM}(N/2) + \overset{\text{Concatenation} \hookrightarrow O(1)}{\nearrow}\!\!\! T_{LU}(N); && (9)\\
T_{BM}(2) &= 1. && (10)
\end{aligned}
$$

Considering that $T_{LU}(N) = 1$, the solution is $T_{BM}(N) = \log_2 N$, as it can be easily checked.

1.4 Comparator Networks and Bitonic Merging

We now take a look at Batcher's original intention to implement merging and sorting in harware. This will give us a better understanding of the structure of the BM algorithm, in ways that will prove very useful even in different chapters.

We call *comparison exchange* an operation two input values, a and b, and two output values, $\min(a, b)$ and $\max(a, b)$. It is customary to graphically represent a unit performing this operation as shown in Figure 3.

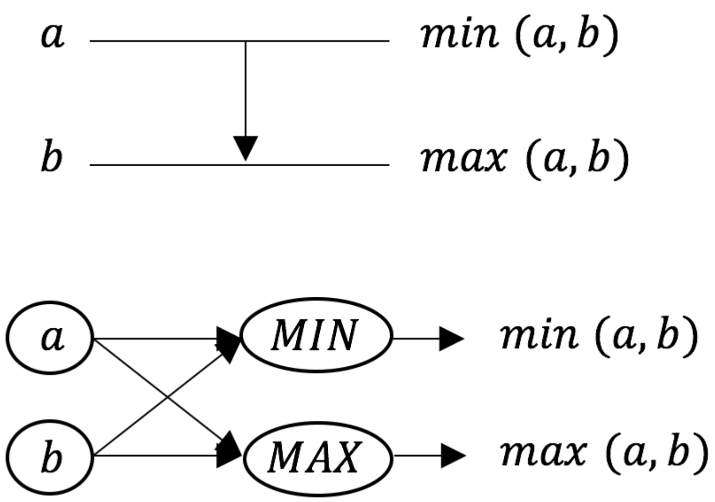


Figure 3: *Symbol and function of a comparator exchanger.* The top part of the figure shows the standard diagram used when drawing comparator networks. The bottom part is used when two-input, one-output operations are preferred.

Equipped with this notation, we now draw the bitonic merging network of comparator exchangers, as illustrated in Figure 4. This is obtained from the recursive scheme of Figure 2, by unfolding the recursion and by representing an L/U operator with $N/2^i$ inputs as a set of $N/2^{i+1}$ comparator exchangers. The network is represented by N horizontal lines, connected (in pairs) by comparator exchangers.

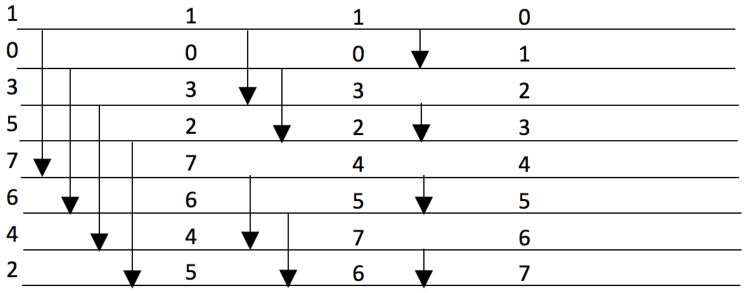


Figure 4: *The bitonic merging network $BM(8)$.* The network has $N = 8$ inputs which "travel" on as many horizontal lines, connected by comparisons exchangers that can exchange the lines of the elements. The lines can be viewed as numbered $0, 1, \ldots, N-1$ from top to bottom. The operation of the network is illustrated on the bitonic input sequence $\mathbf{x} = (x_0, \ldots, x_{N-1}) = (1, 0, 3, 5, 7, 6, 4, 2)$, with input x_i initially placed on line i. As we can see, the sequence produced at the output is sorted, in non decreasing order from top to bottom. With a little thought, one can recognize that the sequences shown vertically in the figure, respectively are $\mathbf{x}$, $L\mathbf{x}U\mathbf{x}$, $LL\mathbf{x}UL\mathbf{x}LU\mathbf{x}UU\mathbf{x}$, and $LLL\mathbf{x}ULL\mathbf{x}LUL\mathbf{x}UUL\mathbf{x}LLU\mathbf{x}ULU\mathbf{x}LUU\mathbf{x}UUU\mathbf{x}$.

One interesting property of the BM algorithms is that, once the input size N is fixed, the connections between the various comparator exchange operations are independent of the specific values of the input. This property makes the algorithm suitable for direct hardware implementation, because the connections among comparators can be hardwired once and for all.[10]

It is not difficult to see that, in general, $BM(N)$ consists of $\log_2 N$ stages, each containing $N/2$ parallel comparator exchangers. The total number of comparisons is $C_{BM}(N) = (1/2)N \log_2 N$.

Proposition 2 (BM algorithm - performance.) *The BM algorithms requires $C_{BM}(N) = (1/2)N \log_2 N$ comparisons, which can be executed in*

[10] Many algorithms do not enjoy such a property. A well known example is quicksort, where even the number of comparisons - let alone their connections - can vary significantly between different inputs of the same size.

time $T_{BM}(N) = \log_2 N$, *on* $P = N$ *processors*[11].

We observe that, in terms of the total number of operations, bitonic merging is suboptimal, since we know (from the standard merging algorithm) that N comparisons are sufficient. The great advantage of bitonic merging is, of course, its high degree of parallelism.

The bitonic approach has been refined to obtain an algorithm, known as *Adaptive Bitonic Merging (ABM)*, requiring $C_{ABM}(N) = O(N)$ comparisons, which can be executed in time $T_{ABM}(N) = O(\log_2 N)$, on $P_{ABM}(N) = O(N/\log_2 N)$ processors; here N is not restricted to be a power of 2.[12] This algorithm will not be discussed in these notes, but it is mentioned to show that the linear bound on comparisons is indeed compatible with very fast, $O(\log_2 N)$ time solution of the merging problem. The qualification "adaptive", in ABM, highlights the fact that the comparisons adapt to the input, therefore ABM does not define a comparator network. It can be proven that any network for merging has at least $(1/4)N \log_2 N$ comparators.[13]

1.5 Bitonic Sorting

We are now ready to examine the merge sort strategy based on bitonic merging, schematically shown in Figure 5. The corresponding algorithm is kwnon as *Bitonic Sorting (BS)*. The analysis yields the results summarized in the next proposition.

Proposition 3 (BS algorithm - performance.) *The BS algorithms requires* $C_{BS}(N) = (1/4)N(\log_2 N + 1)\log_2 N$ *comparisons, which can be executed in time* $T_{BS}(N) = (1/2)(\log_2 N + 1)\log_2 N$, *on* $P = N$ *processors.*

Proof: The recurrence relation for time is

$$\begin{aligned} T_{BS}(N) &= T_{BS}(N/2) + T_{BM}(N), && (11)\\ T_{BS}(2) &= 1; && (12)\\ T_{BM}(N) &= \log_2 N. && (13) \end{aligned}$$

We can solve the recurrence by unfolding:

[11] It is assumed here that each comparison exchange is carried out by two processors, one computing the min and the other the max of the two elements under comparison.

[12] See G. Bilardi and A. Nicolau, Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines, SIAM Journal on Computing 18 (2), 216-228 (1989).

[13] A result established by R.W. Floyd and reported in D.E. Knuth, The Art of Computer Programming: Volume 3: Sorting and Searching, p.230, Addison Wesley 1998.

$$\begin{aligned} T_{BS}(N) &= T_{BM}(2) + T_{BM}(4) + \ldots + T_{BM}(N) && (14) \\ &= \sum_{j=1}^{\log_2 N} T_{BM}(2^j) = \sum_{j=1}^{\log_2 N} j = (1/2)(\log_2 N + 1)\log_2 N. && (15) \end{aligned}$$

Since, at each time step, the number of comparisons is $N/2$, the total number of comparisons is $C_{BS}(N) = T_{BS}(N)(N/2) = (1/4)N(\log_2 N + 1)\log_2 N$.

By assigning all the min and max operations incident on a given line of the network to the same processor, $P = N$ processors can clearly execute each stage of comparisons in one parallel step. *QED*

Observations similar to those made for bitonic merging also apply to bitonic sorting, in particular the logarithmic suboptimality in the number of comparisons. This suboptimality is overcome by using ABM, which leads to the *adaptive bitonic sorting* algorithm. ABS performs $C_{ABS}(N) = O(N \log_2 N)$ comparisons, which can be executed in time $T_{ABS}(N) = O((\log_2 N)^2)$, on $P_{ABS}(N) = O(N/\log_2 N)$ processors.

Interestingly, and perhaps surprisingly, there exists a sorting network with $O(N \log_2 N)$ comparator exchangers organized into $O(\log_2 N)$ stages. It is called the AKS sorting network, from the initials of the authors.[14] This network is mostly of theoretical interest, since the constant factors hidden bythe O notation are very large, for any practical purpose.

[14] Ajtai M., Komlos J., Szemeredi E. (1983) Sorting in $c \log n$ steps. Combinatorica 3:119.

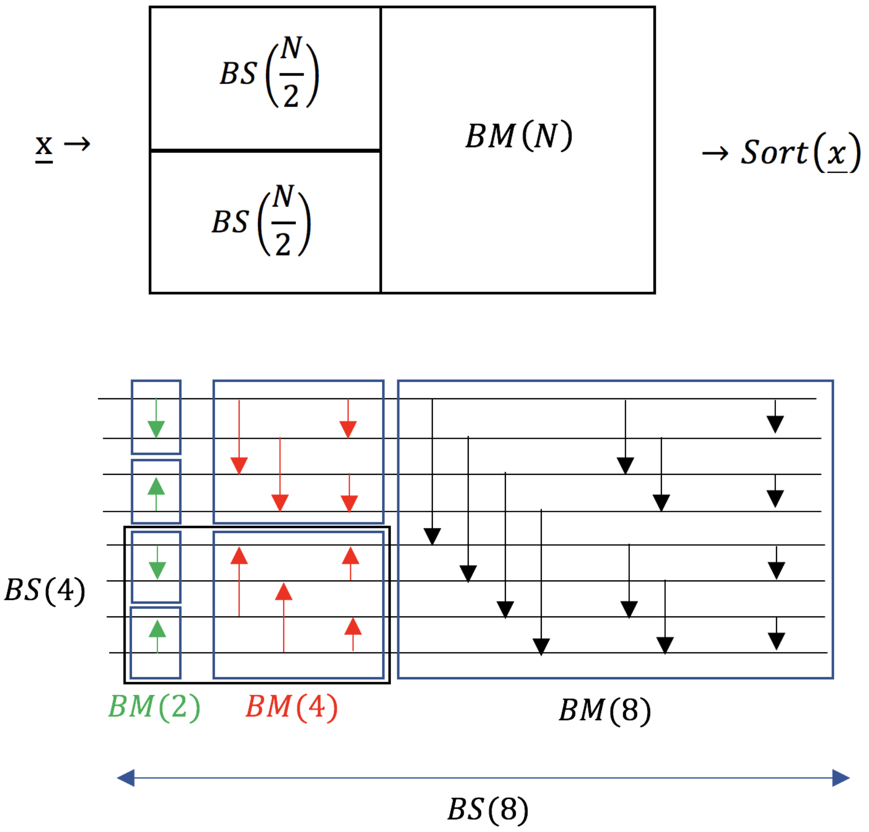


Figure 5: *Recursive view of BS algorithm (top) and the bitonic sorting network $BS(8)$.* The BS algorithm splits the sequence $\mathbf{x} = (x_0, \ldots, x_{N-1})$ into two subsequences each of length $N/2$, which are recursively sorted and then merged by the BM algorithm, outputting $\mathbf{sort}(\mathbf{x})$, the sorted version of $\mathbf{x}$. The full network is shown in the special case $N = 8$. The direction of the comparator exchangers ensures that the input to each merging stage is an up-down (hence bitonic) sequence.

2 Measuring Parallelism in Algorithms

In this section, our objective is to develop a quantitative metric that can reasonably capture the degree of parallelism of an algorithm. As for time, space, and other performance metrics of an algorithm, even the degree of parallelism should be expected to depend upon the input of the algorithm. For example, we have seen in Section 1.5 how, for the bitonic sorting algorithm, parallelism equals the number N of elements to be sorted. For more dynamic algorithms, the degree of parallelism may differ even between two inputs of the same size. Keeping these considerations into account, we will focus on the *execution* of an algorithm on a specific input, and refer to it as to a *computation.*

Intuitively, the degree of parallelism, which we will denote by P^*, should be related to the largest value of P such that the computation can be scheduled on P processors to complete in T steps, with each processor executing at least $\gamma\, T$ operations, where $\gamma \in (0, 1]$ is some chosen constant.[15]

It is then natural to investigate which properties of a computation can limit its parallelism. The key reason why two operations A and B cannot be executed concurrently is that one of them, say B, uses as an operand a result that is (directly or indirectly) influenced by A. This concept plays a key role and deserves a precise definition.

Definition 3 *We say that a sequence of operations $A = A_1, \ldots, A_\ell = B$, within a given computation, forms a* data dependence chain *of* length ℓ *if, for every $j = 2, \ldots, \ell$, the result of A_{j-1} is an operand of A_j. We will also say that operation B is* data dependent *upon operation A.*

It is clear that each operation in a data-dependence chain can only be executed after the previous one has completed. Hence, if a computation contains a data dependence chain of length ℓ, its execution time must satisfy $T \geq \ell$. Thus, we are interested in determining the length L of the longest data dependence chain of a given computation, yielding $T \geq L$.

2.1 The Computation DAG

To systematically explore the dependence chains of a computation, it is convenient to represent such a computation as a directed graph, where operations are represented by nodes and direct dependences by arcs. Dependence chains

[15] Technically, such a value will depend upon γ and should be more precisely denoted as P^*_γ. The analysis developed in this section will provide us with the necessary insight to choose a reasonable value for γ.

will form paths in this graph. This is the informal intuition behind the next definition, which also models inputs and outputs of a computition.

Definition 4 *A* computational directed acyclic graph (CDAG) *is a sextuple* $C = (I, V, E, O, \lambda, \xi)$ *where*[16]*:*

1. *I is the* set of input nodes*; (each node in I represents a value that is provided as an input to the computation).*

2. *V is the* set of operation nodes*; (each node in V represents an operation executed as part of the computation).*

3. *$I \cap V = \emptyset$; (a node represents either an input value or (a value produced by) an operation).*

4. *$\lambda: I + V \rightarrow$* operation types*, associates a specific operation with each node in $I+V$, producing exactly one result; (for example, input(integer),* min*, $+$, $\times$, $NAND$, ...).*

5. *$O \subseteq (I + V)$ is the* set of output nodes*; (a node is included in O if the corresponding value is considered to be an output of the computation).*

6. *$E \subseteq (I + V) \times V$ is the* the set of arcs*; (an arc (u, v) indicates that the value produced by $u \in (I + V)$ - via input or other type of operation - is an operand of operation $v \in V$. Input nodes do not have operands, hence have no incoming arcs - their indegree is zero).*

7. *$(I + V, E)$ is a* directed acyclic graph (DAG)*; (cycles are ruled out to avoid that an operation could be data dependent upon itself).*

8. *$\xi: V \rightarrow$* permutations of arcs to *v, specifies the order of the operands for any operation v; (this is necessary for non commutative operations).*

9. type consistency*: if (u, v) is the k-th arc to node v, then the type of the result of operation u must be the same as the type of the k-th input of operations v.*

10. *If $u \in (I+V)$, then u lies on at least one path from some $w \in I$ to some $z \in O$; (this simplifying requirement rules nodes that are not reachable from any input or cannot reach any output).*

[16] Below, the notations $R + S$ is used to denote the union of two disjoint sets R and S.

The above definition is somewhat pedantic, to ensure a precise meaning to the concept and also its applicability to a variety of situations, beyond our immediate interest in defining the degree of parallelism of a computation. For the latter purpose, the nature of the operations associated to the various nodes is not crucial, as long as we assume, as we shall do for simplicity, that they all take the same time. The structure of a CDAG relevant to parallelism is then captured by the DAG $(I + V, E)$.

2.2 The Greedy Schedule of a CDAG

How fast can we execute a given CDAG? Before formulating an answer we need to better frame the question, by specifying the rules for the execution. Toward this goal, we introduce the following rather idelized machine.

Definition 5 *The* Ideal Machine*, $IM(P)$, is a system with P processors and a shared memory of unbounded capacity. A computation can be executed in a sequence of steps such that, in one step, each processor can execute any operation of a given repertoire, reading the operands from anywhere in memory and writing the result anywhere in memory.*

The machine just defined is idealized, in several respects. In real machines, different operations may take different numbers of machine cycles. Also, individual processing elements would correspond to the functional units of a processor, which can work concurrently, but typically have a specialized repertoire of operations (e.g., integer addition, floating point multiplication, ...).[17] More importantly, as already stressed in Chapter 1, in a large system, reading a value from anywhere in memory can take hundreds or even thousands of cycles. The ideal machine model completely ignores the delays due to communication. This is appropriate in the current context, where we are interested in characterizing the intrinsic parallelism of a given algorithm; in later chapters, we will introduce more realistic models, to more accurately account for the cost of communication in real systems.

We can now reformulate the initial question by asking how fast can a given DAG $G = (I, V, E)$ be executed on $IM(P)$, assuming P to be large enough not to pose any active constraint on execution time.[18]. We further assume that initially, say at time $t = 0$, the input value corresponding to each node in $V_0 = I$ is available in memory. Which nodes can be executed

[17] To further complicate things, functional units are typically pipelined, so as to start a new operation at each cycle, although the result will be available only a few cycles later.

[18] For example, we could assume $P = |V|$, which would suffice to execute all operations of G concurrently.

at time $t = 1$? Clearly, all those nodes whose predecessors are all in V_0; let us call this set V_1. Then, at time $t = 2$, the (new) nodes that can be executed are all those whose predecessors are in $V_0 + V_1$; let us call this set V_2. It is interesting to observe that a node in V_2 must necessarily have at least one predecessor in V_1, otherwise it could have been executed at the previuos step and would be in V_1, rather than in V_2. Continuing this way, we can construct a sequence of disjoint sets, $V_0 = I, V_1, \ldots, V_L$ ending when all nodes have been included, that is when $\sum_{j=0}^{L} V_j = (I + V)$. See Figure 6 for an illustration. The resulting schedule is called the greedy schedule, because each operation is scheduled "greadily," as soon as possible. We package these ideas in the following definition.

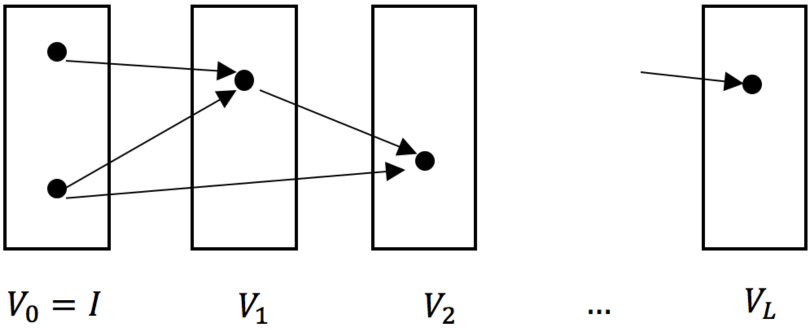


Figure 6: *Greedy schedule of a DAG.* Each operation is scheduled for execution as soon as possible.

Definition 6 *Given a DAG $G = (I, V, E)$, as part of a CDAG as specified in Definition 4, we call* greedy schedule *the sequence $(V_0, V_1, \ldots, V_L)$ of disjoint subsets of $I + V$ such that:*

1. *$V_0 = I$;*
2. *For $j > 0$, V_j is the set of nodes $v \in (V - (V_0 + \ldots + V_{j-1}))$ such that, if $(u, v) \in E$, then $u \in (V_0 + \ldots + V_{j-1})$;*
3. *$\sum_{j=0}^{L} V_j = (I + V)$.*

Positive integer L is called the critical length *of G. The sets $V_1, V_2, \ldots, V_L$ are called the* levels *of the greedy schedule.*

The greedy schedule can be characterized in terms of paths (modelling dependence chains), as stated in the following proposition.

Proposition 4 *With reference to Definition 6, let $v \in I + V$. Then, $v \in V_j$ if and only if the longest paths from input nodes in I to v have length j*[19].

> **Proof:** We proceed by induction on j. The base case, for $j = 0$, is immediate. In fact, if $v \in V_0 = I$, then the only path starting in I and ending at v consists of just node v, with no arcs, whence its length is $j = 0$. Conversely, if the longest path starting in I and ending at v has length 0, then v is the only node on such path, hence it is its starting node, which by assumption belongs to $I = V_0$.
>
> For the inductive step, let now $j > 0$. We know that if $v \in V_j$, then v has an immediate predecessor u in V_{j-1}. That is, there is a $u \in V_{j-1}$ such that $(u, v) \in E$. By the inductive hypothesis applied to $(j - 1)$, there is a path σ, from I to u, of length $(j - 1)$. Then, extending σ with arc (u, v) yields a path $\tau = \sigma(u, v)$, of length $(j - 1) + 1 = j$, from I to v. We now observe that, in the greedy schedule, if $(y, z) \in E$, with $y \in V_h$ and $z \in V_k$, then $k \geq h + 1$. In other words, any arc makes the level number increase by at least 1. As a consequence, no path with more than j arcs could go from I to V_j, whence τ is a longest path from I to v. *QED*

From the greedy schedule, we now obtain a number of interesting lower and upper bounds on the minimum time $T(P)$ to execute the CDAG on $IM(P)$. These bounds will also help arrive at a satisfactory definition of degree of parallelism.

