Models of Parallel Algoritms and Simple Parallel Algorithms

May 11th, 2009

Content

- Models
  - data parallel,
  - task graph,
  - work pool,
  - master-slave,
  - pipeline,
  - hybrids;
- Applying data parallel model,
- Building-block computations
- Sorting networks

Models & Data parallel model

- An algorithm model is typically a way of structuring a parallel alg.
  - by selecting a decomposition and mapping technique and
  - by applying the appropriate strategy to minimize interactions
- The data-parallel model is one of the simplest algorithm models.
  - the tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data.
  - This type of parallelism that is a result of identical operations being applied concurrently on different data items is called data parallelism.
  - The work may be done in phases and the data operated upon in different phases may be different.
  - Typically, data-parallel computation phases are interspersed with interactions to synchronize the tasks or to get fresh data to the tasks.
  - Since all tasks perform similar computations, the decomposition of the problem into tasks is usually based on data partitioning
    - because a uniform partitioning of data followed by a static mapping is sufficient to guarantee load balance

Data parallel model

- Data-parallel algorithms can be implemented in both shared-address-space and message-passing paradigms.
  - The partitioned address-space in a message-passing paradigm may allow better control of placement, and thus may offer a better handle on locality.
  - Shared-address space can ease the programming effort, especially if the distribution of data is different in different phases of the algorithm.
- Interaction overheads in the data-parallel model can be minimized
  - by choosing a locality preserving decomposition
  - by overlapping computation and interaction and
  - by using optimized collective interaction routines.
- A key characteristic of data-parallel problems is that for most problems, the degree of data parallelism increases with the size of the problem, making it possible to use more processes to effectively solve larger problems.
- An example of a data-parallel algorithm is dense matrix multiplication.

Task Graph Model

- The task-dependency graph may be either trivial, as in the case of matrix multiplication, or nontrivial.
  - In certain parallel algorithms, the task-dependency graph is explicitly used in mapping
- In the task graph model, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs.
- Employed to solve probls in which the amount of data associated with the tasks is large relative to the amount of computation associated with them
- Work is more easily shared in paradigms with globally addressable space, but mechanisms are available to share work in disjoint address space.
- Typical interaction-reducing techniques applicable to this model include
  - reducing the volume and frequency of interaction by promoting locality while mapping the tasks based on the interaction pattern of tasks, and
  - using asynchronous interaction methods to overlap the interaction with computation.
- Examples of algorithms based on the task graph model include
  - parallel quicksort,
  - sparse matrix factorization,
  - parallel algorithms derived via divide-and-conquer decomposition.
- This type of parallelism that is naturally expressed by independent tasks in a task-dependency graph is called task parallelism.

Work Pool Model (Task Pool Model)

- Characterized by a dynamic mapping of tasks onto processes for load balancing in which any task may potentially be performed by any process.
- There is no desired premapping of tasks onto processes.
- The mapping may be centralized or decentralized.
- Pointers to the tasks may be stored in a physically shared list, priority queue, hash table, or tree, or they could be stored in a physically distributed data structure.
- Work may be statically available in the beginning, or could be dynamically generated;
  - i.e., the processes may generate work and add it to the global (possibly distributed) work pool.
- If the work is generated dynamically and a decentralized mapping is used,
  - then a termination detection alg. would be required so that all processes can actually detect the completion of the entire program and stop looking for more work.
- In the message-passing paradigm, the work pool model is typically used when the amount of data associated with tasks is relatively small compared to the computation associated with the tasks.
  - tasks can be readily moved around without causing too much data interaction overhead.
- Granularity of the tasks can be adjusted to attain the desired level of tradeoff between load-imbalance & overhead of accessing the work pool for adding and extracting tasks.
- Parallelization of loops by chunk scheduling or related methods is an example of the use of the work pool model with centralized mapping when the tasks are statically available.
- Ex. work pool model where the tasks are generated dynamically:
  - Parallel tree search where the work is represented by a centralized or distributed data structure

Master-Slave Model (Manager-Worker)