Theorem 2 *For any $P \geq 1$, the* minimum time *$T(P)$ to execute a CDAG $G = (I, V, E)$ on the ideal machine $IM(P)$ satisfies the* work lower bound*:*

$$T(P) \geq |V|/P, \tag{16}$$

as well as the critical length lower bound*:*

$$T(P) \geq L, \tag{17}$$

where L is the critical length of G, which equals the number of levels of the greedy schedule of G.

If $P_{max} = \max_{j=1}^{L} |V_j|$ denotes the maximum size of any level of the greedy schedule, then

$$T(P_{max}) = L. \tag{18}$$

Furthermore, for any $P \geq 1$,

$$\max(|V|/P, L) \leq T(P) < |V|/P + L \leq 2\max(|V|/P, L). \tag{19}$$

[19] We recall that the length of a path equals the number of its arcs.

Proof: *Part I*: $T(P) \geq |V|/P$. This bound is a simple consequence of the fact that $|V|$ operations must be executed in the computation represented by G and that $IM(P)$ can execute at most P operations per step.

Part II: $T(P) \geq L$. By Proposition 4, there are paths in G, from $V_0 = I$ to V_L, of length L. Since any such path represents a data dependence chains, at most one operation on it can be executed at any given time, regardless of how many processors are available. Therefore, for any P, $T(P) \geq L$.

Part III: $T(P_{max}) = L$. Given P_{max} processors, for $t = 1, 2, \ldots, L$, we can (greedily) schedule at time t all the operations in V_t, since $|V_t| \leq P_{max}$, by definition of P_{max}. Then, $T(P_{max}) \leq L$[20]. By combining this inequality with Inequality 17, we conclude that $T(P_{max}) = L$.

Part IV: $T(P) < |V|/P + L$, for any $P \geq 1$. We now consider modifying the greedy schedule in a way it can be executed with $P \leq P_{max}$ processors, by partitioning each V_j into $q_j = \lceil |V_j|/P \rceil$ disjoint subsets, $(q_j - 1)$ of which have size P and 1 has size $|V_j| - (q_j - 1)P \leq P$. The nodes in each subset are executed concurrently; this is feasible, since there are no data dependencies within V_j, hence within any of its subsets, and there are at least as many processors as there are operations in a subset. The subsets are executed one at the time. We can then analize the execution time as follows:

$$T(P) \leq \sum_{j=1}^{L} q_j = \sum_{j=1}^{L} \lceil |V_j|/P \rceil < \sum_{j=1}^{L} (|V_j|/P + 1) = |V|/P + L.$$

The first step follows from the fact that we are exhibiting a P-processor schedule with time $\sum_{j=1}^{L} q_j$, an upper bound to the minimum time. The second step follows from the definition of q_j. The third step uses the fact that $\lceil x \rceil < x + 1$, for any real number x. The fourth and last step uses the equality $\sum_{j=1}^{L} |V_j| = |V|$, due to $V_1, \ldots, V_L$ forming a partition of V (as implied by Property 3 of Definition 6, considering that $I = V_0$), as well as the straightforward equality $\sum_{j=1}^{L} 1 = L$.

Part V: The relationship $\max(|V|/P, L) \leq T(P)$ simply combines lower bounds 16 and 17. The relationship $|V|/P + L \leq 2\max(|V|/P, L)$ follows from the fact that the sum of two numbers a and b is always smaller than twice their maximum $\max(a, b)$, that is, $a + b \leq 2\max(a, b)$. (Equivalently, the maximum is no smaller than the average: $\max(a, b) \leq (a + b)/2$.) *QED*

[20] Recall that $T(P_{max})$ is, by definition, the minimum time, over all schedules possible with P_{max} processors. Having established that the greedy schedule takes time L, we know that $T(P_{max})$ will not exceed L, but it could be smaller than L, in the presence of a faster schedule. This possibility is ruled out by Inequality 17.

Remarks. A number of observations can help better interpreting the meaning of the previous theorem.

1. In general, the relashionships in 19 characterize $T(P)$ within a factor of two.

2. For $P = P_{max}$, Equation 18 characterizes $T(P)$ exactly, as $T(P_{max}) = L$. It is easy to argue that, for any $P \geq P_{max}$, $T(P) = L$.

3. Tracing the developments of the proof for specific DAGs and specific values of P, can yield tighter bounds. For example, if all levels of the greedy schedule are exact multiples of P, then $T(P) = |V|/P$. In particular, $T(1) = |V|$.

4. The value $P^* = |V|/L$ makes the work term, $|V|/P$, equal to the critical length term, L. We have

$$L \leq T(P^*) < 2L = 2|V|/P^*. \tag{20}$$

   Thus, using P^* processors[21] achieves a computation time less than double of the absolute minimum (with any number of processors). Moreover, on average, the P^* processors are active more than half of the time; in fact, the average number of operations per time step is $|V|/T(P^*) > |V|/(2|V|/P^*) = (1/2)P^*$.

The above remarks suggest the following definition of degree of parallelism of a CDAG.

Definition 7 *Given a DAG G with $|V|$ nodes and critical length L, the* degree of parallelism *of G is defined as $P^* = |V|/L$.*

Our original goal was to link the degree of parallelism of a DAG G to the quantity P^*_γ, defined as the largest value of P such that the computation can be scheduled on P processors to complete in T steps, with each processor executing at least $\gamma\ T$ operations, where $\gamma \in (0, 1]$ is some chosen constant.

As it turns out, $P^* = |V|/L$ yields a good estimate of $P^*_{1/2}$. In fact, $P^* \leq P^*_{1/2} \leq 2P^*$ (see Problem 2.2.7). A better estimate for $P^*_{1/2}$ is instead rather difficult to obtain, for general DAGs. For these reasons, we will be content with considering P^* as a good characterizations of the degree of parallelism, while remaining aware of the possibility that, for specific DAGs, the analysis can be refined.

[21] Techincally, this statement assumes P^* to be integer. Tedious modifications are needed otherwise, defining $P^* = \lceil |V|/L \rceil$.

3 Prefix Computation

In this section, we consider the *prefix computation* problem, where the input is a sequence of elements from a set endowed with a binary *associative* operation, generically called *product*, and the output is the sequence of the products corresponding to the prefix subsequences of the input, ordered by increasing length. Why are we interested in this problem?

- Prefix computations arise in many different contexts, for many different types of associative operations. A special, very important case, arises in the parallel computation of the evolution of finite state machines (FSMs). In the further specialized case where the FSM computes the addition of two binary numbers, the theory we will develop will yield the design of very fast adders, of the type that are implemented in hardware (within processors, digital filters, FPGAs, etc...).
- While sequential prefix computation is almost a trivial problem[22], parallel prefix computation does exhibit some intriguing aspects, which make it methodologically instructive.

3.1 Problem Formulation and Sequential Algorithm

We begin by introducing the algebraic structure called semigroup.

Definition 8 *A* semigroup *is a pair $S = \langle A, \cdot \rangle$, where A is a set and $\cdot : A \times A \to A$ is a binary* associative *operation, that is,*

$$\forall (x, y, z) \in A^3 \quad (x \cdot y) \cdot z = x \cdot (y \cdot z).$$

Next, we define the prefix computation problem.

Definition 9 Prefix computation*, for a given semigroup $S = \langle A, \cdot \rangle$, is the computational problem where the input is a sequence $\mathbf{x} = (x_0, \ldots, x_{N-1}) \in A^N$ and the output is a sequence $\mathbf{y} = (y_0, \ldots, y_{N-1}) \in A^N$ such that*

$$y_j = \Pi_{i=0}^{j} x_i \qquad j = 0, \ldots, N-1, \tag{21}$$

where Π denotes the iterated $\cdot$ operation. We call associative product *the problem with the same input $\mathbf{x}$, but where only y_{N-1} is required as output.*

[22] In fact, prefix computation is seldom mentioned in typical textbooks on sequential algorithms.

Example: prefix sums. As an example, let $A = \mathbb{Z}$ be the set of the integer numbers and $\cdot$ the addition operations $+$. Then, the prefixes of sequence $\mathbf{x} = (x_0, \ldots, x_{N-1}) \in \mathbb{Z}^N$ are the partial sums $y_j = \sum_{i=0}^{j} x_i$. In particular, y_{N-1} is the total sum of the sequence. (If the sequence samples a function $x(t)$, the total sum approximates the definite integral of x while the partial sums approximate the indefinite integral of x.)

Considering the associativity of "$\cdot$", Equation 21 can be rewritten as

$$y_j = x_0 \cdot x_1 \ldots \cdot x_j, \tag{22}$$

where any parenthesization of the product is legitimate, since they all give the same result. It is trivial to show that the prefixes can be defined inductively as follows:

$$\begin{aligned} y_0 &= x_0; && (23) \\ y_j &= y_{j-1} \cdot x_j, \qquad j = 1, \ldots, N-1. && (24) \end{aligned}$$

The inductive definition immediately yields a very simple sequential algorithm for prefix computation, implemented by the following code fragment:

```
$y_0 = x_0$;
 for $j = 1$ to $(N-1)$ do
    $y_j = y_{j-1} \cdot x_j$;
```

Figure 7 shows the CDAG corresponding to this simple algorithm. All the operations of this algorithm lie on the same data dependence chain of length $L = N - 1$, hence there is no opportunity for concurrent execution of operations. Correspondingly, $P^* = 1$. We also observe that this CDAG admits just one schedule.

3.2 Parallel Prefix Computation

At first, developing a parallel algorithm for prefix computation may appear as an hopeless target. After all, each of the nodes in the CDAG of Figure 7 does produce a required output of the problem and the structure of the CDAG is rooted in the very inductive definition of the problem. So, where can we start to explore parallelization?

Well, we may remember that, in Section 2.2 of Chapter 1, we have outlined a simple divide and conquer method to add N numbers in $O(\log_2 N)$ parallel time. The idea was to divide the input sequence into two halves, concurrently add each half, and then add the corresponding sums. For a general N, this

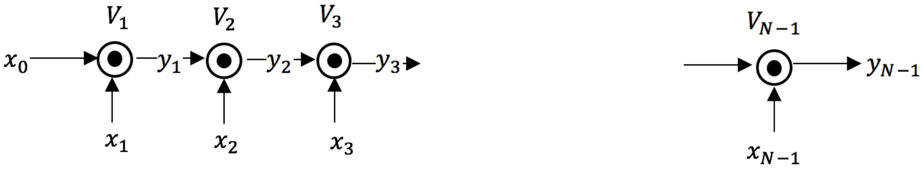


Figure 7: *CDAG of sequential prefix computation.* Input nodes $x_0, x_1, \ldots, x_{N-1}$ form set $I = V_0$. Operation $(\cdot)$ nodes form set V. The output set is $O = V \cup \{x_0\}$. Sets $V_1, V_2, \ldots, V_{N-1}$ are the levels of the greedy schedule, each containing just one node. Since $L = N-1$ and $|V| = N-1$, the degree of parallelism is $P^* = |V|/L = 1$. In fact, at most one operation at the time can be executed. Hence, $T(P) = 1$, for any $P \geq 1$.

corresponds to a recursive implementation of the following formula[23]:

$$y_{N-1} = (x_0 + \ldots + x_{\lfloor N/2 \rfloor - 1}) + (x_{\lfloor N/2 \rfloor} + \ldots + x_{N-1}).$$

The correctness of this formula is based just on the associativity of addition, hence the same approach can be used for the general associative product problem:

$$\begin{aligned} y_{N-1} &= (x_0 \cdot \ldots \cdot x_{\lfloor N/2 \rfloor - 1}) \cdot (x_{\lfloor N/2 \rfloor} \cdot \ldots \cdot x_{N-1}) \quad N > 1, & (25) \\ y_0 &= x_0. & (26) \end{aligned}$$

The CDAG corresponding to the resulting algorithm is a binary tree with N leaves (the inputs $x_0, \ldots, x_{N-1}$, from left to right) and $N-1$ internal nodes (each representing a $\cdot$ operation). It is easy to see that $|V| = N-1$, $L = \lceil \log_2 N \rceil$, hence $P^* \approx N/\log_2 N$. An example, for $N = 10$, is illustrated in Figure 8. We can summarize the discussion with the following proposition.

Proposition 5 *The* associative product *problem can be solved by a parallel algorithm, called the* tree algorithm, *with work* $|V| = N-1$ *and critical length* $L = \lceil \log_2 N \rceil$. *The CDAG of the tree algorithm is a binary tree with* N *leaves. The degre of parallelism is* $P^* = (N-1)/\lceil \log_2 N \rceil$.

A time efficient, but work inefficient algorithm. Since, in the prefix computation problem, each prefix y_j is an associative product, we can separately compute each prefix by the tree algorithm. The resulting CDAG has

[23] We have chosen the first subproblem of size $\lfloor N/2 \rfloor$ and the second subproblem size $N - \lfloor N/2 \rfloor = \lceil N/2 \rceil$. If N is even, the two subproblems have exactly the same size; if N is odd, the size of the second problem is larger, by 1.

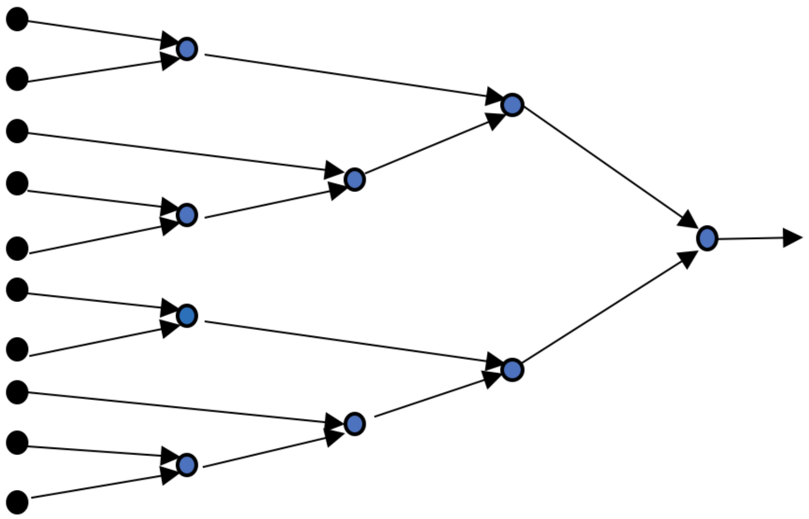


Figure 8: *CDAG of parallel associative product computation.* The CDAG is a binary tree with $N = 10$ leaves (black nodes), corresponding to inputs $x_0, \ldots, x_9$ (from top to bottom) and $N - 1 = 9$ operations (blue nodes). The CDAG is drawn according to its greedy schedule, with the nodes in the same level being vertically aligned. The levels are V_0, V_1, V_2, V_3, V_4, and their respective sizes are $(10, 4, 2, 2, 1)$. The number of levels is $L = 4 = \lceil \log_2 10 \rceil = \lceil \log_2 N \rceil$.

one tree of $j+1$ leaves for each $j = 0, 1, \ldots, N-1$. The various trees share only input nodes, that is, leves. Therefore, the critical length of the CDAG equals that of the largest tree, that is $L = \lceil \log_2 N \rceil$. This is an interesting results since it does translate into

$$T(P) = \lceil \log_2 N \rceil,$$

at least for $P \geq P_{max}$[24]. However, the parallelization has been obtained at the expense of a great increase of work. The number of operations is, in fact,

$$|V| = \sum_{j=1}^{N-1} j = N(N-1)/2.$$

As do we assess the situation? On the positive side, we now know that prefix computation is not inherently sequential, as we may have initially

[24] It is easy to see that $P_{max} \leq |V|/2 = N(N-1)/4$. The exact value of P_{max} as a function of N is somewhat tricky to express analitically and will be explored in the problem section.

feared. We further know that it belongs to NC, since we have an algorithm with polylogarithmic time (specifically, logarithmic) and a polynomial number of processors (specifically, quadratic). On the negative side, a number of operations $N(N-1)/2$ is practically unacceptable for a problem that can be solved with $N-1$ operations, sequentially.

The natural next step is to explore the possibility of reducing work without an excessive increase in parallel time.

3.3 An Optimal Parallel Prefix Algorithm

The sequential prefix computation algorithm is very work efficient because it reuses the work done for prefix y_{j-1} in order to obtain y_j, with little extra effort, as $y_j = y_{j-1} \cdot x_j$. This strength becomes a weakness in the parallel context, because a long dependence chain is formed. On the other hand, the parallel approach of the preceding section recomputes each y_j from scratch, without any reuse of work, leading to a very small critical length, but to a very high amount of work. Can we find an intermediate approach, which achieves low work by means of substantial reuse, but without forming critical paths of significant length?

To make the idea more specific, let us consider the tree that computes y_{N-1}. Each internal node of this tree computes a product $y_{h,k} = x_h \cdot x_{h+1} \cdot \ldots \cdot x_k$ corresponding to some infix of the input sequence. Can we efficiently obtain all the y_j's by suitable compositions of the given infixes? The answer is positive and leads to the Twisted Reflected Tree algorithm for prefix computation[25].

The structure of the TRT CDAG is illustrated in Figure 9, for input size $N = 16$. The first half of the CDAG is a tree that computes y_{N-1}, in general, and y_{15} in the figure example. Here, we are interested in using the values computed by all nodes in the tree in order to obtain all other prefixes as well. In the example these values, from top to bottom in each stage (with reference to the drawing), are:

- Stage 0: $y_{0,0}{=}x_0$, $y_{1,1}{=}x_1$, $y_{2,2}{=}x_2$, $y_{3,3}{=}x_3$, $y_{4,4}{=}x_4$, $y_{5,5}{=}x_5$, $y_{6,6}{=}x_6$, $y_{7,7}{=}x_7$, $y_{8,8}{=}x_8$, $y_{9,9}{=}x_9$, $y_{10,10}{=}x_{10}$, $y_{11,11}{=}x_{11}$, $y_{12,12}{=}x_{12}$, $y_{13,13}{=}x_{13}$, $y_{14,14}{=}x_{14}$, $y_{15,15}{=}x_{15}$.
- Stage 1: $y_{0,1}{=}x_0 \cdot x_1$, $y_{2,3}{=}x_2 \cdot x_3$, $y_{4,5}{=}x_4 \cdot x_5$, $y_{6,7}{=}x_6 \cdot x_7$, $y_{8,9}{=}x_8 \cdot x_9$, $y_{10,11}{=}x_{10} \cdot x_{11}$, $y_{12,13}{=}x_{12} \cdot x_{13}$, $y_{14,15}{=}x_{14} \cdot x_{15}$.

[25] The TRT was introduced by G. Bilardi and F.P. Preparata, Digital filtering in VLSI, in VLSI Algorithms and Architectures, Aegean Workshop on Computing, Loutraki, Greece, July 8-11, 1986, Proceedings, pp. 111, 1986.

- Stage 2: $y_{0,3}{=}x_0 \cdot x_1 \cdot x_2 \cdot x_3$, $y_{4,7}{=}x_4 \cdot x_5 \cdot x_6 \cdot x_7$, $y_{8,11}{=}x_8 \cdot x_9 \cdot x_{10} \cdot x_{11}$, $y_{12,15}{=}x_{12} \cdot x_{13} \cdot x_{14} \cdot x_{15}$.
- Stage 3: $y_{0,7}{=}x_0 \cdot x_1 \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 \cdot x_6 \cdot x_7$, $y_{8,15}{=}x_8 \cdot x_9 \cdot x_{10} \cdot x_{11} \cdot x_{12} \cdot x_{13} \cdot x_{14} \cdot x_{15}$.
- Stage 4: $y_{0,15}{=}x_0 \cdot x_1 \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 \cdot x_6 \cdot x_7 \cdot x_8 \cdot x_9 \cdot x_{10} \cdot x_{11} \cdot x_{12} \cdot x_{13} \cdot x_{14} \cdot x_{15}$.

The left half of the CDAG forms further infixes as follows:

- Stage 5: no new products are computed;
- Stage 6: $y_{0,11}{=}y_{0,7} \cdot y_{8,11}$.
- Stage 7: $y_{0,5}{=}y_{0,3} \cdot y_{4,5}$; $y_{0,9}{=}y_{0,7} \cdot y_{8,9}$; $y_{0,13}{=}y_{0,11} \cdot y_{12,13}$.
- Stage 8: $y_0{=}y_{0,0}$, $y_1{=}y_{0,1}$, $y_2{=}y_{0,1} \cdot y_{2,2}$, $y_3{=}y_{0,3}$, $y_4{=}y_{0,3} \cdot y_{4,4}$, $y_5{=}y_{0,3} \cdot y_{4,5}$, $y_6{=}y_{0,5} \cdot y_{6,6}$, $y_7{=}y_{0,7}$, $y_8{=}y_{0,7} \cdot y_{8,8}$, $y_9{=}y_{0,7} \cdot y_{8,9}$, $y_{10}{=}y_{0,9} \cdot y_{10,10}$, $y_{11}{=}y_{0,7} \cdot y_{8,11}$, $y_{12}{=}y_{0,11} \cdot y_{12,12}$, $y_{13}{=}y_{0,11} \cdot y_{12,13}$, $y_{14}{=}y_{0,13} \cdot y_{14,14}$, $y_{15}{=}y_{0,15}$.

Figure 10 shows how, with the addition of a few nodes and arcs, the second half of the CDAG also becomes a tree, acting from the root to the leaves. This tree is reflected with respect to the first one, and also twisted by one line upward, whence the name of the CDAG and of the algorithm.

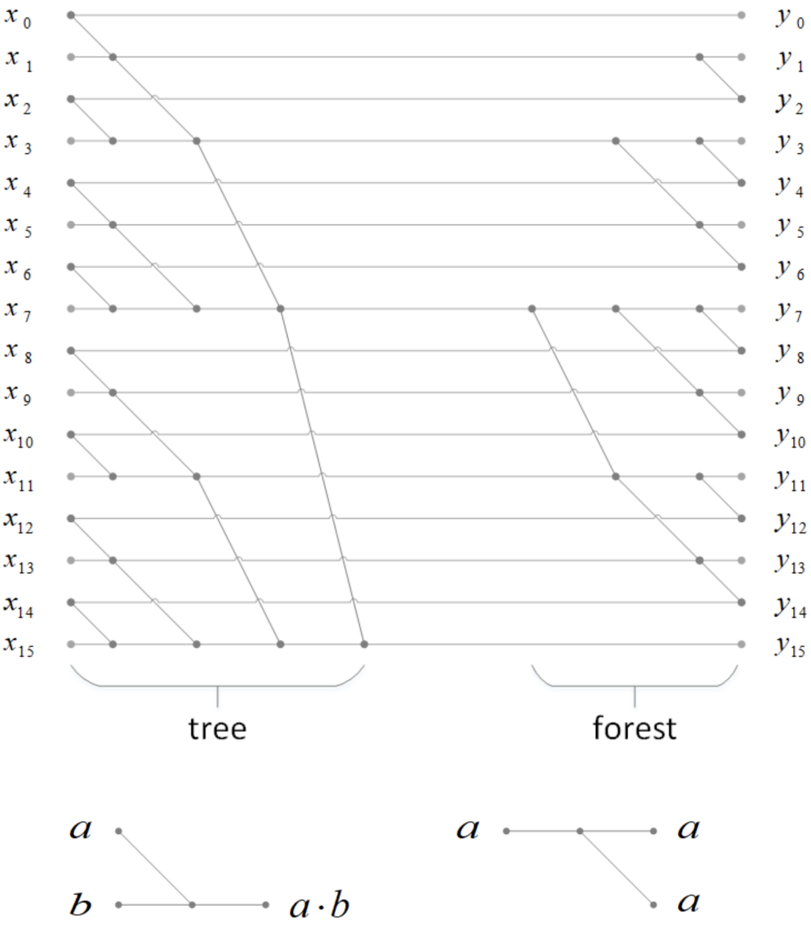


Figure 9: *The Twisted Reflected Tree (TRT) CDAG for Prefix Computation.* The figure shows a $TRT(N)$ CDAG with $N = 16$ inputs. All arcs are meant to be directed from left to right. Nodes with no incoming arcs are input nodes. Nodes with one input and two outputs replicate the input on all output arcs. Nodes with two inputs are operation nodes performing the semigroup product $a \cdot b$, where a is the upper incoming arc and b is the lower incoming arc. The first (left) half of the DAG is a binary tree with N leaves, drawn in an unusual, but convenitent way. Each node of this tree computes the product of some infix of the input sequence. The second (right) half of the DAG is a forest, which combines the infixes to obtain the prefixes, which are output at the leaves.

Proposition 6 *The* prefix computation *problem can be solved by the* Twisted reflected Tree (TRT) *algorithm with work* $|V| = 2(N-1) - \log_2 N$ *and critical length* $L = 2(\log_2 N - 1)$*, assuming* N *to be a power of 2. The degree of parallelism is* $P^* \approx (N-1)/(\log_2 N - 1)$.

Proof: *Correctness.* ...

Work. Consider the augmented TRT, as shown in Figure 10 and observe that, at each level of the second tree, broadcast nodes alternate with operation nodes, starting at the top with a broadcast node. The number of operation nodes is then $N/2 + N/4 + \ldots + 2 + 1 = N - 1$. In the first tree, the operation nodes are exactly the internal nodes, which are $N-1$. In total, the number of operations in the augmented TRT is $N-1$.

In the (simple) TRT, the removal of the dashed arcs transforms a node in each level, except at the root level, from operation to broadcast. Thus, there are $\log_2 N$ fewer nodes than in the augmented TRT, for a total of $2(N-1) - \log_2 N$ operation nodes.

Critical length. Consider the path starting at the top leaf of the first tree, going to the root of the top subtree ($\log_2 N - 1$ arcs), then to the root of the bottom subtree of the second subtree (1 arc), then to the second to bottom leaf of this subtree ($\log_2 N - 1$ arcs). The length of this path is then $2\log_2 N - 1$. It is easy, if a bit tedious, to argue that there is no longer path. *QED*

With linear work and logarithmic critical length, the TRT is highly parallel with very negligible increase of work with respect to its sequential counter part. We can in fact show that, under reasonable assumptions, the TRT algorithm is optimal, both in work and in critical length. The lower bound argument is of rather general applicability, since is based solely on the property that the output actually depends on all input variables.

Proposition 7 (Lower bounds on functions of N variables.) *We say that a function* $F(x_0, \ldots, x_{N-1})$ *effectively depends upon variable* x_j *if there are two inputs* $\mathbf{x}'$ *and* $\mathbf{x}''$*, which differ only for the value of* x_j*, such that* $F(\mathbf{x}') \neq F(\mathbf{x}'')$.

Let F *be a function returning one value, which effectively depends upon* N *variables. Then, any CDAG computing* F*, using unary or binary operations, satisfies the following lower bounds:*

1. $|V| \geq N - 1$, *(work lower bound);*

2. $L \geq \log_2 N$, *(critical length lower bound).*

Proof: Let $G = (I, V, E)$ be the DAG component of a CDAG computing F and let r be the node of G that outputs the value of F. Without loss of generality, assume that r is reachable by each node of G[26].

In particular, r must be reachable by every node in I, whose element correspond to the N variables of F. This claim can be established by contradiction. Let r be non reachable from some x_j. Then, no matter what the value of the other input variables is, solely changing the value of input x_j cannot affect the value of r, hence the output of F, which contradicts the assumption that F does effectively depend upon x_j.

Let now $\mathcal{T} = (I, V, B)$ be the tree obtained by a reverse breadth first search (BFS) of G, starting at r. That is, in the search, the arcs of E are traversed from target to source, starting at r. The tree arcs (set B) are assigned the same direction as they have in E, so they go from child to parent. By the reachability assumption, all nodes of $I + V$ belong to $\mathcal{T}$.

Work lower bound. Let $W \subseteq I + V$ be the set of leaves of $\mathcal{T}$. Clearly, $I \subseteq W$, since a node in I has no incoming arc, so it cannot be the parent of another node [27]. Recalling now that, in any tree where each node has at most two children, the number of internal nodes is no smaller than the number of leaves, we have, for tree $\mathcal{T}$:

$$|V| - |V \cap W| \geq |W| - 1.$$

Since $|V| \geq |V| - |V \cap W|$ and $|W| \geq |I|$, we concude that $|V| \geq |I| - 1 = N - 1$, as stated.

Critical length lower bound. Let D be the depth of $\mathcal{T}$. Recalling now that, in any tree where each node has at most two children, the number of leaves is at most two to the depth, we have $|W| \leq 2^D$, whence $D \geq \log_2 |W| \geq \log_2 |I|$. Since any path in $\mathcal{T}$ is also a path in G, we have $L \geq D$, from which we conclude that $L \geq \log_2 |I| = \log_2 N$, as stated. *QED*

An application of the previous proposition to prefix computation yields the next result.

Proposition 8 *The TRT algorithm is optimal, for prefix computation, in terms of both work and critical length, under the assumptions that the primitive operation is the binary semigroup product and that the algorithm works for any semigroup.*

[26] Otherwise, the argument can be applied to the DAG obtained from G, by removing all nodes from which r is not reachable.

[27] For clarity, we observe that it is possible for a node $v \in V$ to belong to W, if the BFS reaches the predecessors of v by successors different from v, so that the arcs entering v are not tree arcs. Thus we cannot claim that $I = W$.

Proof: The argument applies Proposition 7 to the computation of

$$y_{N-1} = F(x_0, \ldots, x_{N-1}) = x_0 \cdot \ldots \cdot x_{N-1}.$$

We just need to show that, at least for one semigroup, the product of N factors effectively depends on each factor. The proof is particularly simple for the (semi)group $\langle \{0, 1\}, \oplus \rangle$, where "$\oplus$" denotes the exclusive OR, or sum modulo 2, operation, which equals 1 if and only if the two operands are different. For this (semi)group we have:

$$y_{N-1} = x_0 \oplus \ldots \oplus x_{N-1},$$

and it is easy to see that flippling just one variable changes the product, for any assignment of values to the remaining variables.

Comparing Propositions 8 and 7, we conclude that the TRT algorithm is work-optimal, to within a factor of 2, and critical-length optimal, also to within a factor of 2. *QED*

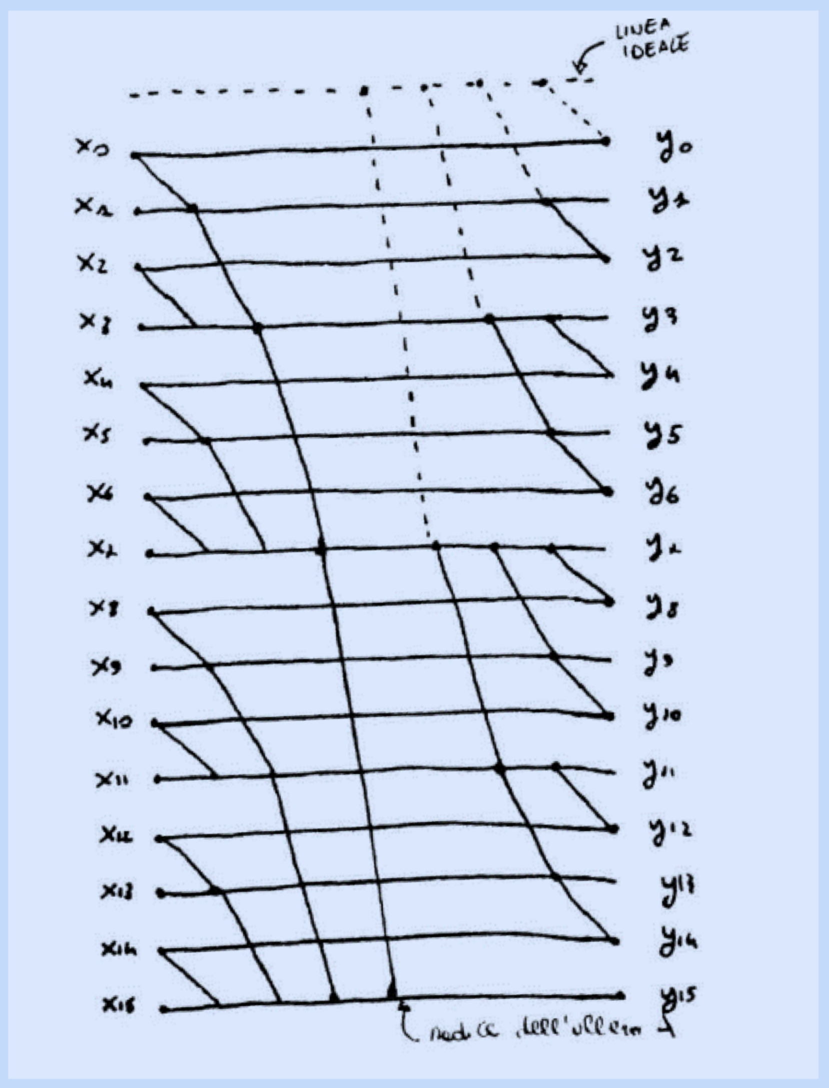


Figure 10: *The Twisted Reflected Tree (TRT) CDAG for Prefix Computation - Augmented version.* The figure shows a $TRT(N)$ CDAG with $N = 16$ inputs with the addition of some nodes connecetd by arcs shown as dashed lines. With this addition, the right half of the DAG becomes a second binary tree, which is reflected and twisted (shifted up by one line) with respect to the first tree. Beside highlighting the origin og the CDAG name, this addition enables the left multiplication of all the outputs by a semigroup value to be input on the dashed horizontal line. This feature is useful in some applications, including the use of the TRT to process an input signal of arbitrary length, applied to the inputs N at the time.

4 Evolution of Finite State Machines

The concept of *dynamical system* is central to science and engineering. Dynamical systems have an internal configuration, described by a state, which evolves in time under the influence of an input from the environment; as a function of state and input, the system produces an output, which is observable and can impact the enviroment. Physical systems are generally modelled as dynamical systems, with the relationships between input, state, and output described by (ordinary or partial) differential equations. In information engineering - control systems, communication systems, and signal processing systems are also of a dynamical nature, often described by ordinary differential equations, or by difference equations, when time is discretized by means of a clock. (Discretization also arises as a technique to numerically solve differential equations.)

Computing systems, such as a Turing machine or a modern processor, are also dynamical systems. Although computing systems can function in a “streaming” mode, by processing an input signal whose samples arrive at different times, more often they work by “state transformation,” with the input of the computational problem to be solved being encoded in the initial state and the output of the computation being encoded in the final state. While, mathematically, it is often useful to assume that inputs, states, and outputs are elements from infinite sets, in any engineering system these sets are actually finite. Such systems are known as finite state machines, where time is typically assumed to be discrete. These are the systems that we will study in this section.

4.1 Problem Formulation

We begin with a formal definition of finite state machines.

Definition 10 (Finite State Machine (FSM).) *A* finite state machine *is a quintuple $M = (\Sigma, \Gamma, Q, \delta, \eta)$ where:*

- Σ *is a finite set, called the* input set *or the* input alphabet*;*
- Γ *is a finite set, called the* output set *or the* output alphabet*;*
- Q *is a finite set, called the* state set*;*
- $\delta: \ Q \times \Sigma \to Q$ *is the* state transition function*;*
- $\eta: \ Q \times \Sigma \to \Gamma$ *is the* output transformation function*.*

An FSM evolves in discrete time. If $u(t) \in \Sigma$, $q(t) \in Q$, and $z(t) \in \Gamma$, respectively denote the value of the input, the state, and the output, at time t, the evolution is governed by the following equations[28]*:*

$$q(t+1) = \delta(q(t), u(t)); \tag{27}$$

$$z(t) = \eta(q(t), u(t)). \tag{28}$$

The main objective of this section is the development of efficient parallel algorithms to solve the following problem.

Definition 11 (Evolution of an FSM.) *Let M be an FSM. The computational problem* evolution of an FSM *is defined as follows:*
Input: *A state $q_0 \in Q$ and a sequence of input values $(u_0, \ldots, u_{N-1}) \in \Sigma^N$.*
Output: *A sequence of states $(q_1, \ldots, q_N) \in Q^N$ and a sequence of outputs such that $(z_0, \ldots, z_{N-1}) \in \Gamma^N$ such that:*

$$\begin{aligned} q_{t+1} &= \delta(q_t, u_t); \quad \text{for} \ \ t = 0, \ldots, N-1; & (29) \\ z_t &= \eta(q_t, u_t); \quad \text{for} \ \ t = 0, \ldots, N-1. & (30) \end{aligned}$$

We observe that, algorithmically, the crucial part is the computation of the states $q_1, \ldots, q_N$. Once these states are available, it is straightforward to compute, in parallel, $\eta(q_t, u_t)$, for each t.

From the perspective of sequential algorithms, computing the evolution of an FSM is conceptually a simple matter, proceeding step after step. For a fixed FSM M, each evaluation of the functions δ and η takes time independent of N. Therefore, the problem can be solved in time $O(c_M N)$. The term c_M is constant with respect to N, the number of steps for which we want to simulate the machine. However, c_M does depend upon the FSM M; different state transition and output transformation functions will exhibit different computational complexity.

At first, parallelization may appear hopeless. Even though all the values of the system input are immediately available (as problem inputs), the state transition at time t cannot be computed without knowing the state $q(t)$. Although this obstacle may appear insurmountable, we will find a way to

[28] The type of system defined here is often called *Mealy FSM* (1955). Often used is the special case where the output directly depends only upon the state ($\eta : Q \to \Gamma$, $z(t) = \eta(q(t))$), which is called a *Moore FSM* (1956). In the theory of computation, the notion of *Deterministic Finite Automaton (DFA)* is introduced. A DFA can be viewed as a special case of Moore FSM, with $\Gamma = \{accept, not{-}accept\}$; customarily, the function η is encoded by the set $F = \{q \in Q : \eta(q) = accept\}$, known as the set of final states. An initial state $\bar{q} \in Q$ is also specified, when using the DFA to define a *regular language* $L \subseteq \Sigma^*$.

circumvent it, by reducing the problem of the evolution for a given FSM M to the problem of prefix computation for a semigroup S_M, constructed from M in a specific way. Once this reduction is accomplished, we can leverage the efficient parallel prefix algorithms presented in the previous section.

4.2 FSM Evolution and Prefix Computation

The basic idea enabling the connection between FSM evolution and semigroup prefixes is as simple as it is brilliant. It is based on introducing, for each input symbol $a \in \Sigma$, a function $\delta_a : Q \to Q$, mapping states into states, defined as

$$\delta_a(q) = \delta(q, a).^{29} \tag{31}$$

The reader may be justifiably surprised at hearing that Equation 31 encodes a brilliant idea. After all, it amounts to a change of notation, so how can it be so significant? Well, a change of notation often carries a change in perspective, which may have profound consequences. Let us see the idea in action, in the context of FSMs. Suppose we want to connect $q(2)$ to $q(0)$. By Equation 27 we obtain:

$$q(2) = \delta(q(1), u(1)) = \delta(\delta(q(0), u(0)), u(1)).$$

This equation is rather opaque. Imagine extending it to $q(3), q(4), \ldots$: it soon becomes quite ackward. Now, let us rewrite the equation using the notation of Equation 31. We find that

$$q(2) = \delta_{u(1)}(q(1)) = \delta_{u(1)}(\delta_{u(0)}(q(0)) = (\delta_{u(0)} * \delta_{u(1)})(q(0)),$$

where "$*$" denotes the operation of composition of two functions. The definition of composition of two functions is reviewed in Problem 3.1.4, at the end of this chapter, where the reader is also asked to show that composition is an associative operation, giving rise to the semigroup $\langle Q^Q, * \rangle$, where Q^Q denotes the set of functions from Q to Q.

In general, we can write:

$$q(t) = (\delta_{u(0)} * \delta_{u(1)} * \ldots * \delta_{u(t-1)})(q(0)), \qquad \text{for } t = 1, 2, \ldots, N. \tag{32}$$

[29] The process of going from a function of two variables to a function where one of the two variables remains as an argument, whereas the other variable becomes a label, is known as *currying*. The name has been introduced in honor of the logician Haskell Curry (1900-1982), after whom the programming language Haskell is also named. Currying plays a role in many areas of mathematics and computer science, such as logic, set theory, category theory, functional analysis, lambda calculus, and programming languages.

In conclusion, if we let $\delta = (\delta_{u(0)}, \delta_{u(1)}, \ldots, \delta_{u(N-1)})$ and

$$\xi = (\xi_0, \xi_1, \ldots, \xi_{N-1}) = \text{prefixes}(\delta), \tag{33}$$

then

$$q(t) = \xi_{t-1}(q(0)). \tag{34}$$

4.3 Parallel algorithm for FSM evolution

The preceding discussion suggests an algorithmic approach to FSM evolution, whose steps we outline next.

1. Input $q_0 \in Q$ and $(u_0, u_1, \ldots, u_{N-1}) \in \Sigma^N$.

2. For each $t = 0, 1, \ldots, N-1$ obtain, in parallel, δ_{u_t}.

3. Using an efficient parallel algorithm, such as the TRT, compute the following prefixes, with respect to function composition:

$$\xi_t = \delta_{u_0} * \delta_{u_1} * \ldots * \delta_{u_t}, \qquad t = 0, \ldots, N-1. \tag{35}$$

4. For each $t = 1, \ldots, N$, in parallel, obtain q_t by evaluating the prefix ξ_t at the initial state q_0:

$$q_t = \xi_{t-1}(q_0), \qquad t = 1, \ldots, N. \tag{36}$$

5. For each $t = 0, 1, \ldots, N-1$, in parallel, obtain the output at time t as

$$z_t = \eta(q_t, u_t), \qquad t = 0, \ldots, N-1. \tag{37}$$

6. Output $(q_1, q_2, \ldots, q_N) \in Q^N$ and $(z_0, z_1, \ldots, z_{N-1}) \in \Gamma^N$.

Before the above outline can be considered a fully fledged algorithm, the representation of the various types of data has to be specified. It is natural to assume the representation of elements of Σ, Γ, and Q to be part of the specification of machine M, since it will generally be determined by the context in which the machine is meant to operate[30].