- One or more master processes generate work and allocate it to worker processes.
  - the tasks may be allocated a priori if the manager can estimate the size of the tasks or
  - if a random mapping can do an adequate job of load balancing.
  - In another scenario, workers are assigned smaller pieces of work at different times.
    - preferred if it is time consuming for the master to generate work and hence it is not desirable to make all workers wait until the master has generated all work pieces.
- In some cases, work may need to be performed in phases, and work in each phase must finish before work in the next phases can be generated.
  - In this case, the manager may cause all workers to synchronize after each phase.
- Usually: no desired premapping of work to processes; any worker can do any job assigned.
- The model can be generalized to hierarchical or multi-level manager-worker model
  - the top-level manager feeds large chunks of tasks to second-level managers, who further subdivide the tasks among their own workers and may perform part of the work themselves.
- This model is generally equally suitable to shared-address-space or message-passing paradigms since the interaction is naturally two-way;
  - the manager knows that it needs to give out work and
  - workers know that they need to get work from the manager.
- Care should be taken
  - to ensure that the master does not become a bottleneck (may happen if tasks are too small/fast)
  - granularity of tasks: cost of doing work >> cost of transferring work & cost of synchronization.
  - asynchronous interaction may help
    - overlap interaction and the computation associated with work generation by the master.
    - It may also reduce waiting times if the nature of requests from workers is nondeterministic.

Pipeline or Producer-Consumer Model

- A stream of data is passed on through a succession of processes, each of which perform some task on it.
- This simultaneous execution of diff.progrs on a data stream is called stream parallelism.
- With the exception of the process initiating the pipeline, the arrival of new data triggers the execution of a new task by a process in the pipeline.
- The processes could form such pipelines in the shape of linear or multidimensional arrays, trees, or general graphs with or without cycles.
- A pipeline is a chain of producers and consumers.
  - Each process in the pipeline can be viewed as
    - a consumer of a sequence of data items for the process preceding it in the pipeline and
    - as a producer of data for the process following it in the pipeline.
- The pipeline does not need to be a linear chain; it can be a directed graph.
- The pipeline model usually involves a static mapping of tasks onto processes.
- Load balancing is a function of task granularity.
- The larger the granularity, the longer it takes to fill up the pipeline,
  - for the trigger produced by the first process in the chain to propagate to the last process, thereby keeping some of the processes waiting.
- Too fine a granularity may increase interaction overheads because processes will need to interact to receive fresh data after smaller pieces of computation.
- The most common interaction reduction technique applicable to this model is overlapping interaction with computation.
- An example of a two-dimensional pipeline is the parallel LU factorization algorithm.

Hybrid models

- In some cases, more than one model may be applicable to the problem at hand, resulting in a hybrid algorithm model.
- A hybrid model may be composed
  - either of multiple models applied hierarchically
  - or multiple models applied sequentially to different phases of a parallel algorithm.
- In some cases, an algorithm formulation may have characteristics of more than one algorithm model.
  - For instance, data may flow in a pipelined manner in a pattern guided by a task-dependency graph.
  - In another scenario, the major computation may be described by a task dependency graph, but each node of the graph may represent a supertask comprising multiple subtasks that may be suitable for data-parallel or pipelined parallelism.
- Parallel quicksort is one of the applications for which a hybrid model is ideally suited.

Applying Data Parallel Model - 1st example

- Consider the problem of constructing the list of all prime numbers in the interval [1, *n*] for a given integer $n > 0$.
- A simple algorithm that can be used for this computation is the **sieve of Eratosthenes.**
  - Start with the list of numbers 1, 2, 3, 4, . . . , *n* represented as a "mark" bit-vector initialized to 1000 . . . 00.
  - In each step, the next unmarked number *m* (associated with a 0 in element *m* of the mark bit-vector) is a prime.
  - Find this element *m* and mark all multiples of *m* beginning with m^2.
  - When $m^2 > n$, the computation stops and all unmarked elements are prime numbers.
- The list of numbers and the current prime are stored in a shared memory that is accessible to all processors.
- An idle processor simply refers to the shared memory, updates the current prime, and uses its private index to step through the list and mark the multiples of that prime.
- Division of work is thus self-regulated.
- We next examine a data-parallel approach in which the bit-vector representing the *n* integers is divided into *p* equal-length segments, with each segment stored in the private memory of one processor.
  - All of the primes whose multiples have to be marked reside in Processor 1, which acts as a coordinator:
  - It finds the next prime and broadcasts it to all other processors, which then proceed to mark the numbers in their sublists

Applying Data Parallel Model – 1st example

- The overall solution time now consists of two components:
  - the time spent on transmitting the selected primes to all processors (communication time) and
  - the time spent by individual processors marking their sublists (computation time).
- Typically, communication time grows with the number of processors, though not necessarily in a linear fashion.
- Because of the above-mentioned communication overhead, adding more processors beyond a certain optimal number does not lead to any improvement in the total solution time or in attainable speed-up.
- Finally, consider the data-parallel solution, but with data I/O time also included in the total solution time.
  - Assuming for simplicity that the I/O time is constant and ignoring communication time, the I/O time will constitute a larger fraction of the overall solution time as the computation part is speeded up by adding more and more processors.
  - If I/O takes 100 seconds, say, then there is little difference between doing the computation part in 1 second or in 0.01 second.
- We will later see that such “sequential” or ““unparallelizable” portions of computations severely limit the speed-up that can be achieved with parallel processing.