The situation is different for the representation of state functions. In general, they are members of Q^Q. However, when considering a machine M, only functions in a specific subset of Q^Q can occur, during the execution the

[30] Strictly speaking, this observation does not necessarily apply to the representation of the states, the choice of which is in fact often a degree of freedom in the design of a machine satisfying given input/output specifications. However, here we consider the case when the representation of the states has already been chosen and the study of the machine evolution has to comply with it.

above algorithm. More precisely, only finite compositions of functions in the set $\{\delta_a : a \in \Sigma\}$ can arise, as a result of the computation in Equation 35. As it is easily shown, these compositions form a semigroup, which we will denote $S_M = \langle A_M, *\rangle$, to highlight its connection to FSM M. For some FSMs, $|A_M|$ can be much smaller than $|Q|^{|Q|}$, which may lead to a more compact representation of the relevant functions. To select one representation, among the many that are possible, we may want to consider the impact of the representation on the computations of our algorithm involving state functions. These computations include: obtaining the representation of δ_a from the representation of a (Step 2); performing semigroup multiplications (Step 3); and evaluating a function when the argument is the initial state q_0 (Step 4). While there is no general recipe to find the best representation for any machine, the preceding points can serve as guidelines to find a good one.

The next theorem states the main result of this section.

Theorem 3 *The algorithm presented in this section computes the evolution of an FSM M, over an interval of N steps, with critical length $L = O(\lambda_M \log_2 N)$ and work $O(c_M N)$. The terms λ_M and c_M are independent of N, but do depend upon M, as well as on the specific representation adopted for state functions.*

Proof: The correctness of the algorithm has been established by the developments of this section. The critical length and work bound are essentially those of the TRT algorithm, invoked in Step 3 of the algorithm. Steps 2,4, and 5 all contribute work proportional to N and critical length independent of N. *QED*

The material presented in this section is not inherentlty complicated, but may result a bit difficult to grasp on a first reading, for the somewhat abstract nature of the concepts. The next section applies the results of the present one to an FSM, M_{add}, which adds binary integers, leading to the design of a fast parallel adder. Specializing the general framework of the present section to M_{add} will also provide a concrete illustration of the abstract concepts introduced here.

5 A Parallel Binary Adder

Integer addition is a basic operation. Nearly all processors are equipped with functional units to add numbers, direclty in hardware. To be fast, these units implement parallel algorithms. In this section, we focus on the addition of integers with the standard binary representation, although the ideas we will discuss do apply to other bases and representations. For simplicity, we will restrict our attention to the non negative integers (*i.e.*, the natural numbers).

Definition 12 *An input of the* binary addition *computational problem is a pair of N-bit strings, $a_{N-1} \ldots a_1 a_0 \in \{0,1\}^N$ and $b_{N-1} \ldots b_1 b_0 \in \{0,1\}^N$. The corresponding output is the unique $(N+1)$-bit string $s_N s_{N-1} \ldots s_1 s_0 \in \{0,1\}^{N+1}$ such that*

$$\sum_{j=0}^{N} s_j 2^j = \sum_{j=0}^{N-1} a_j 2^j + \sum_{j=0}^{N-1} b_j 2^j. \tag{38}$$

5.1 An FSM for Binary Addition

When we add two numbers by hand, we typically follow the "long addition" algorithm, learnt in primary school[31]. The bits, one per operand, are scanned from the rightmost (least significant) position to the leftmost (most significant) position. At position j, the corresponding bit s_j of the sum is produced as well as a "carry" bit, c_{j+1} which will be used in obtaining the sum bit for the next position. For the first position, the received carry is set to zero ($c_0 = 0$). When executing long addition, we behave like an FSM, as sketched in Figure 11: at each step, the machine inputs two bits (one per operand), updates a state representing the carry, and outputs a bit of the sum. This FSM is formally defined next.

Definition 13 *The* FSM for binary addition *is $M_{add} = (\Sigma, Q, \Gamma, \delta, \eta)$, where:*

- $\Sigma = \{0,1\} \times \{0,1\}$;
- $Q = \{0,1\}$;
- $\Gamma = \{0,1\}$;
- $\delta(q,(a,b)) = \text{threshold}_2(a,b,q)$;[32]

[31] The algorithm is for base-10 numbers, but is easily adapted to base-2 numbers.

[32] By definition, the Boolean function $\text{threshold}_k(x_1,\ldots,x_N)$ of the N Boolean variables $x_1,\ldots,x_N$, equals 1 if at least k of the arguments are 1, and equals 0, otherwise. Threshold functions are monotone and symmetric; they play a role in several domains.

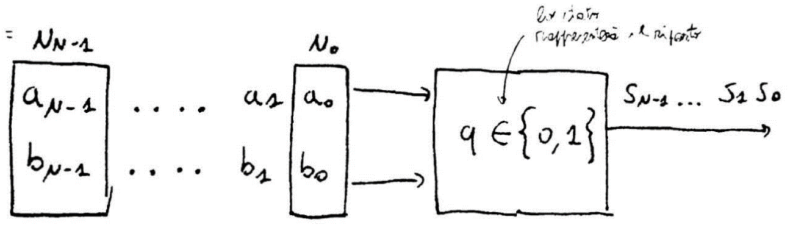


Figure 11: M_{add}*: input, state, output.* At a given time step, M_{add} reads one bit for each operand and writes the corresponding bit of the sum. The state keeps the carry from the previous position. Positions are processed in order of increasing significance.

- $\eta(q, (a, b)) = a \oplus b \oplus q$.

It is a simple exercize to verify that $\text{threshold}_2(a, b, q)$ equals the carry for the next position (there is a carry if and only if $a + b + q \geq 2$), while $a \oplus b \oplus q$ equals the sum for the current position (the sum bit equals the parity of (a, b, q)). Thus, we can make the following statement.

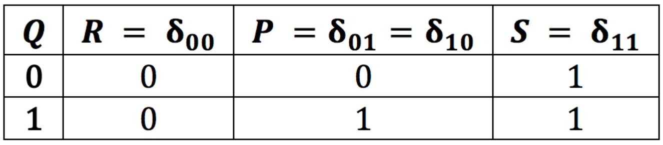
Proposition 9 *If the input of M_{add} is set to $u(t) = (a_t, b_t)$, for $t = 0, 1, \dots, N-1$, and the initial state is set as $q(0) = 0$, then $z(t) = s_t$ and $q(t) = c_t$, for $t = 0, 1, \dots, N-1$. Finally, $q(N) = s_N$.*

5.2 The Carry Semigroup

We now study the semigroup $S_{M_{add}}$ associated with FSM M_{add}, which is known in the literature as the *carry semigroup*, denoted $S_{carry} = \langle A_{carry}, * \rangle$. We begin by determining, for each element u of the input alphabet $\Sigma = \{00, 01, 10, 11\}$, the corresponding state function function $\delta_u : \{0,1\} \to \{0,1\}$, according to Equation 31. We have:

$$\begin{aligned}
\delta_{00}(q) &= \delta(q, (0,0)) = \text{threshold}_2(0, 0, q) = 0; \\
\delta_{01}(q) &= \delta(q, (0,1)) = \text{threshold}_2(0, 1, q) = q; \\
\delta_{10}(q) &= \delta(q, (1,0)) = \text{threshold}_2(1, 0, q) = q; \\
\delta_{11}(q) &= \delta(q, (1,1)) = \text{threshold}_2(1, 1, q) = 1.
\end{aligned}$$

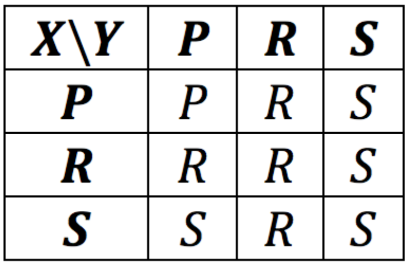
We observe that δ_{00} is the constant function equal to 0, δ_{11} is the constant function equal to 1, and $\delta_{01} = \delta_{10}$ is the identity function. In the context of binary addition, it is customary to call these three functions *carry reset*, *carry set*, and *carry propagate*, and to denote them as R, S, and P. They



Q	$R = \delta_{00}$	$P = \delta_{01} = \delta_{10}$	$S = \delta_{11}$
0	0	0	1
1	0	1	1

Figure 12: *Generators of the carry semigroup S_{carry}.*

are the generators of the machine semigroup (see Figure 12). The semigroup includes all the finite compositions of the generators. We begin by considering the composition $X * Y$ of any two generators. By definition, $(X * Y)(q) = Y(X(q))$. A straightforward calculation will establish the composition table shown in Figure 13. The calculation can be conveniently broken down into three cases, based on the value of the second factor Y:



$X \backslash Y$	P	R	S
P	P	R	S
R	R	R	S
S	S	R	S

Figure 13: *Composition table of the carry semigroup S_{carry}.* The composition product $X * Y$ is shown in row X of column Y. The carry reset R and the carry set S are right zeros; the carry propagate P is the identity.

$$X * R = R; \tag{39}$$
$$X * S = S; \tag{40}$$
$$X * P = X. \tag{41}$$

Clearly, $(X * R)(q) = R(X(q)) = 0$, $(X * S)(q) = S(X(q)) = 1$, and $(X * P)(q) = P(X(q)) = X(q)$. Since multiplying two generators reproduces a generator, the set $\{R, S, P\}$ exhausts the semigroup[33]. Since our machine has two states, $Q = \{0, 1\}$, the set $Q^Q = \{0, 1\}^{\{0,1\}}$ of state functions contains four elements. In addition to R, S, and P, there is the *complement function*,

[33] In general, the product of two generators is not necessarily a generator, and the composition table has to be extended, until a set closed under composition is reached. Our case is particular, in that the set of generators is already closed under composition.

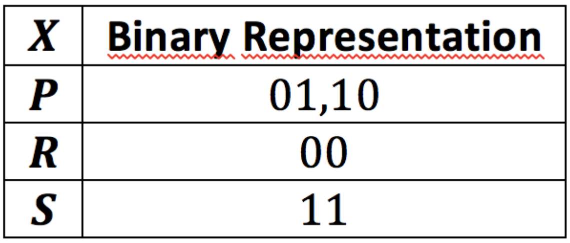
C, defined as $C(q) = 1 - q$. In summary, $Q^Q = \{R, S, P, C\}$, while $A_{carry} = \{R, S, P\}$, a proper subset of Q^Q. This illustrates the general principle that the semigroup associated with a machine needs not include all possible state functions. Here, we only "save" one function, but for other machines the savings may be very significant.

In conclusion, the carry semigroup, associated with FSM M_{add}, is

$$S_{carry} = \langle A_{carry}, * \rangle = \langle \{P, R, S\}, * \rangle.$$

5.3 Representation for the Carry Semigroup

We now introduce a binary representation for the elements of the carry semigroup, which will give us a concrete basis to compute in the semigroup. Our choice is shown in Figure 14. By representing δ_u by the string u, Step 2 of the FSM evolution algorithm (Section 4) becomes trivial, actually requiring no computation. Since $\delta_{01} = \delta_{10} = P$, element P has two representations[34].



X	Binary Representation
P	01,10
R	00
S	11

Figure 14: *A binary representation of the elements of S_{carry}.* Different elements must have different representations, but the same element can admit several representations, as in the case of P. The chosen representation has the advantage that the representation of δ_u is u itself, which makes Step 2 of the algorithm presented in Section 4 trivial.

5.4 Computation in the Carry Semigroup

Having chosen a representation for the semigroup elements, we now consider the problem of computing the product $Z = X * Y$, that is, of obtaining a representation of Z from given representations of X and Y (see Figure 15). Since an important application of the parallel addition algorithm is in the

[34] Having multiple representations for the same abstract object is not uncommon, and in fact is often crucial to the design of efficient algorithms. For example, when dictionaries (sets with insertion, deletion, and search) are implemented via balanced trees, many different trees represent the same set.

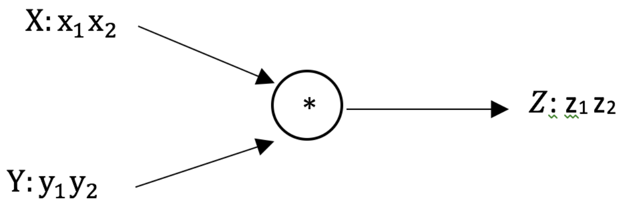


Figure 15: *Semigroup multiplier.* The semigroup multiplier is a key bulding block of the TRT algorithm for prefix computation. Once a binary representation is chosen, the bits representing the product, z_1 and z_2, are Boolean functions of the bits representing the two factors, x_1,x_2, y_1, and y_2.

context of hardware design, we are interested in computing the semigroup product by a Boolean circuit. Figure 16 shows the thruth table of z_1 and z_2 (the bits of the product) as a function of x_1, x_2, y_1, and y_2 (the bits of the operands). The functions are obtained by combining the abstract multiplication table (Figure 13) with the adopted representation (Figure 14).

The d_i's denote "don't care" conditions; a pair $(d, \bar{d})$ arises when $Z : z_1 z_2 = P$, since both $(z_1, z_2) = (0, 1)$ $(d = 0)$ and $(z_1, z_2) = (1, 0)$ $(d = 1)$ are valid representations. In the design of a Boolean circuit implementing the function, the "don't care" values $(d_1, d_2, d_3$ and $d_4)$, can be set to either 0 or 1, for example, to reduce the number of circuit gates.

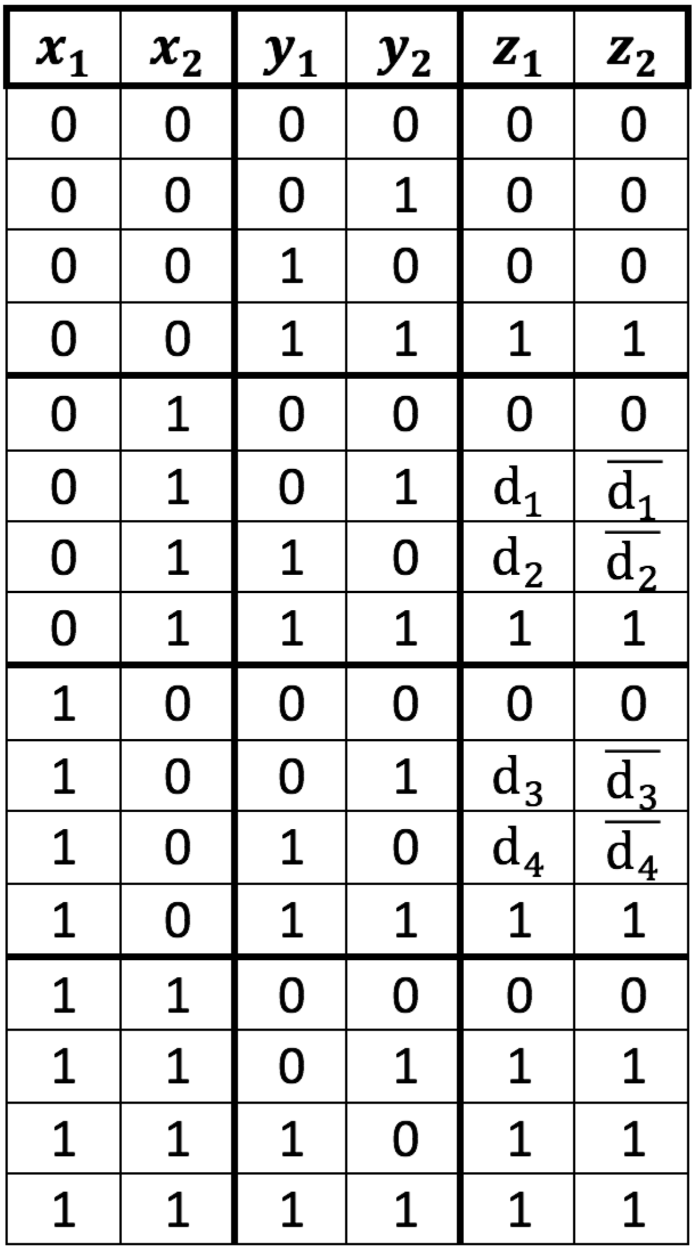
We can check that, if $d_1 = d_2 = 1$ and $d_3 = d_4 = 0$, the Boolean functions in Figure 16 can be written as:

$$z_1 = y_1 y_2 + x_2 y_2 + x_2 y_1, \quad (42)$$

$$z_2 = y_1 y_2 + x_1 y_2 + x_1 y_1, \quad (43)$$

where xy $(x + y)$ denotes the Boolean AND (OR) of x and y. We can also directly check the correctness of Equations 16, by the following case analysis, based on the three equations starting at 39.

- $Y = R$. Here, $y_1 = y_2 = 0$, whence $z_1 = z_2 = 0$, so that $Z = R$.
- $Y = S$. Here, $y_1 = y_2 = 1$, whence $z_1 = z_2 = 1$, so that $Z = S$.
- $Y = P$. We have two subcases:
  - $y_1 = 0$ and $y_2 = 1$, whence $z_1 = x_2$ and $z_2 = x_1$;
  - $y_1 = 1$ and $y_2 = 0$, whence $z_1 = x_1$ and $z_2 = x_2$.



x_1	x_2	y_1	y_2	z_1	z_2
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	d_1	$\overline{d_1}$
0	1	1	0	d_2	$\overline{d_2}$
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	d_3	$\overline{d_3}$
1	0	1	0	d_4	$\overline{d_4}$
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

Figure 16: *Boolean functions for the semigroup multiplier.* The functions are obtained by combining the abstract multiplication table (Figure 13) with the adopted representation (Figure 14). The d_i's denote “don't care” conditions; a pair $(d, \bar{d})$ arises when $Z : z_1 z_2 = P$, since both $(z_1, z_2) = (0, 1)$ $(d = 0)$ and $(z_1, z_2) = (1, 0)$ $(d = 1)$ are valid representations.

In the second subcase, it is obvious that $Z = X$, as it should be. In the first subcase, the same conclusion holds, because x_1x_2 and x_2x_1 do represent the same semigroup element.

Finally, as shown in Figure 17, we can draw a Boolean circuit, with AND and OR gates, that computes the carry-semigroup product, with the chosen representation. The Boolean circuit can be viewed as a CDAG, with 7 operations and critical length equal to 2. The parallel algorithm of Section 4.3, in addition to requiring semigroup multiplications in Step 3 as part of the TRT computation, also relies on the evaluation of a semigroup element on a state q, in Step 4. We claim that, for the the carry semigroup, given an element X represented as x_1x_2, we have:

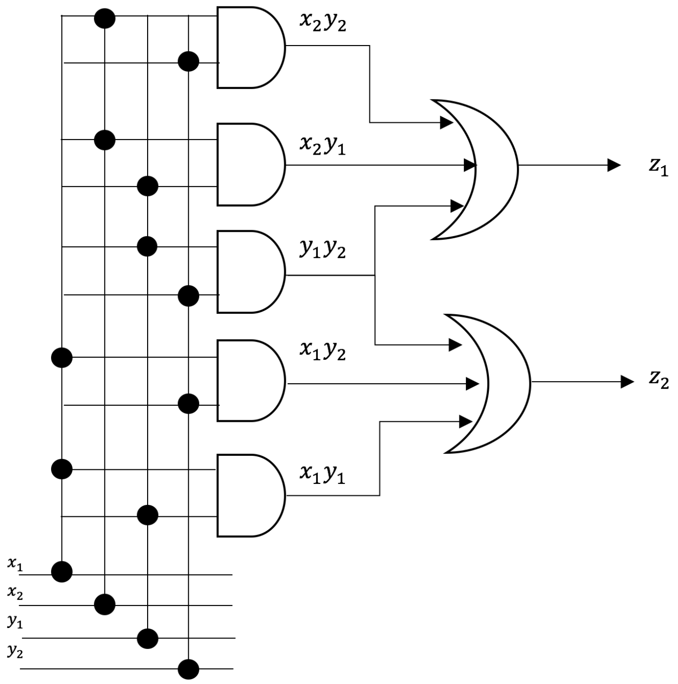


Figure 17: *Boolean circuit for the carry-semigroup multiplier.* The circuit implements Equations 42 and 43, with 7 gates and critical length = 2.

$$X(q) = x_1x_2 + x_1q + x_2q. \tag{44}$$

For example, if $X = R$, $x_1 = 0$ and $x_2 = 0$, which sets the right hand side to 0, in agreement with the fact that $R(q) = 0$, for $q \in \{0, 1\}$. Similarly, if $X = S$, $x_1 = 1$ and $x_2 = 1$, which sets the right hand side to 1, in agreement with the fact that $S(q) = 0$, for $q \in \{0, 1\}$. Finally, when $X = P$, $\{x_1, x_2\} =$

$\{0, 1\}$, and in either case the right hand side becomes q, in agreement with $P(q) = q$. If the initial carry is set to $q(0) = 0$, then Equation 44 simplifies to $X(0) = x_1 x_1$, implementable by just an AND gate.

5.5 The TRT Adder

We can now assemble all the pieces of the parallel adder. The part that computes the carries is shown in Figure 18. Another stage (not shown in the figure to avoid cluttering it) is needed to get the sum bits, according to equation:

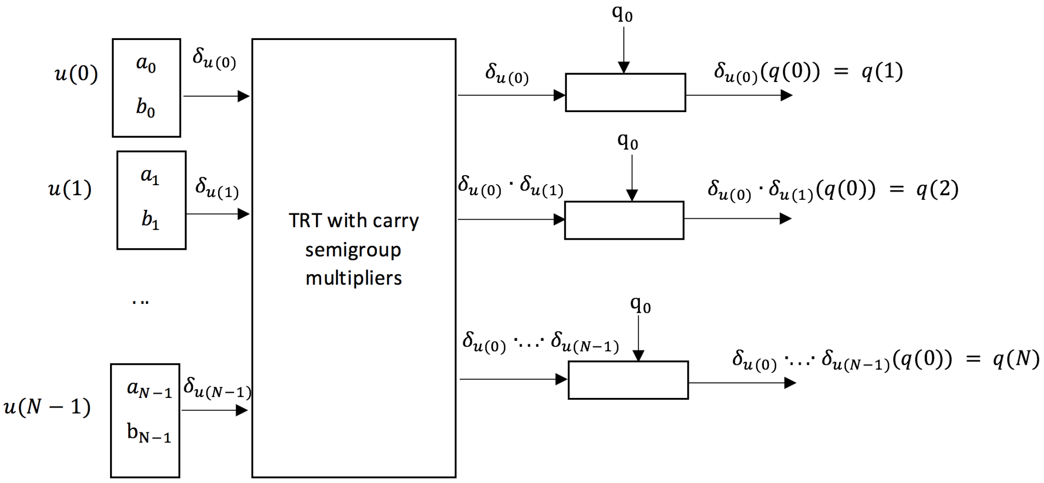

u(0)
a_0
b_0
δ_{u(0)}
u(1)
a_1
b_1
δ_{u(1)}
...
u(N − 1)
a_{N−1}
b_{N−1}
δ_{u(N−1)}
TRT with carry semigroup multipliers
q_0
δ_{u(0)}
δ_{u(0)}(q(0)) = q(1)
q_0
δ_{u(0)} · δ_{u(1)}
δ_{u(0)} · δ_{u(1)}(q(0)) = q(2)
q_0
δ_{u(0)} ·...· δ_{u(N−1)}
δ_{u(0)} ·...· δ_{u(N−1)}(q(0)) = q(N)

Figure 18: *The TRT adder.* The scheme implements the FSM evolution algorithm for M_{add} od Section 4.3. With the chosen representation of the carry-semigroup elements, the stage transforming each input into a semigroup element is trivially the identity function. The TRT stage is that of Figure 9, with the semigroup multipliers realized as in Figure 17. The evaluation at $q(0)$ is done via Equation 44. Not shown is the stage that computes the sum bits, according to Equation 45.

$$s_t = z(t) = a_t \oplus b_t \oplus q_t, \qquad \text{for } t = 0, 1, \ldots, N-1. \tag{45}$$

Finally, $s_N = q_N$. Using Proposition 8 and a simple additional analysis, we conclude as follows.

Proposition 10 *The TRT adder produces the sum of two N-bit numbers, with $O(N)$ Boolean operations and $O(\log_2 N)$ critical length.*

6 Parallel Discrete Fourier Transform

The Fourier transform is one of the most impactful ideas in mathematics, science, and engineering, including information engineering. The Fourier transform can be defined for functions over several types of domain and range. The domain, for example, can be a finite or an infinite interval in continous (real) or discrete (integer) time, or a corresponding multidimensional version of such intervals. The range is usually algebrically structured as a ring, often as a field. The complex field is probably the most common case, but Galois fields and other forms of finite rings are also used (*e.g.*, in signal processing, in coding theory, and in cryptography). The relevance of the Fourier transform has different roots, some of which are briefly mentioned next.

1. *Time-invariant, linear systems.* Time-invariant linear systems are very common models in physics and engineering. In such systems, the eigenfunctions are exponential functions, such as $\exp(i2\pi ft)$, with the frequency f as the corresponding eigenvalue. The Fourier transform of a signal yields the expansion of that signal in the basis of the eigenfunctions, which is extremely useful and insightful. (For systems distributed over space, similar considerations apply to translation invariance.)

2. *Algorithmic efficiency.* Many important operations, such as polynomial product, integer product, cyclic and linear convolution, product of circulant matrices, product of Toeplitz matrices, inversion of Vandermonde matrices, etc ..., can be efficiently reduced to the Fourier transform. Therefore, an efficient algorithm to compute the transform yields efficient algorithms for all of the above operations. Moreover, at the state of the art, the most efficient known algorithms for the above operations are all based of the Fourier transform.

3. *Linear combination of independent random variables.* In probability theory, it is often useful to introduce the *characteristic function* of a random variable, which - if the random variable admits a probability density - is the Fourier transform of such density. Characteristic functions are particularly useful in the study of linear combinations of independent random variables, for example, in establishing the central limit theorem, one of the most important results of probability theory.

4. *Conjugate quantities in quantum mechanics.* In both classical and quantum physics, there are pairs of quantities that are called conjugate pairs. Examples are position and momentum, time and energy, orientation and angular momentum. In quantum mechanics, the result of measuring a quantitity is a random variable, whose probability

> density function is a function of the state of the system under measurement. For any given state, the probability densities of conjugate quantities are connected by the Fourier transform. *Heisenberg's uncertainty principle* is then a consequence of the mathmatical property that the product of the standard deviations of two densities connected by the Fourier transform cannot be smaller than a certain value. The same mathematical property tells us that, reducing the duration of a signal necessarily increases its bandwidth and, dually, reducing the bandwidth increases the duration.

A far reaching generalization of the Fourier transform arises in the representation theory of *finite groups.* The Fourier transform of a complex function defined over a finite group is a sequence of matrices (possibly of different sizes) and enjoys a convolution property with respect to a generalized notion of convolution, reflecting the structure of the group. Representation theory and Fourier transforms can be further extended to *compact topological groups*, which play a pivotal role in quantum field theories, such as QED and QCD, which are built around a group. The irreducible representations of this group, one per (matrix) component of the Fourier transform, correspond to the particles that carry the interaction (the photon and the W^+, W^-, and Z bosons in QED; the 8 gluons in QCD). The representation theory of groups was developed by mathematicians without any "applied" motivation. It is perhaps the most spectacular example, in history, of a rather esoteric mathematical theory, developed for its own sake, which has been subsequently found to describe, with high precision, a fundamental aspect of the physical world.

6.1 The Discrete Fourier Transform (DFT)

For computational purposes, Fourier analysis is always based on the *Discrete Fourier Transform* (DFT). The DFT is defined for sequences of elements in a ring. Below, we formally introduce the concept and the related notation. We assume that the reader is familiar with some basic notions about rings and roots of the multiplicative identity. At any rate, these notions are briefly reviewed in footnotes.

Definition 14 *Let $\mathcal{R} = \langle R, +, \cdot, 0, 1\rangle$ be a* ring[35]*. Let $\omega \in R$ be an N-*

[35] In this section, a *ring* is an algebric structure $\mathcal{R} = \langle R, +, \cdot, 0, 1\rangle$, where R is a set and "+" (*addition*) and "$\cdot$" (*multiplication*) are binary operations on R, such that: $\langle R, +\rangle$ is an *abelian group*, with 0 as the additive identity; $\langle R, \cdot\rangle$ is a *monoid*, with 1 as the multiplicative identity; and "+" is *distributive* with respect to "$\cdot$". By convention, the symbol "$\cdot$" is often omitted in expressions.

th principal root[36]. *Given the sequence* $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1}) \in R^N$*, its* discrete Fourier transform *(DFT) is the sequence* $\mathbf{X} = (X_0, X_1, \ldots, X_{N-1}) \in R^N$ *defined as*

$$X_i = \sum_{j=0}^{N-1} x_j \omega^{ij}, \qquad \text{for } i = 0, \ldots, N-1. \tag{46}$$

Introducing the $N \times N$ *Fourier matrix* F_ω*, with entries* $F_\omega(i,j) = \omega^{ij}$*, for* $i, j = 0, \ldots, N-1$*, one can write* $\mathbf{X} = F_\omega \mathbf{x}$*. The notation* $\mathbf{X} = DFT_\omega(\mathbf{x})$ *is also used.*

6.2 The Fast Fourier Transform (FFT)

A very efficient DFT algorithm, known as the *Fast Fourier Transform* (FFT), was published by Cooley and Tukey in 1965. It was later recognized that the key idea behind the divide and conquer strategy of the FFT had already been discovered by Gauss in 1805, in work published only posthumously, which went largely unnoticed[37].

If $N = p_1 p_2 \ldots p_k$, where each p_h is a prime, then the FFT algorithm performs $O(N(p_1 + p_2 + \ldots + p_k))$ work. When $N = 2^d$, then $k = d$ and $p_1 = p_2 = \ldots = p_d = 2$, and the work becomes $O(N \log_2 N)$, as well known.

In this section,after reviewing the FFT algorithmic scheme, we will investigate its degree of parallelism, to discover that, for N a power of two, $P^* = \theta(N)$. This is one of those cases where an excellent sequential algorithm is also an excellent parallel algorithm. The analysis of the FFT CDAG,

[36] An N-th *root of unity* in a ring is an element $\omega \in R$ such that $\omega^N = 1$. A root is called *primitive* if the powers $\omega^0, \omega^1, \ldots, \omega^{N-1}$ are all distinct. (Equivalently, if whenever $k \neq h \mod N$, then $(\omega^k - \omega^h) \neq 0$.) A root is called *principal* if, for $i \mod N \neq 0$, $\sum_{j=0}^{N-1} \omega^{ij} = 0$. (Equivalently, if whenever $k \neq h \mod N$, then $(\omega^k - \omega^h)$ is a ring *unit*, that is, it admits a *multiplicative inverse* $\zeta \in R$ such that $\zeta(\omega^k - \omega^h) = 1$.) A principal root is always a primitive root. In a general ring, a primitive root is not necessarily principal, because $(\omega^k - \omega^h)$ may differ from 0, but not admit an inverse. In a field, a primitive root is always a principal root, since all non-zero elements have an inverse. In the *complex field*, for every $N \geq 1$, $\omega_N = \exp(i2\pi/N)$ is an N-th principal root of unity and the root ω_N^k is principal if and only of k and N are relatively prime (*i.e.*, gcd(k,N)=1). In arbitrary rings, an N-th principal root exists only for special values of N. A finite field of order ν, *i.e.*, with ν elements, exisits if and only if ν is of the form p^n, with p prime; such a field is (up to isomorphism) the Galois field $GF(p^n)$. In $GF(p^n)$, an N-th principal root exists if and only if N is a divisor of $(p-1)$. Similar necessary and sufficient conditions are also known for arbitrary finite rings.

[37] This story would makes for a very interesting case study on the developments of ideas, particularly in light of the fact that Fourier introduced “his” series (in a famous publication on the theory of heat) only in 1822. One more reason, among many, for which Guass does deserve the title of *Prince of Mathematics*.

useful to determine P^*, will also reveal to us an interesting structure, which will be quite helpful when implementing the FFT in models of computation where communication costs do matter.

Informally, the FFT is based on a divide and conquer strategy applicable based on a non trivial, ordered factorization (p, q) of N, such that $N = pq$, with $p, q > 1$. It reduces a problem instance of size N to (1) q problem instances of size p; (2) a set of N multiplications; and (3) p problem instances of size q. This strategy can be applied recursively, until the subproblem sizes become prime, and are solved by a direct application of the definition (Equation 46). We call this recursive approach the FFT algorithmic scheme, since different specific algorithms can be obtained, by choosing different factorizations of the problem size at the various nodes of the recursion tree[38]. When $N = 2^d$, two choices are most typically used: the factorization $(N/2, 2)$, which - when applied at all levels of recurssion - yields what is known as the *decimation in time* FFT. Similarly, the factorization $(2, N/2)$ yields the *decimation in frequency* FFT.

Figure 19 describes the FFT algorithmic scheme more precisely. To help visualize the actions of the algorithm and analyze its correctness, the recursive case of the procedure is formulated in terms of four auxiliary $p \times q$ arrays, A_0, A_1, A_2, A_3, where (p, q) is the chosen factorization of N. The sequence $\mathbf{x}$ is initially written in A_0, in row-major order. Then, the columns of A_0 are transformed and the result is written in A_1. Then, each element of A_1 is multiplied by a suitable power of ω (traditionally called a twiddle factor, in the signal processing literature) and the result is written in A_2. The rows of A_2 are then transformed and the result is written in A_3. Finally, $\mathbf{X}$ is read out of A_3, in column-major order. For the details, we refer to Figure 19. The correctness of the algorithms is proven next.

[38] For example, if $N = 12$, at the root of the recursion tree, the following factorizations (p, q) are possible: $(2, 6), (3, 4), (4, 3), (6, 2)$. Size 4 admits only the factorization (2, 2, while size 6 admits $(2, 3)$ and $(3, 2)$.

```
**DFT**$((x_0, x_1, \ldots, x_{N-1}), \omega)$;
% $(x_0, x_1, \ldots, x_{N-1}) \in R^N$ and $\omega \in R$, an $N$-th principal root;
  **If** $N$ is prime OR $N = 1$ **then** % base case - apply DFT definititon
      **for** $i = 0, \ldots, N-1$ **do**
          $X_i = \sum_{j=0}^{N-1} x_j \omega^{ij}$;
  **else** % $N$ is composite - apply FFT divide&conquer strategy
      choose a factorization $(p, q)$ of $N$; % $N = pq$
      set up arrays $A_0, A_1, A_2, A_3 \in R^{p \times q}$;
      % *input format:* write $\mathbf{x}$ into $A_0$, in row-major order
      **for** $r = 0, \ldots, p-1$ and $s = 0, \ldots, q-1$ **do**
          $A_0(r, s) = x_{qr+s}$;
      % *column transforms:* let the $s$-th column of $A_1$ be $DFT_{\omega^q}$ of
      % the $s$-th column of $A_0$ ($\omega^q$ is a $p$-th principal root)
      **for** $s = 0, \ldots, q-1$ and $t = 0, \ldots, p-1$ **do**
          $A_1(t, s) = \sum_{r=0}^{p-1} A_0(r, s)(\omega^q)^{tr}$;
      % *twiddle factors:* multiply the $(t, s)$ entry of the array by $\omega^{ts}$
      **for** $t = 0, \ldots, p-1$ and $s = 0, \ldots, q-1$ **do**
          $A_2(t, s) = A_1(t, s)\omega^{ts}$;
      % *row transforms:* let the $t$-th row of $A_3$ be the $DFT_{\omega^p}$ of
      % the $t$-th row of $A_2$ ($\omega^p$ is a $q$-th principal root)
      **for** $t = 0, \ldots, p-1$ and $u = 0, \ldots, q-1$ **do**
          $A_3(t, u) = \sum_{s=0}^{q-1} A_2(t, s)(\omega^p)^{us}$;
      % *output format:* read $X$ from $A_3$, in column-major order
      **for** $u = 0, \ldots, q-1$ and $t = 0, \ldots, p-1$ **do**
          $X_{t+pu} = A_3(t, u)$;
  **return** $(X_0, X_1, \ldots, X_{N-1})$;
**end** % DFT
```

Figure 19: *The FFT algorithmic scheme.*

Theorem 4 (Cooley-Tukey, 1965) *For any non trivial factorization of the problem size, the FFT algorithmic scheme described in Figure 19 generates an algorithm which, on input* $(\mathbf{x}, \omega)$*, outputs* $DFT_\omega(\mathbf{x})$*.*

> **Proof:** The correctness of the base case is obvious, since it consists of a direct application of the definition of the output in terms of the input.
>
> For the divide&conquer case, we inductively assume that the scheme is correct for input sizes that are proper factors of N and we show its correctness for input size N. By composing the expressions of $A_3(t,u)$ in terms of $A_2(t,s)$, of $A_2(t,s)$ in terms of $A_1(t,s)$, of $A_1(t,s)$ in terms of $A_0(r,s)$, and of $A_0(r,s)$ in terms of x_{qr+s}, we obtain:
>
> $$\begin{aligned} A_3(t,u) &= \sum_{s=0}^{q-1}\sum_{r=0}^{p-1} x_{qr+s}(\omega^q)^{tr}\omega^{ts}(\omega^p)^{us}, \\ &= \sum_{s=0}^{q-1}\sum_{r=0}^{p-1} x_{qr+s}\omega^{qtr+ts+pus}, \\ &= \sum_{s=0}^{q-1}\sum_{r=0}^{p-1} x_{qr+s}\omega^{t(qr+s)+pus+urpq}, \\ &= \sum_{s=0}^{q-1}\sum_{r=0}^{p-1} x_{qr+s}\omega^{(t+pu)(qr+s)}, \\ &= \sum_{j=0}^{N-1} x_j\omega^{(t+pu)j}, \\ &= X_{t+pu} \end{aligned}$$
>
> where we have (a) collected the exponents of ω, (b) added $urpq = urN$ to the exponent of ω, thus multiplying by $\omega^{urN} = (\omega^N)^{ur} = 1^{ur} = 1$; (c) algebrically rearranged the exponent as $(t+pu)(qr+s)$; (d) applied the change of variable $j = qr+s$ and observed that when $r = 0,\ldots,p-1$ and $s = 0,\ldots,q-1$, we have $j = (qr+s) = 0,1,\ldots,N-1$, and (e) used the definition of DFT, with the output index $i = t+pu$. We conclude that the algorithm correctly outputs $A_3(t,u)$ as X_{t+pu}, as can be seen from the "output format" step, where we also observe that $t+pu$ ranges over the entire set $\{0,1,\ldots,N-1\}$. *QED*

Our next objective is the analysis of the complexity of the FFT. It is useful to introduce the following function of a positive integer N.

Definition 15 (Number and sum of prime factors.) *Let N be a positive integer. We define the functions* number of factors, ν, *and* sum of factors, χ, *as follows.* $\nu(1) = 1$ *and* $\chi(1) = 1$. *For* $N \geq 2$ *and* $N = \prod_{h=1}^{k} p_h$, *with* $p_1, p_2, \ldots, p_k$ *prime numbers,* $\nu(N) = k$ *and* $\chi(N) = \sum_{h=1}^{k} p_h$.

We observe that Definition 15 is well posed since, by the *fundamental theorem of arithmetic*, the factorization of N is unique, up to a permutation of the factors, which does not affect either their number of their sum.

Proposition 11 *The FFT algorithmic scheme described in Figure 19 generates algorithms with work* $T(N) = 2N\chi(N) + N(\nu(N) - 1)$ *and critical length* $L(N) = \log_2 N + 2\nu(N) - 1$.

Proof: We measure the work by the sequential time $T(N)$, counting ring operations. We can write the following recurrence:

$$\begin{aligned} T(pq) &= qT(p) + pT(q) + pq, \quad N \ factored \ as \ N = pq; & (47) \\ T(p) &= 2p^2, \quad N = p \ prime. & (48) \end{aligned}$$

For the base case, we can see that the matrix-vector multiplication takes p^2 multiplications and $p(p-1)$ additions, which we have approximated (from above) as a total of $2p^2$ operations. For a prime p, we have $\chi(p) = p$ and $\nu(p) = 1$, so we can write $T(p) = 2p^2 = 2p\chi(p) + (\nu(p) - 1)$, in agreement with the stated formula (with $N = p$).

For the recursive case, we have accounted for the execution of q column DFTs of size p, of p row DFTs of size q, and of pq multiplications by "twiddle factors." Inductively assuming the stated formula for $T(p)$ and $T(q)$, we have:

$$\begin{aligned} T(N) &= T(pq) = qT(p) + pT(q) + pq \\ &= q(2p\chi(p) + q(\nu(p) - 1)) + p(2q\chi(q) + q(\nu(q) - 1)) + pq \\ &= 2pq(\chi(p) + \chi(q)) + pq(\nu(p) + \nu(q) - 1) \\ &= 2N\chi(N) + N(\nu(N) - 1), \end{aligned}$$

where we have made use of the relationships $\chi(p) + \chi(q) = \chi(pq)$ and $\nu(p) + \nu(q) = \nu(pq)$, which are easy consequences of Definition 15, and of $N = pq$.

Since the colum DFTs, the twiddle factor multiplications, and the row DFTs are scheduled in sequence, we can write the following recurrence for the critical length:

$$\begin{aligned} L(pq) &= L(p) + 1 + L(q), \quad N \ factored \ as \ N = pq; & (49) \\ L(p) &= \log_2(p) + 1, \quad N = p \ prime. & (50) \end{aligned}$$

For the base case, we can see that the matrix-vector multiplication can be cast as a CDAG of p trees, one per inner product, with one level of multiplications and (approximately) $\log_2 p$ levels of additions (the exact value would be $\lceil \log_2 p \rceil$). Hence, for a prime p, we have $L(p) = \log_2 p + 1 = \log_2 p + 2\nu(p) - 1$, considering that $\nu(p) = 1$.

For the recursive case, assuming inductively the stated formula, we have:

$$\begin{aligned} L(N) &= L(pq) = L(p) + 1 + L(q) \\ &= \log_2 p + 2\nu(p) - 1 + 1 + \log_2 q + 2\nu(q) - 1 \\ &= \log_2(pq) + 2(\nu(p) + \nu(q)) - 1) \\ &= \log_2 N + 2\nu(N) - 1, \end{aligned}$$

where we have used $\nu(p) + \nu(q) = \nu(pq)$ and of $N = pq$. *QED*

6.3 The FFT for Powers of Two

We now consider an input size $N = 2^d \geq 2$ and a ring that admits an N-th principal root of 1. It is easy to see that $\nu(N = 2^d) = d = \log_2 N$ and that $\chi(N = 2^d) = 2d = 2\log_2 N$. Then, Proposition 11 yields $T(N) = 3N \log_2 N - N$ and $L(N) = 3\log_2 N - 1$. These bounds can be somewhat improved, by considering that some of the operations are multiplications by 1 and can be avoided (with small modifications of the algorithm). More importantly, we are interested in understanding the structure of the CDAG.

We study in detail the factorization strategy $N = (2, N/2)$. The building block is the DFT for $N = 2$. Since the only principal square root is (-1)[39], we have:

$$F_{-1} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{51}$$

so that the transform of a sequence (column vector) (z_0, z_1) is

$$\begin{bmatrix} Z_0 \\ Z_1 \end{bmatrix} = F_{-1} \begin{bmatrix} z_0 \\ z_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \end{bmatrix} = \begin{bmatrix} z_0 + z_1 \\ z_0 - z_1 \end{bmatrix}. \tag{52}$$

In words, the Fourier transform of order two computes the sum and the difference of the two input samples.

We trace now the steps of the FFT algorithm (see Figure 19). Placing $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$ into a $2 \times N/2$ array yields:

$$A_0 = \begin{bmatrix} x_0 & x_1 & \ldots & x_{N/2-1} \\ x_{N/2} & x_{N/2+1} & \ldots & x_{N-1} \end{bmatrix}. \tag{53}$$

[39] In the complex field, this is well known. In a general ring, recalling that $\langle R, + \rangle$ is an *abelian group*, element (-1) is defined as the additive inverse of 1, i.e.: $1 + (-1) = 0$. Multiplying each side by (-1) and using the distributive property yields $(-1)(1) + (-1)(-1) = 0$, whence $(-1)^2 = 1$, by straightforward algebric manipulations. Therefore, (-1) is a square root of 1. To establish that (-1) is a principal square root, we exploit the existence of the N-th pincipal root, ω. We begin by showing that $\omega^{N/2} = -1$. In fact, (a) $(\omega^{N/2})^2 = \omega^N = 1$, whence $\omega^{N/2}$ is a square root of 1; (b) $\omega^{N/2} \neq \omega^N = 1$, since ω is principal; (c) there are only two square roots of 1, thus $\omega^{N/2}$ must be (-1). Finally, if (-1) where not principal, then $1 - (-1) = 1 - \omega^{N/2}$ would have zero divisors, contradicting the principal nature of ω.

Computing the DFT of each column (of size $p = 2$), according to Equation 52, yields:

$$A_1 = \begin{bmatrix} x_0 + x_{N/2} & x_1 + x_{N/2+1} & \dots & x_{N/2-1} + x_{N-1} \\ x_0 - x_{N/2} & x_1 - x_{N/2+1} & \dots & x_{N/2-1} - x_{N-1} \end{bmatrix}. \tag{54}$$

The multiplication by twiddle factors yields:

$$A_2 = \begin{bmatrix} x_0 + x_{N/2} & x_1 + x_{N/2+1} & \dots & x_{N/2-1} + x_{N-1} \\ \omega^0(x_0 - x_{N/2}) & \omega^1(x_1 - x_{N/2+1}) & \dots & \omega^{N/2-1}(x_{N/2-1} - x_{N-1}) \end{bmatrix}, \tag{55}$$

considering that, for $t = 0$ (first row), $\omega^{ts} = 1$ and, for $t = 1$ (second row), $\omega^{ts} = \omega^s$. Next, since rows have length $q = N/2$ so that $\omega^p = \omega^2$, the DFT_{ω^2} has to be (recursively) computed for the two rows. This step yields A_3 which, in view of the output format, is

$$A_3 = \begin{bmatrix} X_0 & X_2 & \dots & X_{N-2} \\ X_1 & X_3 & \dots & X_{N-1} \end{bmatrix}. \tag{56}$$

We see that the two rows of the final array, A_3, contains the two sampling of period two of the transform. The sampling is in the frequency domain, with the even frequencies in the first row and the odd frequencies in the second row, which is at the origin of the terminology "decimation in frequency" often used for this algorithm. In the divide and conquer paradigm view, the divide phase (computing A_1 and A_2) prepares two subproblems which are then solved (computing A_3), providing the solution with no further "conquer" work.

If we take a closer look at the structure of A_2, we may recognize a similarity to that of the L/U operation in bitonic merging (see Definition 1 in Section 1.3). The similarity consists in the fact that processing occurs on pairs of the form $(x_j, x_{j+N/2})$. The difference lies in the fact that, while in L/U the minimum and the maximum of the pair elements are computed, in the FFT the sum and the "modulated" difference are computed, resulting in a two-input, two-output "mini" CDAG, for which we will use the symbol shown in Figure 20. Unfolding the FFT recursion, one obtains the CDAG illustrated in Figure 21, for $N = 8$. The CDAG has the same topology as the one we have seen for bitonic merging (Figure 4). A key difference must instead be observed about the output format. Specifically, while in the bitonic network line i outputs the component y_i of the output sequence, in the FFT network the line i outputs the component $X_{\rho(i)}$, where ρ denotes the bit-reversal permutation, defined next.

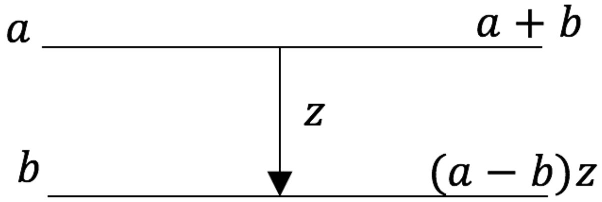


Figure 20: *The butterfly.* The symbol introduced in the figure represents a CDAG with two inputs, a and b, a “coefficient” z, which computes two outputs, $a+b$ and $a-zb$. It is a useful building block for FFT CDAGs.

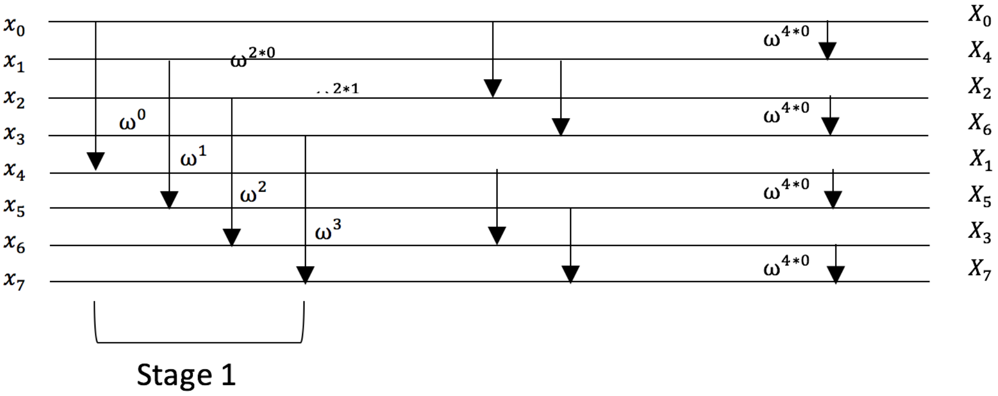


Figure 21: *The decimation in frequency FFT CDAG, for $N = 8$.* The twiddle factors in the first stage are the first 4 powers of ω (exponent $=0, 1, 2, 3$), in the second state are the first 2 powers of ω^2 (exponent $=0, 1$), and in the third stage the first power of ω^4 (exponent=0. Line i inputs x_i and outputs $X_{\rho(i)}$, where ρ denotes the bit-reversal permutation.

Definition 16 (Bit reversal permutation.) *Let $N = 2^d$. The* bit reversal permutation $\rho : \{0, 1, \ldots, N-1\} \to \{0, 1, \ldots, N-1\}$ *if the function that maps the integer $i = \sum_{j=0}^{d-1} i_j 2^j$, with $i_j \in \{0, 1\}$, into the integer*

$$\rho(i) = \sum_{j=0}^{d-1} i_{d-1-j} 2^j.$$

It is clear that the function ρ just defined is invertible, hence it is a permutation. In fact, it is an *involutory* function, meaning that it is the inverse of itself: $\rho(\rho(i)) = i$, for every i. The name given to this permutation is also self explanatory, since the binary representation of $\rho(i)$ is the string reversal of the binary representation of i. Bit reversal is an example of the class of bit permutations.

Definition 17 (Bit permutations.) *A permutation $\pi : \{0, 1, \ldots, 2^d - 1\} \to \{0, 1, \ldots, 2^d - 1\}$ is a* bit permutation *if the binary representation of $\pi(i)$ is a permutation $\sigma : \{0, 1, \ldots, d-1\} \to \{0, 1, \ldots, d-1\}$ of the representation of i. Specifically, if $i = \sum_{j=0}^{d-1} i_j 2^j$, with $i_j \in \{0, 1\}$, then*

$$\pi(i) = \sum_{j=0}^{d-1} i_{\sigma(j)} 2^j.$$

Bit permutations of various types are of interest in parallel computing, and other areas as well.

7 Off-Line Message Routing

Routing in a network is the task of transporting messages to their destinations. The *message routing* problem can have many different formulations, reflecting the contest in which it arises. Some of the dimensions of the routing problem are:

- the *topology* of the network connecting the *terminals* among which messages are exchanged (*e.g.*, a *tree*, a *torus*, a *hypercube*, a *random graph*);
- the *source-destination pattern* of messages (*e.g.*, a *broadcast*, where one source sends the same message to many destinations; a p*ermutation*, where each terminal sends exactly one message and each terminal receives exactly one message; an *all-to-all*, where each terminal send one (distinct) message to every other terminal);
- the *time when the pattern becomes known* (*e.g.*, in advance of when the body of the messages becomes known and needs to be routed, as in *off-line* routing; just when the body of the messages becomes known and needs to be routed, as in *on-line* routing);
- the *admissible algorithms*, which in turn may entail constraints along different dimensions:
  - *deterministic routing*, where the same pattern is routed always in the same way;
  - *randomized routing*, where decisions can be based on random number generation (which can be quite effective in achieving a reasonably balanced distrbution of traffic, without requiring expensive, global coordination);
  - *oblivious routing*, where the path assigned to a message depends only upon source and destination of that message (which is simple, but prone to congestion).

Routing is a fundamental problem in parallel computing, since machines do need to route messages, either direclty between processors, or between processors and memories. Given the relevant cost of communication, effective routing is crucial to good performance.

In this section, we will consider only a rather specialized type of problem, which is *off-line, permutation* routing. Some of our motivation is connected to the following points.

- Methodologically, while far from sufficient to manage communications in parallel systems, off-line permutation routing does provide a simple introduction to the subject of routing networks and algorithms, a subject which could well fill a course just by itself.

- Off-line permutation routing is a useful *benchmark* for the communication capabilities of different networks, being simple enough to admit analytical treatment, but also versatile enough to probe interesting aspects of a network bandwidth.

- Off-line permutation routing is also often instrumental in developing *simulations* of a guest network over a host network, both to efficiently port on the host algorithms written for the guest, and to compare the relative communication power of host and guest.

7.1 Problem Formulation

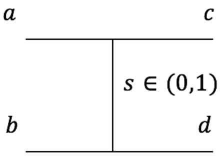
Consider a scenario where we are interested in sending messages from a set of N source terminals to a set of the same number, N, of destination terminals. Our goal is to design an efficient network, connecting the sources to the destinations. Our specific assumptions and objectives are listed next.

- The *building block* for our networks is the *binary switch*, a device with two input terminals, a and b, two output terminals, c and d, and one state or configuration bit, $s \in \{0,1\}$. If $s = 0$, (*straight* configuration), then $c = a$ and $d = b$. If $s = 1$, (*crossing* configuration), then $c = b$ and $d = a$. See Figure 22. In equations, $c = (1-s)a + sb$ and $d = sa + (1-s)b$.

- *Network efficiency* is measured both in terms of the *number of switches* (harware cost) and in terms of the *length of the paths* from inputs to outputs (routing delays).

- The *message pattern* of interest is the *permutation.* If input (source) terminals are labelled $s_0, s_1 \ldots, s_{N-1}$ and output (destination) terminals are labelled $d_0, d_1 \ldots, d_{N-1}$, a pattern will be defined as a permutation
$$\pi : \{0,1,\ldots,N-1\} \to \{0,1,\ldots,N-1\},$$
which specifies that the message originally at s_i has to be delivered at $d_{\pi(i)}$.

- We target the design of *fully rearrangeable permutation networks* where, for each permutation π, there exists at least one configuration of the network switches that simultaneouly connects each source terminal s_i to the corresponding destination terminal $d_{\pi(i)}$.
- We will focus on *off-line* routing and are interested in efficient *switch configuration* algorithms which, given a permutation π, compute an assignment of state for each switch in the network that realizes π.



$s = 0$	$s = 1$
$c = a$	$c = b$
$d = b$	$d = b$

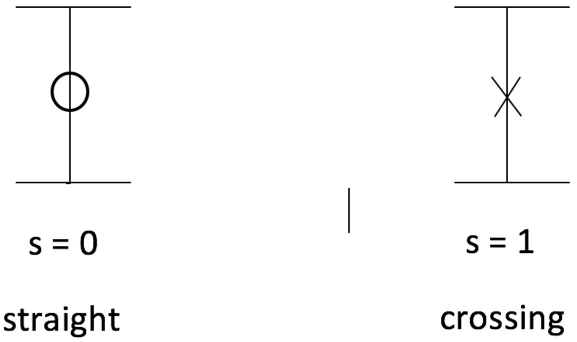


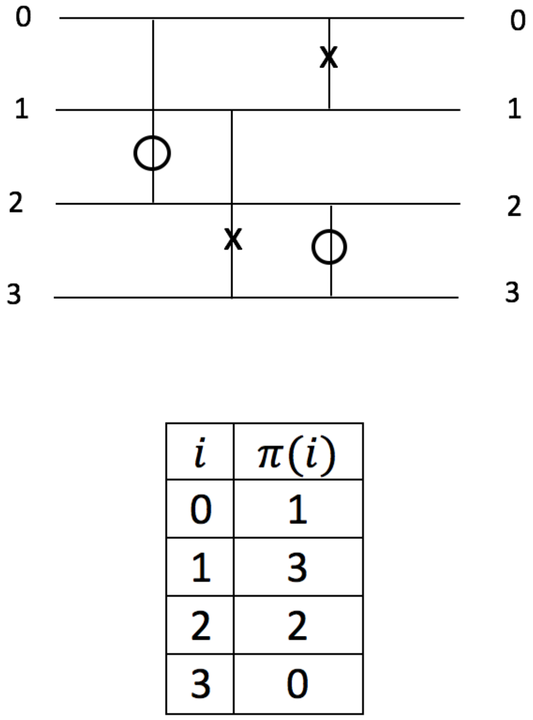
Figure 22: *Binary switch.* Top: Two inputs a and b are connected to two outputs c and d. A state bit s decides which of the two possible permutations is realized between inputs and outputs. Bottom: terminology and graphical symbols for the two switch configurations.

7.2 Lower Bounds and Connection to Sorting

How many switches are needed to realize all the $N!$ permutations? A limit from below can be obtained by a simple counting argument, which yields the next result.

Proposition 12 *The number S of binary switches of any network that can realize all permutations from N inputs to N outputs satisfies the lower bound*

$$S \geq \log_2(N!) \geq N \log_2 N - (\log_2 e)N + (\text{l.o.t.}), \tag{57}$$



i	$\pi(i)$
0	1
1	3
2	2
3	0

Figure 23: *A small switching network.* Top: A network with 4 lines connected by 4 switches, shown in a given configuration. Bottom: The permutation π realized by the network in the configuration shown in the top part of the figure.

where $\log_2 e \approx 1.44$ *and l.o.t. stands for "lower order terms." The length* L *of the longest path from an input to an output satisfies*

$$L \geq \log_2 N. \tag{58}$$

Proof: Since each switch can be in one of two states, the total number of configurations of the network is 2^S. Clearly, there must be at least one configuration for each permutation that the network is capable of realizing. Therefore, if the network can realize each of the $N!$ permutations, we have

$$2^S \geq N!$$

By taking the logarithm to the base 2 of each side, we obtain the leftmost inequality in 57. The rightmost inequality follows from Stirling's approximation of the factorial. [40]

The proof of Inequality 58 is a simple inductive argument similar to the one given in Proposition 7, considering that each output terminal must be reachable from N distinct input terminals. *QED*

A comparator exchanger can be viewed as a switch whose configuration is determined by the relative values of the input keys, as opposed to being independently set from outside. A sorting network can act as an *on-line* permutation router: if π is a permutation of $(0, 1, \ldots, N-1)$ and the record input on line i is given $\pi(i)$ as a key, sorting will bring that record on output line $\pi(i)$. Therefore, if in a sorting network we replace the comparator exchangers with switches, we obtain a permutation network.

The above discussion tells us that, from the bitonic sorter, we can obtain a switching network with $\theta(N(\log_2 N)^2)$ switches. On the other hand, since sorting is a more complex operation than off-line routing, there is a reasonable hope to improve this bound, as will be done in the next section.

It may be instructive to compare the $\log_2(N!)$ lower bound on the number of switches (Equation 57) with the analogous lower bound on the number of comparisons for comparison-based sorting algorithms, that the reader is likely to be familiar with. Clearly, the permutation routing lower bound directly applies to networks of comparator exchangers. For general comparison-based sorting algorithms, the spirit of the lower bound argument is the same: the comparisons must be sufficient to determine the permutation that applied to the input sequence produces a sorted sequence. However, the technicalities of the argument are a bit more involved, because the algorithm can choose the next comparison as a function of the results of the previous ones. As a consequence, different inputs may require a different number of comparisons and the lower bound only applies to the worst-case (as well as to the average case), but not to the best case.[41]

[40] The approximation for the factorial function named after Stirling (1692-1770), which refines an earlier result by de Moivre (1667-1754), is:

$$N! = \sqrt{2\pi N}(N/e)^N(1 + O(1/N)).$$

A tighter bound, established by Robbins (in 1955), is

$$\sqrt{2\pi N}(N/e)^N e^{1/(12N+1)} < N! < \sqrt{2\pi N}(N/e)^N e^{1/(12N)}.$$

By taking logarithms and other simple manipulations, one can see that the lower order term in 57 is very close to $(1/2)\log_2(2\pi N) + 1/(12N)$, which is quite negligible.

[41] For example, an algorithm could start by checking whether the input is already sorted, by executing $N-1$ comparisons, one per pair of consecutive inputs. If the input is indeed sorted, then it can be output without further comparisons, achieving an $N-1$ best case.

7.3 Beneš̃ **Permutation Network**

Beneš̃ developed the network that goes under his name in 1965, as part of a study of telephone networks based on *circuit switched* routing, where the path between source and destination is established once and then kept constant for all the duration of the telephone call. Today, by and large, network routing is *packet switched*, with different packets of the same "conversation" possibly travelling on different paths. Nevertheless, the results of Beneš̃ remain of great interest. The design of the Beneš̃ network follows a divide and conquer strategy, described next.

Definition 18 *The* Beneš̃ permutation network*, denoted $BN(N)$, is defined for any number of input terminals of the form $N = 2^d \geq 2$. The base case, $BN(2)$, is a simple binary switch. For $N = 2^d \geq 4$, $BN(N)$ is the cascade of (i) a "divide" stage, with $N/2$ parallel switches; (ii) two networks $BN(N/2)$ that can concurrently process $N/2$ inputs each; and (iii) a "conquer" stage, with $N/2$ parallel switches. In the divide stage, the outputs of each switch are connected to two homologous input terminals of the two copies of $BN(N/2)$. Simmetrically, any two homologous output terminals of the two copies of $BN(N/2)$ are connected to the inputs of one switch in the conquer stage.*

The network $BN(8)$ is illustrated in Figure 24.

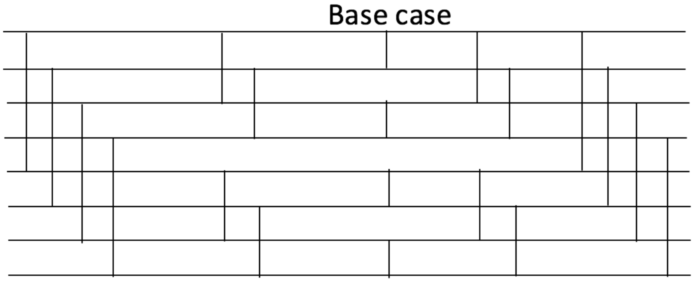


Figure 24: *The* Beneš̃ *network on $N = 8$ inputs.*

The performance analysis of the Beneš̃ network is quite straightforward.

Proposition 13 (BN network - performance.) *The network $BN(N)$ includes $S_{BN}(N) = N \log_2 N - (1/2)N$ binary switches organized into $L_{BN}(N) = 2 \log_2 N - 1$ levels.*

Proof: From the recursive construction and the base case, we can easily obtain the recurrence of the number of switches:

$$\begin{aligned} S_{BN}(N) &= 2S_{BN}(N/2) + N, \qquad N \geq 4, \\ S_{BN}(2) &= 1. \end{aligned}$$

Similarly, for the number of levels:

$$\begin{aligned} L_{BN}(N) &= L_{BN}(N/2) + 2, \qquad N \geq 4, \\ L_{BN}(2) &= 1. \end{aligned}$$

Both recurrences are quite typical and standard techniques give the stated solution. It may be interesting to observe that $S_{BN}(N) = L_{BN}(N)(N/2)$, since there are exactly $N/2$ switches per level. *QED*

The Ω and Ω^{-1} networks. The first $\log_2 N$ stages $BN(N)$ form what is known as the $\Omega(N)$ switching network. The last $\log_2 N$ stages $BN(N)$ form what is known as the $\Omega^{-1}(N)$ (read omega inverse) switching network. The sequences of stages in $\Omega(N)$ and $\Omega^{-1}(N)$ are one the reverse of the other.

It is interesting to observe that the $\Omega(N)$ can be obtained by the bitonic merging network $BM(N)$, simply by replacing each comparator exchanger by a switch. Thus, the two networks have the same topology, a topology which we have also encountered in the FFT network.

Sometimes, we will write $BN(N) = \Omega(N)\Omega^{-1}(N)$, to indicate that the Beneš̃ network is equivalent to the cascade of an omega and an omega inverse network. To be precise, in $\Omega(N)\Omega^{-1}(N)$, the two central stages are the same; however, since the cascade of two switches can perform just the two permutations realizable by one switch, we can equate the network to $BN(N)$.

The networks $BN(N)$ and $\Omega(N)\Omega^{-1}(N)$ can realize all permutations. Since $\Omega(N)$ has $(N/2)\log_2 N$ switches, it is a simple corollary of Inequality 57 that it cannot realize all permutations. It is natural to wonder whether $\Omega(N)\Omega(N) = \Omega(N)^2$ can; the number of switches does not violate the lower bound 57, but this is only a necessary condition. Somewhat surprisingly, after about half a century since this question was first asked, the answer has not yet been discovered.[42] It is instead known that $\Omega(N)^3$ can realize all permutations.

[42] Generally, if two problems appear similar, we expect their solutions to have similar degrees of difficulty. This is a reasonable guideline, but we have to be aware that there are exceptions. The Ω^2 problem is one of them; its similarity with the $\Omega\Omega^{-1}$ case turns out to be misleading.

7.4 A Configuration Algorithm for the Benes̃ Network

In this section, we outline an argument that $BN(N)$ can realize all permutations between its input and output terminals. The argument is based on an algorithm which, given a permutation, determines one setting of the network switches that realizes that permutation.

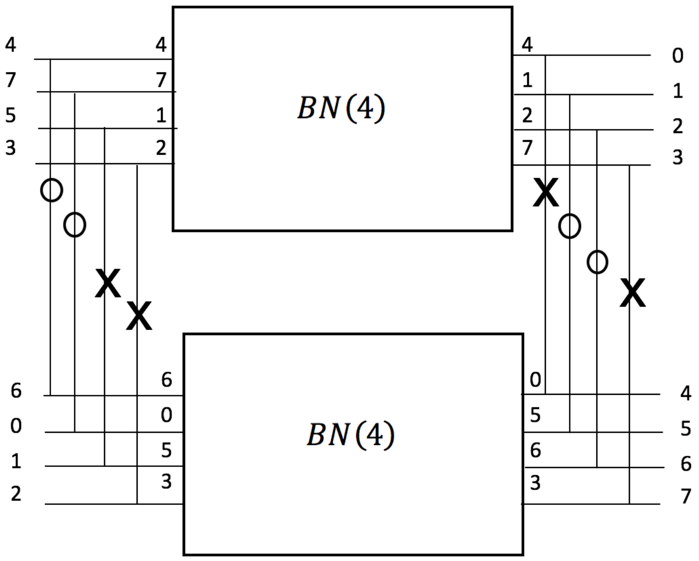


Figure 25: *Setting the switches of BN to realize a given permutation.* A $BN(8)$ is shown, decomposed into a "divide" stage, two $BN(4)$ subnetworks, and a "conquer" stage. The permutation to be routed is $\pi = (4, 7, 5, 3, 6, 0, 1, 2)$ and is shown by indicating, at each input terminal $i \in \{0, 1, 2, 3, 4, 5, 6, 7\}$, the index $\pi(i)$ of the output terminal to which the message at input i has to be sent. The configuration algorithm alternates between setting one switch in the divide stage and one in the conquer stage. At the same time, it determines which permutations have to be routed by the two $BN(4)$ subnetworks. In these subnetworks, the terminals to be connected are identified by the same number, which is the ultimate destination of the message going through. For the steps of the algorithm, please refer to the explanation in this section. The switches are processed in the following order, where [m] indicates that the message destined to output terminal m is routed: $sw_0(0, 4)[4], sw_4(0, 4)[0]$; $sw_0(1, 5)[7], sw_4(3, 7)[3]$; $sw_0(3, 7)[2], sw_4(2, 6)[6]$; $sw_0(2, 6)[5], sw_4(1, 5)[1]$.

We will describe this algorithm informally, based on an example illustrated in Figure 25, where a particular permutation has to be routed on $BN(8)$.

In general, we assume that the lines of a network are numbered from 0 to $N-1$, from top to bottom (thus, from 0 to 7, in the example). The permutation is specified by showing, at each input terminal i, the number of the output terminal $\pi(i)$ to which the message has to be sent. In the example, the input of line 0 is labeled 4, hence the message entering at 0 has to exit at 4, that is, $\pi(0) = 4$. Similarly, $\pi(1) = 7$, $\pi(2) = 5$, $\pi(3) = 3$, $\pi(4) = 6$, $\pi(5) = 0$, $\pi(6) = 1$, and $\pi(7) = 2$.

For $N = 2^d$, the network $B(N)$ has $L_{BN}(N) = 2\log_2 N - 1 = 2d - 1$ stages (see Proposition 13), which we number from 0 to $2d-2$. We denote by $sw_j(h, k)$ a switch in $stage(j)$ (with $j = 0, 1, \ldots, 2d-2$) between lines h and k (with $h, k \in \{0, 1, \ldots, N-1\}$). In the example, $N = 8$ and $d = \log_2 8 = 3$; the network $BN(8)$ (see also Figure 24) has $2 * 3 - 1 = 5$ stages, numbered $0, 1, 2, 3, 4$.

Throughout, we denote messages by their destination in square brackets. Thus, the message at input terminal i is $[\pi(i)]$. To practice with the notation we have introduced, in the running example, message [5] enters at line 2 of $BN(8)$, then it goes through switch $sw_0(2, 6)$ (set to crossing), which sends it to line 6 of $BN(8)$, that is, line 2 of the bottom $BN(4)$. $BN(4)$ will route [5] to its output line 1, which is line 5 of $BN(8)$. Finally [5] goes through switch $sw_4(1, 5)$ (set to straight) and is (correctly) output on line 5 of $BN(8)$.

The procedure to configure $BN(N)$ sets first the switches in $stage(0)$ and in $stage(2d-2)$; then it recursively processes the two $BN(N/2)$ subnetworks. In the running example, the focus is on the following switches:

$$stage(0): \quad sw_0(0,4) \;\; sw_0(1,5) \;\; sw_0(2,6) \;\; sw_0(3,7),$$

$$stage(4): \quad sw_4(0,4) \;\; sw_4(1,5) \;\; sw_4(2,6) \;\; sw_4(3,7).$$

The procedure begins by arbitrarily (i) selecting one switch in $stage(0)$; (ii) selecting a setting for the chosen switch; (iii) selecting an input message $[m]$ to the chosen switch; and (iv) *tracing* $[m]$, i.e., identifying which subnetwork will route $[m]$ together with the corresponding input and output lines. The procedure works correctly for any possible selection; however, to be definite, we will (i) choose $sw_0(0, N/2)$; (ii) set it straight; and (iii) select the message on its top input, $[\pi(0)]$. In the example, the switch chosen and set to straight is $sw_0(0, 4)$ and the message to be traced is [4].

Tracing [4] consists of the following steps:

- switch $sw_0(0, 4)$ being straight will leave [4] on the same line, therefore [4] will enter the top $BN(4)$ subnetwork at line 0;

- considering that the two subnetworks are disconnected, [4] will be output by the top subnetwork;

- by definition, [4] has to be output by line 4 of $BN(8)$; therefore [4] must arrive at the top input of $sw_4(0,4)$, the only switch in the last stage connected to line 4, hence at line 0 of the top subnetwork;

- since [4] arrives at the top line of $sw_4(0,4)$ and must exit at the bottom line, switch $sw_4(0,4)$ must be set to crossing.

The tracing of [4] has been enabled by the arbitrary setting of $sw_0(0,4)$. To continue the procedure, we now build on the setting of $sw_4(0,4)$ and, more specifically, by tracing message [0], the other output of this switch. The tracing of [0] proceeds backward, from output to input; the mechanism is the same that we have used forward for message [4]. In particular:

- switch $sw_4(0,4)$ being crossing means that [0] must come from the bottom line; therefore [0] will exit the bottom $BN(4)$ subnetwork at line 4;

- considering that the two subnetworks are disconnected, [0] will be input by the bottom subnetwork;

- we see that [0] is input by line 5 of $BN(8)$; therefore [0] must arrive from the bottom line of $sw_0(1,5)$, the only switch in the fisrt stage connected to line 5;

- since [0] is both input and output by the bottom line of $sw_0(1,5)$, this switch must be set to straight.

We can next focus on tracing forward message [7], the other input of $sw_0(1,5)$. And so on, alternating forward and backward tracing. The reader is invited to continue this process, which - once understood - is easier to execute directly than it is to read; the results can be also checked against Figure 25.

After tracing [7] (forward), [3] (backward), [2] (forward), and [6] (backward), we encounter a new situation. The switch in $stage(0)$ through which we trace [6] is $sw_0(0,4)$, which must be set straight, to handle [6] correclty. Attention! We have already set this switch, arbitrarily, to get the process started. Luckily, we had set it straight, so we can satisfy the requirement posed by [6]. But what if [6] required to set the switch in the crossing state? *It can be shown that such a contradictory requirement is never created by the algorithm.* We emphasize this property, because it is crucial to the correctness of the algorithm.

Another issue has to be handled: how do we continue when a message is traced back to a switch that had already been set? This is a simple issue, because it suffices to act as at the start, by arbitrarily selecting a switch not yet set, a setting for it, and one of its two input messages to be traced.

Once the tracing is complete, not only all the switches in the first and last stage of $BN(N)$ have been configured, but a specific permutation between inputs and outputs has been determined for each of the two $BN(N/2)$ subnetworks, on which we can proceed recursively. The bottom of the recursion occurs for $N = 2$. Since $BN(2)$ consists, by definition, of just one switch, it is straightforward to configure it to realize the required permutation: if $\pi = (0, 1)$ then set $sw_0(0, 1)$ to straight $(s = 0)$ else $(\pi = (1, 0))$ set $sw_0(0, 1)$ to crossing $(s = 1)$.

The preceding discussion illustrates all the key ideas upon which the configuration algorithm is built and could be proven correct. In these notes, we will not insist on a more formal description of the algorithm nor on a very detailed proof of its correctness. We will just state the key result and provide a sketch of the proof.

Theorem 5 *The* configuration algorithm *described in this section does determine, for every permutation π of order N, a setting of the switches of the* Beneš̃ *network $BN(N)$, that realizes π. As a corollary, $BN(N)$ is a* fully rearrangeable permutation network.

> **Proof:** (Sketch.) The proof is by induction on d. The base case, for $d = 1$, hence $N = 2$, is trivial, since $BN(2)$ is just a switch.
>
> For $d \geq 2$, the argument is based of the reduction - provided by the configuration algorithm - of an arbitrary permutation π of order N to two permutations π_1 and π_2, of order $N/2$, to be realized by the two subnetworks $BN(N/2)$. By the inductive hypothesis, $BN(N/2) = BN(2^{d-1})$ is fully rearrangeable, whence the subnetworks can be configured to route the required permutations.
>
> The key point remains then the feasibility of such reduction. It is not difficult to see that the configuration algorithm could "malfunction" only at the end of a backward tracing phase for some message $[m]$, if a switch already set were involved, and the setting required by $[m]$ were different from the setting already assigned to the switch.
>
> To see why such a scenario cannot arise, we observe that, if the tracing of a message $[m'']$ is enabled by the tracing of a message $[m']$, then the two messages go through the same switch (either in $stage(2d - 2)$, if $[m']$ is traced forward, or in $stage(0)$, if $[m']$ is traced backward). Therefore, $[m']$ and $[m'']$ must be routed through different subnetworks. When a switch in $stage(0)$ is reached for the second time, say by a message $[m]$, this happens after an even number of tracings, hence message $[m]$ must come from the other subnetwork with respect to the message $[m_0]$ arbitrarily selected to be traced the first

time. In conclusion, $[m_0]$ and $[m]$ pose for that switch the same (satisfiable) requirement, given that they go to different subnetworks. *QED*

Although, in off-line routing, the configuration time does not affect the routing time, it is still useful to achieve fast configuration. Our algorithm is indeed quite efficient.

Proposition 14 *The configuration algorithm for the* Benes̃ *permutation network $BN(N)$ computes the setting of the switches for a given permutation, in optimal sequential time $T(N) = \theta(N \log_2 N)$.*

Proof: The configuration algorithm reduces a problem of size N (setting the switches of $BN(N)$) to two subproblems of size $N/2$ (setting the switches of the two $BN(N/2)$ subnetworks), with $O(N)$ work, that is, $O(1)$ work for each of the N switches in the first and in the last stage of $BN(N)$.

Then, $T(N) = 2T(N/2) + O(N)$, with $T(2) = O(1)$, which yields $T(N) = O(N \log_2 N)$, as well known.

Any configuration algorithm for the Benes̃ network has to output $N \log_2 N - N/2 = \Omega(N \log_2 N)$ bits, one per switch. Naturally, sequential time will be subject to the lower bound $T(N) = \Omega(N \log_2 N)$. We can then conclude that the configuration algorithm presented in this section is optimal. [43] *QED*

The configuration algorithm presented in this section is highly sequential, since switches are enabled for processing one at the time. Parallel algorithms for configuring permutation networks are known, for example, with critical length $O((\log_2 N)^2)$ and work $O(N(\log_2 N)^2)$.[44]

[43] In fact, for any fully rearrangeable permutation network, the sequential configurationn time must satisfy $T(N) = \Omega(S(N)) = \Omega(N \log_2 N)$, considering lower bound 57 on the number of switches.

[44] A pioneering paper on this topic is by G. Lev, N. Pippenger, and L. Valiant, A fast parallel algorithm for routing in permutation networks, IEEE Transactions on Computers, vol.30, pp. 93-100, 1981. We have already mentioned Pippenger for his formulation of the class NC. Leslie Valiant has contributed to several research areas in computer science (including parallelism and learning) as recognized in the citation of his 2010 Turing Award.

8 Problems

8.1 Parallel Merging and Sorting

Problem 1.2.1. - Parallel Satandard Merge Sort. The objective of this problem is to estimate the practical potential of the standard merge sort, on a small multiprocessor. Assume that one processor can execute 1 comparison in 1 $ns = 10^{-9}$ s. Consider sorting a sequence of $N = 2^{40} = 1.10$ 10^{12} elements.

- Evaluate the time (in seconds and in hours) to merge sort the sequence on $P = 1$ processor, assuming $T(N, 1) = N \log_2 N$ ns.
- Let now $P_0 = 2^{\lfloor \log_2 \log_2 N \rfloor} = 32$. Assuming $T(N, P_0) = 4N$ ns, evaluate (in seconds and in hours) the time $T(2^{40}, 32)$ to sort the sequence, with $P_0 = 32$ processors.
- For what fraction of the time are the $P_0 = 32$ processors utilized, on average?

Problem 1.2.2. - Overlapped, Parallel Satandard Merge Sort. In Section 1.2, we have analized the parallelization strategy based on the recursion tree:

- Sequentially execute one level of the tree at the time, from the leaves to the root. (Starting at the leaves is possible, essentially because all the work of merge sort is in the conquer phase.)
- Concurrently execute all the nodes in the same tree level.

In this problem, we explore the possibility of obtaining further parallelism by overlapping the execution of the work pertaining to different levels of the tree.

1. Observe that the merging algorithm consumes its input sequences and produces its output sequence in non decreasing order. Conclude that, for $\ell = \log_2 N - 2, \ldots, 1, 0$, the merging procedure at level ℓ can start just one step after the merging procedure at level $\ell + 1$.
2. Schedule the merging operations based on the observation of the previous point. For each level ℓ, specify the starting time t^s_ℓ and the finish time t^f_ℓ of the merging at level ℓ, assuming $t^s_{\log_2 N - 1} = 0$.
3. Compute $T_{sort}(N) = t^f_0$.

4. Argue that $N/2$ processors are sufficient to implement the schedule.

5. By what factor has “overlapping” increased the parallelism?

6. What is the maximum number of levels that are simultaneaously active?

Problem 1.3.1. - Bitonic sequences. Let $\mathbf{x} = (x_0, \ldots, x_{N-1})$ be bitonic.

- Show that any subsequence of $\mathbf{x}$ is bitonic.
- Show that the reversal of $\mathbf{x}$, that is, $\mathbf{x}^R = (x_{N-1}, \ldots, x_0)$, is bitonic.
- If $\mathbf{xx}$ is bitonic, what can we say of $\mathbf{x}$?
- If $\mathbf{xx}^R$ is bitonic, what can we say of $\mathbf{x}$?

Problem 1.3.2. - Number of bitonic permutations. An invertible function $\pi : \{0, \ldots, N-1\} \to \{0, \ldots, N-1\}$ is called a *permutation*, of order N. A permutation π is called bitonic is the sequence $(\pi(0), \ldots, \pi(N-1))$ is bitonic. Let $B(N)$ be the number of bitonic permutations of order N.

- Show, by case analysis, that $B(1) = 1$, $B(2) = 2$, $B(3) = 6$, $B(4) = 16$.
- Show, by a combinatorial argument, that for $N \geq 2$,

  $$B(N) = N2^{N-2}$$

  .

  (Hint 1 : count first the bitonic permutations with $\pi(0) = 0$. Hint 2: a set of n elements has 2^n subsets.)
- How many bits are needed to specify a bitonic permutation of order N?
- The fraction $B(N)/N!$ of all permutations that are bitonic vanishes very rapidly with N. To get a sense, compute this fracion for $N = 10$.

Problem 1.3.3. - Sorting a bitonic sequence.

- Show how to find the position of the minimum in a bitonic sequence with $O(\log_2 N)$ comparisons, sequentially.
- Show that a bitonic sequence can be sorted with $N + O(\log_2 N)$ comparisons, sequentially.

8.2 Measuring Parallelism in Algorithms

Problem 2.2.1. - Greedy schedule. Show that every node will be in some set V_j, according to Definition 6. Hint: use Property 10 of Definition 4.

Problem 2.2.2. - Bounds on the critical length. Show that:

1. For any DAG $G = (I, V, E)$, the *critical length* satisifes the upper bound $L \leq |V|$.

2. Show that, for any $N \geq 1$, there exists a DAG with $L = |V| = N$.

Problem 2.2.3. - Greedy schedule and topological sorting. Show that, given the greedy schedule for G, one can easily derive a topological order of G. (Going in the other direction, from any topological order to the greedy schedule, requires more thought, but it can be done efficiently, as illustrated by the next problem.)

Problem 2.2.4. - Computing the greedy schedule, sequentially. Given a DAG $G = (I, V, E \subseteq (I + V) \times V)$ and a node $w \in I + V$, let $g(w)$ be the level of w in the gedy schedule of G; in nother words, let $w \in V_{g(w)}$.

Design and analyze a sequential algorithm to compute $g(w)$ for all $w \in I + V$, in linear time, $T = O(|I| + |V| + |E|)$, according to the following outline.

- Assume that the input DAG G is specified via the *predecessor adjacency list* and that the nodes are named so that $I + V = \{1, 2, \dots, |I| + |V|\}$.

- Let $w_1, w_2, \dots, w_{|I|+|V|}$ be a *topological order* of G. (This can be obtained in time $T_{topSort} = O(|I| + |V| + |E|)$, by known algorithms.)

- Let $pred(w)$ denote the set of predecessors of w (which is empty if $w \in I$). Detrmine $g(w)$ if $w \in I$. Show how, if $w \in V$ and $pred(w) = \{u_1, \dots, u_k\}$, then $g(w)$ can be obtained from $g(u_1), \dots, g(u_k)$.

- Show how $g(w)$ can be computed by scanning G in topological order and using the formula derived in the previous point.

Problem 2.2.5. - Time L with fewer than P_{max} processors. By Theorem 2, we know that $T(P_{max}) = L$. However, for some CADGs, $T(P) = L$ is achievable even for some values $P < P_{max}$. This problem explores this issue.

1. Show that, if $T(P) = L$, then $P \geq P^*$.

2. Consider a DAG $G = (I, V, E)$ with $I = \{0, 1, 2\}$, $V = \{3, 4, 5, 6\}$, and $E = \{(0, 3), (1, 4), (2, 5), (5, 6)\}$.
   (a) Draw G.
   (b) Determine $|V|$, L, $(V_0, V_1, \ldots, V_L)$, $P^* = |V|/L$, and P_{max}.
   (c) Show that $T(P^*) = L$, even though $P^* < P_{max}$.

Remark. The problem of finding $\min\{P : T(P) = L\}$, for an input DAG G, is NP-hard.

Problem 2.2.6. - Greedy schedule for bitonic merging.

1. Redraw the network $BM(8)$, shown in Figure 4, as a CDAG, with the comparison-exchanges implemented by min and max operations, as in the bottom part of Figure 3.

2. In your figure, mark the sets $V_0 = I, V_1, \ldots, V_L$.

3. Observe, in your drawing of $BM(8)$, that (i) $|V_j|$ is independent of j, that:
   (a) all levels have the same size;
   (b) all arcs go from a level to the next one, that is, if $(u, v) \in E$, $u \in V_h$ and $v \in V_j$, then $j = h + 1$;
   (c) all paths from I to V_L have the same length, L.

4. Argue that the three properties in the previous point hold for any $BM(N)$.

5. Show that, for any DAG G where $|V_j|$ is independent of j, we have $P^* = P_{max}$. Furthermore, G can be executed on $IM(P^*)$ in time L, so that all processors are active at each step.

6. Find a schedule according to whicht $BM(8)$ can be executed in time $T(6) = 4$ on $P = 6$ processors. (Compare with the bound implied by Equation 19, $T(6) < |V|/P + L = 24/6 + 3 = 7$, that is, $T(6) \leq 6$, since $T(6)$ is an integer.)

Problem 2.2.7. - Degree of parallelism: P^* vs. $P^*_{1/2}$. Recall that $P^* = |V|/L$. Also recall that $P^*_{1/2}$ is the largest value of P such that the computation can be scheduled on P processors to complete in T steps, with each processor executing at least $(1/2)T$ operations.

1. Show that $P_{1/2}$ is the largest value of P such that $|V|/T(P) \geq (1/2)P$.

2. Show that $P^* \leq P^*_{1/2}$; (derive and use the relationship $P^*T(P^*) < 2|V|$).

3. Show that $P^*_{1/2} \leq 2P^*$; (derive and use the relationship $T(2P^*) \geq L$).

8.3 Prefix Computation

Problem 3.1.1. - Associativity of Boolean operations.

1. A binary Boolean operation is a function $f: \{0,1\}^2 \rightarrow \{0,1\}$. Enumerate the 16 operations of this type.

2. Which of the 16 binary Boolean operations are associative?

Problem 3.1.2. - Prefix computation for N not a power of 2. Some algorithms for prefix computation are naturally developed in the case when N is a power of 2. One example is the TRT algorithm. In this problem, we investigate how to embed an instance of prefix computation of arbitrary size N into one of size $\hat{N} \geq N$. We will assume that our semigroup is a monoid, with an indentity element e such that $e \cdot x = x \cdot e = x$, for any x[45].

1. Let $\mathbf{y} = (y_0, \ldots, y_{N-1})$ be the prefixes of $\mathbf{x} = (x_0, \ldots, x_{N-1})$ and let $\hat{N} \geq N$. Show that the prefixes of the sequence of length $\hat{N}$,

   $$\hat{\mathbf{x}} = (x_0, \ldots, x_{N-1}, e, \ldots, e),$$

   form the sequence

   $$\hat{\mathbf{y}} = (y_0, \ldots, y_{N-1}, y_{N-1}, \ldots, y_{N-1}),$$

   from which $\mathbf{y}$ can be easily recovered.

2. Prove that, for any $N \geq 1$, we have $N \leq 2^{\lceil \log_2 N \rceil} < 2N$. Hence, by padding an input of size N to become one of size $\hat{N} = 2^{\lceil \log_2 N \rceil}$, the problem size increases by less than a factor of 2.

3. Argue that e can be easily replaced by another element, when augmenting $\mathbf{x}$, without changing the nature of the conclusion.

[45] It is easy to show that if $\langle A, \cdot \rangle$ is a semigroup without identity, then by adjoining an element e such that $e \cdot x = x \cdot e = x$ for any x, one always obtains a monoid $\langle A + \{e\}, \cdot \rangle$. Of course, an implementation of "$\cdot$" has to be adapted to handle the input e.

Problem 3.1.3. - Powers of an element. Problem: Given a semigroup element $a \in A$ and a non negative integer N, compute the first N powers of a, that is, the sequence $(a^0, a^1, \dots, a^{N-1})$.

1. Observe that the problem can be viewed as a special case of prefix computation, for a suitable sequence $\mathbf{x} = (x_0, x_1, \dots, x_{N-1})$.

2. Recalling that a^j can be computed in $O(\log_2 j)$ operations by the method of "repeated squaring", outline a parallel algorithm that solves the problem in time $T(N) = O(\log_2 N)$, with $P = N$ processors, with total work $O(N \log_2 N)$.

3. Let now N be a power of two (the general case is very similar.) Develop a divide and conquer approach involving just one subproblem of size $N/2$ and a conquer strategy involving $O(N)$ operations executable in time $O(1)$ on $N/2$ processors. Shw that this approch leads to a parallel algorithm running in time $T(N) = O(\log_2 N)$, with $P = N$ processors, with total work $O(N)$.

Problem 3.1.4. - Associativity of function composition. Let Q be a set; then Q^Q denotes the set of all functions $f: Q \to Q$. If $f, g \in Q^Q$, we call *composition* of f and g the function $f * g \in Q$ such that $(f * g)(x) = g(f(x))$, for any $x \in Q$[46].

1. Prove that the cardinality of Q^Q is $|Q|^{|Q|}$.

2. Prove that function composition is associative, that is, for any $f, g, h \in Q^Q$, we have $(f * g) * h = f * (g * h)$.

3. Conclude that $\langle Q^Q, * \rangle$ is a semigroup.

Remark. The material in this problem is very useful for the parallel simulation of finite state machines in Section 4.

Problem 3.1.5. - Cayley's theorem for monoids.
A *monoid* is a semigroup with an identity element e, such that $e \cdot x = x \cdot e = x$, for every x in the semigroup.
Theorem: Any finite monoid is isomorphic to a monoid of functions[47]. Prove the stated theorem along the following lines:

[46] Composition is usually defined so that the second operand is applied to the argument first, that is, $(f*g)(x) = f(g(x))$. The less usual definition adopted here is more convenient for the developments concerning finite state machines.

[47] If the monoid is actually a *group* (every element x has an inverse y such that $x \cdot y = y \cdot x = e$), the above isomorphism is known as Cayley's theorem: any finite group is

1. Let $\langle A, \cdot \rangle$, with A finite, be a monoid.

2. To each $u \in A$, associate the function $f_u : A \to A$ such that, fow any $z \in A$, $f_u(z) = z \cdot u$.

3. Show that $\langle B, * \rangle$, where $B = \{f_u : u \in A\}$ and $*$ is the composition operation, is a monoid. (In fact, it is a submonoid of $\langle A^A, * \rangle$.)

4. Show that the map $u :\to f_u$ is one to one, that it, $f_u = f_v$ if and only if $u = v$. (Hint: If $u \neq v$, consider $f_u(e)$ and $f_v(e)$.)

5. Show that $f_u * f_v = f_{u \cdot v}$.

Remark. This problem helps providing a perspective on semigropus and monoids and some practice with important concepts. It also highlights the generality of the semigroups based on function composition. Cayley's theorem as such is not used in this chapter.

Problem 3.2.1. - Scheduling the tree algorithm, for $N = 10$. Consider the binary tree $Tree(N)$ arising as a CDAG of the algorithm discussed in Section 3.2.

1. Determine the critical length $L(10)$ and $P_{max}(10)$, for $Tree(10)$, which is shown in Figure 8.

2. Do your best to schedule $Tree(10)$ on the ideal machine $IM(P)$ and determine the corresponding upper bounds to $T(P)$, for $P = 1, 2, 3, 4$.

3. Using work and critical length arguments, find lower bounds to $T(P)$, for $P = 1, 2, 3, 4$.

4. If your upper and lower bound do not match, try to improve them till they do.

Problem 3.2.2. - P_{max} for $Tree(N)$. Let $p(N) = P_{max}(N)$ be the critical length of $Tree(N)$. The objective of this problem is to determine an exact formula for $p(N)$. While the recurrence relation for $p(N)$ is easy to derive and has a simple exact solution when N is a power of two, the solution for arbitrary N has a "strange" behaviour, alternating constant segments to

isomorphic to a group of permutations. This theorem appeared in a 1854 paper, which introduced the modern concept of group. Too abstract for an engineer? Well, Cayley (1821-1895) was a laywer, although he eventually abandoned his flourishing practice to become a professor of mathematics at Cambridge. Group theory has several applications in computer science and in science in general.

linear segments, with the size of the segments doubling every other segment. The various steps of the problem provide a guide to the solution, while trying to illustrate the way an "expert" may attack the problem.

1. Prove that

$$\begin{aligned} p(N) &= p(\lfloor N/2 \rfloor) + (\lceil N/2 \rceil), \qquad N > 2; & (59)\\ p(2) &= 1, & (60)\\ p(1) &= 0. & (61) \end{aligned}$$

2. Consider first the case when N is a power of two, for which the recurrence simplifies to

$$\begin{aligned} p(N) &= 2p(N/2), \qquad N > 2; & (62)\\ p(2) &= 1. & (63) \end{aligned}$$

   since, throuought the recursion, $\lfloor N/2 \rfloor) = \lceil N/2 \rceil = N/2$, and the case $N = 1$ is never reached. Show that the solution is:

$$p(N) = N/2; \qquad \text{for } N \text{ a power of 2.} \qquad (64)$$

   The result is intuitive, since in this case $Tree(N)$ is fully balanced, with the N input leaves all at the same depth, with V_1 containing exactly the $N/2$ parents of the leaves, and with the subsequent levels of the greedy schedule being all smaller, so that $P_{max} = \max_j |V_j| = |V_1| = N/2$.

3. Numerically solve recurrence 59, for $P = 0, 1, 2, \ldots, 32$, to develop some intuition on the behavior of $p(N)$. Formulate an hypothesis on the shape of the function $p(N)$, which you can then try to prove by induction. You will find the answer in the next two steps; therefore, to learn the most from this step, try to do it before reading further.

4. Prove, by induction, that if $2^k < N \leq 2^k + 2^{k-1}$, then $p(N) = 2^{k-1}$. (Observe how here $p(N)$ is constant on 2^{k-1} consecutive values.)

5. Prove, by induction, that if $2^k + 2^{k-1} < N \leq 2^{k+1}$, then $p(N) = N - 2^k$. (Observe how here $p(N)$ is linear on 2^{k-1} consecutive values. Also observe that, for $N = 2^{k+1}$, $p(N) = N - 2^k = 2^{k+1} - 2^k = 2^k = N/2$, in agreement with Equation 62.)

Problem 3.2.3. - Polynomial evaluation. A *ring* is an algebric structure $\langle R, +, \cdot \rangle$, where R is a set and "+" and "$\cdot$" are binary operations on R, such that: $\langle R, +\rangle$ is an *abelian group*, $\langle R, \cdot\rangle$ is a *semigroup* (or, according to some authors, a *monoid*), and "+" is *distributive* with respect to "·".

A *polynomial* is a ring function of a ring variable z, of the form

$$P(z) = \sum_{j=0}^{N-1} a_j z^j,$$

where $a_0, \ldots, a_{N-1}$ are ring elements, called the *coefficients* of the polynomial.

Polynomial evaluation: Given a polynomial, by means of its coefficients, and a ring element z, compute the ring element $P(z)$.

1. Describe an algorithm for polynomial evaluation, with critical length $L(N) = O(\log_2 N)$ and $|V(N)| = O(N)$ operations.

2. Provide the most accurate analysis you can of both $L(N)$ and $|V(N)|$.

8.4 FSM Evolution

Problem 4.1.1. - Counting modulo 3. Consider an FSM $M = (\Sigma, \Gamma, Q, \delta, \eta)$, with $\Sigma = \Gamma = \{0, 1\}$, $Q = \{0, 1, 2\}$, and $\eta(q, a) = (q + a) \mod 3$.

1. Define the transition function $\delta(q, a)$ so that, if the initial state is $q(0) = 0$, then $z(t) = (\sum_{j=0}^{t} u(t)) \mod 3$.

2. Describe the functions $\delta_0(q) = \delta(q, 0)$ and $\delta_1(q) = \delta(q, 1)$.

3. Construct the semigroup $S_M = \langle A_M, *\rangle$ of the finite compositions of (*i.e.*, generated by) δ_0 and δ_1. Compare $|A_M|$ with $|Q|^{|Q|} = 3^3 = 27$.

4. Choose a representation of the elements of A_M by binary strings and write the Boolean functions that give the bits representing $Z = X * Y$ in terms of the bits of the representations of X and Y.

8.5 Parallel Binary Addition

Problem 5.1.1. - FSM for decimal addition. Let $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be the decimal alphabet.

1. Design an FSM $M_{ADD} = (\Sigma, \Gamma, Q, \delta, \eta)$, with $\Sigma = D \times D$, $\Gamma = D$, and $Q = \{0, 1\}$, for the addition of two natural numbers in base 10.

2. Determine the semigroup of the machine designed in the previous step.

Problem 5.2.1. - Carry semigroup. Define the binary operation $*$ over the set $A = \{P, R, S\}$ so that, for every $X \in A$, $X * P = X$, $X * R = R$, and $X * S = S$. (We may recognize that $\langle A, * \rangle$ is isomorphic to the carry semigroup.)

1. Prove that $*$ is *associative*, that is,

   $$\forall X, Y, Z \in A: \ (X * Y) * Z = X * (Y * Z),$$

   so that $S = \langle A, * \rangle$ is a semigroup. Suggestion: try a case analysis on Z.

2. Prove that $*$ is *idempotent*, that is, $\forall X \in A : X * X = X$.

3. Prove that $*$ is *not commutative.*

4. Prove that S is a *monoid.*

5. List the elements of S that have an *inverse.*

Problem 5.2.2. - Inputs of addition that only propagate the carry. With reference to the FSM M_{add} for binary addition, we are told that the prefixes computed in Step 4 of the evolution algorithm (Section 4.3) are all "carry propagate." In equations, $\xi_t = P$, for $t = 0, 1, \ldots, N-1$.

1. What is the value of the sum $\sum_{t=0}^{N} s_t 2^t$?

2. How many different pairs of inputs satisfy the specified condition?

3. In general, is $\sum_{t=0}^{N} s_t 2^t$ uniquely determined by the prefixes $(\xi_0, \xi_1, \ldots, \xi_{N-1})$? Justify your answer.

8.6 Parallel Discrete Fourier Transform

Problem 6.1.1. Principal versus primitive roots. Consider the *ring of the integers modulo 15.*

1. Partition the numbers $\{0, 1, \ldots, 14\}$ into units and zero divisors. Examples: 4 is a *unit*, since $4 \cdot 4 \mod 15 = 1$, so that $4^{-1} \mod 15 = 4$, while 3 is a *zero divisor*, since $3 \cdot 5 \mod 15 = 0$. Hint: there are 8 units and 7 zero divisors.

2. Show that 7 is a 4-th *primitive* root of 1, but it is not a *principal* root. Hint for primitive : show that $7^1, 7^2, 7^3, 7^4 = 1$ are four distinct elements of the ring. Hint for principal: show that there exist a pair of these elements whose difference is a zero divisor.

3. Compute $\det(F_7)$. Is it a zero divisor?

4. Show that 14 is a principal root of order 2 and write the *Fourier matrix* F_{14}.

5. Compute $\det(F_{14})$. Is it a unit?

6. Observe that $14 = -1$ so that $F_{14} = F_{-1}$. Compare with Equation 51.

Errata and updates. In the first version of this problem, there were two typos: 7 is a 5-th root (instead of 4-th); 4 is a principal root (instead of 14). Further help to check the answers has been also added.

Problem 6.1.2. Fourier and Vandermonde.

1. Show that the Fourier matrix F_ω is a symmetric, Vandermonde matrix[48].

2. Write the determinant of F_ω.

3. Show that, if ω is principal, then the value of the determinant is a unit in the ring (which is necessary and sufficient for a matrix to be invertible). Hint: the product of two units is a unit.

4. Show that if ω is primitive, but not principal, then the value of the determinant is a zero divisor in the ring (which is necessary and sufficient for a matrix to be singular). Hint: the product of a zero divisor by any element is a zero divisor.

Problem 6.1.2. F_ω with ω not principal. The requirement that ω be principal, in the definition of the DFT, stems form the interest in having an *invertible* transform. A key property of the DFT is the cyclic convolution property (CCP): the transform of the convolution is the component-wise product of the transforms of the sequences being convolved. Without invertibility, the CCP would be considerably less useful.

1. Show that the CCP is satisfied by F_ω even if ω is primitive but not principal. (You can locate the proof in a textbook and check the steps to make sure that the fact that ω is principal is never used.)

[48] If $\mathbf{y} = (y_0, y_1, \ldots, y_{N-1}) \in R^N$, the corresponding *Vandermonde matrix* $V(\mathbf{y})$ has components $V(i,j) = (y_i)^j$, for $i, j = 0, 1, \ldots, N-1$. It is well known that

$$\det(V(\mathbf{y})) = \prod_{0 \le h < k \le N-1} (y_k - y_h).$$

2. Argue that, if F_ω were not invertible, then the CCP would not enable the computation of the convolution via the DFT.

The CCP characterizes the Fourier tranform, in the sense that any invertible, linear transform is a DFT. This is not too difficult to prove, but it takes some time and some clarity of mind.

Problem 6.2.1. - Number and sum of prime factors. Let $\nu(N)$ and $\chi(N)$ be the functions introduced in Definition 15. Prove the following relashionships:

1. $\nu(NM) = \nu(N) + \nu(M)$.
2. $1 \leq \nu(N) \leq \log_2 N$. What can we conclude from $\nu(N) = 1$? What can we conclude from $\nu(N) = \log_2 N$? (Argue your answer.)
3. $\chi(NM) = \chi(N) + \chi(M)$.
4. $2 \log_2 N \leq \chi(N) \leq N$. What can we conclude from $\chi(N) = 2 \log_2 N$? What can we conclude from $\chi(N) = N$? (Argue your answer.)

Problem 6.2.2. - N perfect square. Let $N = p^2$, with p prime. Let ω be an N-th principal root of 1.

1. Find the value of $\nu(N) = \nu(p^2)$ and of $\chi(N) = \chi(p^2)$.
2. Determine the work $T(N)$, the critical length $L(N)$, and the degree of parallelism P^* of computing DFT_ω via FFT.
3. Let $V_0, V_1, \dots V_L$ the levels of the FFT CGAD. Determine the exact size $|V_j|$, for $j = 0, 1, \dots, L(N)$.

Problem 6.2.3. - Decimation in time. Analyze the FFT algorithmic scheme for $N = 2^d$ and the factorization choice $(N/2, 2)$, along the lines of Section 6.3.

8.7 Off-line Permutation Routing

Problem 7.2.1. - Special N for permutation networks. If a permutation network of size N can realize each of the $N!$ permutations exactly by one configuration, what is the value of N?

Problem 7.3.1. - Inverse permutations on $BN(N)$. Given a configuration of the switches of $BN(N)$ that realizes a permutation π, obtain a configuration that realizes the inverse permutation π^{-1}.

Problem 7.3.2. - Identity permutation on $BN(N)$. Show that the identity permutation is realized by $BN(N)$, by at least $2^{(N/2)(\log_2 N-1)}$ different configurations.

Problem 7.3.3. - Cyclic shifts on Ω and Ω^{-1}. Show that the networks Ω and Ω^{-1} can route all cyclic shifts. Hints:

- A direct proof, based on the definitions of Ω and cyclic shifts, is possible and quite instructive, but it will take some effort. It is possible to proceed by induction, exploting the observation that $\Omega(N)$ can be viewed as one stage of $N/2$ switches followed by two "parallel" $\Omega(N/2)$.
- For Ω, there is a rather simple proof, using results from Section 1.3.
- Given the result for Ω, to deal with Ω^{-1}, remeber that a cyclic shift is an invertible permutation and its inverse is a cyclic shift.

Problem 7.3.4. - Path uniqueness in Ω.

1. Prove that, for any $i, j \in \{0, 1, \ldots, N-1\}$, there exsits a unique path in $\Omega(N)$ from input terminal i to output terminal j.
2. Assuming the property in the previous point, outline an efficient algorithm to decide whether a given permutation π can be realized by $\Omega(N)$. If you find such an algorithm, it will probably run in time $T(N) = O(N \log_2 N)$.

Problem 7.3.4. - Removing the middle stage from $BN(N)$. Let $BN'(N)$ the network obtained from $BN(N)$, by removing all the switches of the middle stage ($stage(d-1)$). Prove the following properties:

1. $BN'(N)$ consists of two disconnected copies of $\Omega(N/2)\Omega^{-1}(N/2)$.
2. $BN'(N)$ can realize $((N/2)!)^2$ permutations.

Problem 7.4.1. - Pruned Beneš̃ **Network.** Since the configuration algorithm allows to set one switch of the first stage arbitrarily, we can omit that switch from the network (which is equivalent to always setting that switch straight).

1. Exploiting this observation recursively, show that $N/2 - 1$ switches can be removed from $BN(N)$ without jeopardizing its permutation capability.

2. The resulting network, call it the Pruned BN, has a number of switches $S_{PBN}(N) = N \log_2 N - N + 1$. Compare this number with the lower bound 57.

3. Argue that, in $BN(N)$, each permutation can be realized by at least $2^{N/2-1}$ distinct configurations.

Problem 7.4.2. - Randomized Beneš **Network.** Assume that $RBN(N)$ has been obtained from $BN(N)$, by replacing the $N/2$ switches in $stage(0)$ with $N/2$ switches of the form $sw_0(i, N/2+\sigma(i))$, where σ is a permutation of $(0, 1, \ldots, N/2-1)$. Prove that $RBN(N)$ is a fully rearrangeable permutation network. (Hint: Reflect on the configuration algorithm for BN.)