Applying Data Parallel Model – 2nd example

- problem of distributing a vector **x** with *n* elements *x*0, *x*1, . . . , *xn*−1 on *p* processors.
- Cyclic distribution

  0 : *x*0 *x*5 *x*10 *x*15 *x*20
  1 : *x*1 *x*6 *x*11 *x*16 *x*21
  2 : *x*2 *x*7 *x*12 *x*17
  3 : *x*3 *x*8 *x*13 *x*18
  4 : *x*4 *x*9 *x*14 *x*19
- Block distribution

  0 : *x*0 *x*1 *x*2 *x*3 *x*4
  1 : *x*5 *x*6 *x*7 *x*8 *x*9
  2 : *x*10 *x*11 *x*12 *x*13 *x*14
  3 : *x*15 *x*16 *x*17 *x*18 *x*19
  4 : *x*20 *x*21
- Block-cyclic distribution
  - It improves upon the badly balanced block distribution,
  - Restores some locality of reference memory access of the cyclic distribution.

  0 : *x*0 *x*1 *x*10 *x*11 *x*20 *x*21
  1 : *x*2 *x*3 *x*12 *x*13
  2 : *x*4 *x*5 *x*14 *x*15
  3 : *x*6 *x*7 *x*16 *x*17
  4 : *x*8 *x*9 *x*18 *x*19

Applying Data Parallel Model – 3rd example

- Data mining differs from standard database queries in that its goal is to identify implicit trends and segmentations, rather than simply look up the data requested by a direct, explicit query.
  - For example, finding all customers who have bought cat food in the last week is not data mining
  - however, segmenting customers according to relationships in their age group, monthly income, preferences in pet food, cars, is.
- A particular, quite directed type of data mining is **mining for associations**.
  - the goal is
    1. to discover relationships (associations) among the information related to different customers and their transactions, and
    2. to generate rules for the inference of customer behavior.
  - For example, the database may store for every transaction the list of items purchased in that transaction.
    - The goal of the mining may be to determine associations between sets of commonly purchased items that tend to be purchased together;
    - for example, the conditional probability that a certain set of items is found in a transaction given that a different set of items is found in that transaction.

Applying Data Parallel Model – 3rd example

- Given a database in which the records correspond to customer purchase transactions
- Each transaction has a transaction id & a set of attributes or items, e.g. the items purchased.
- The first goal in mining for associations is to examine the database and determine which sets of *k* items, say, are found to occur together in more than a given threshold fraction of the transactions.
  - A set of items (of any size) that occur together in a transaction is called an *itemset*,
  - An itemset that is found in more than that threshold percentage of transactions is called a *large itemset*.
- Once the large itemsets of size *k* are found with their frequencies of occurrence in the database of transactions, determining the association rules among them is quite easy.
- The problem we consider therefore focuses on discovering the large itemsets of size *k* and their frequencies.
- A simple way to solve the problem is
  - to first determine the large itemsets of size one.
  - From these, a set of candidate itemsets of size two items can be constructed and their frequency of occurrence in the transaction database counted
    - using the basic insight that if an itemset is large then all its subsets must also be large
  - This results in a list of large itemsets of size two.
  - The process is repeated until we obtain the large itemsets of size *k*.
- There is *concurrency* in examining large itemsets of size *k-1* to determine candidate itemsets of size *k*, and in counting the number of transactions in the database that contain each of the candidate itemsets.

Fundamental Building-block Computations

- five fundamental building-block computations:

1. Semigroup (reduction, fan-in) computation
2. Parallel prefix computation
3. Packet routing
4. Broadcasting, and its more general version, multicasting
5. Sorting records in ascending/descending order of their keys

Semigroup Computation

- Let ọ be an associative binary operator;
  - i.e., (*x* o *y*) ọ *z*= *x* ọ (*y* ọ *z*) for all *x, y, z* iṇ *S.*
- A semigroup is simply a pair (*S,* o), where *S* is a set of elements on whicḥ is defined.
- *Semigroup* (also known as *reduction* or *fan-in*) *computation* is defined as:
- Given a list of *n* values x_0, x_1, . . . , x_{n-1}, compute x_0 ọx_1 ọ. . . ọ x_{n-1} .
- Common examples for the operator ọ include +, *, ∩, max, min.
- The operator ọ may or may not be commutative,
  - i.e., it may or may not satisfy *x* ọ *y* = *y* ọ *x*
  - all of the above examples are, but the carry computation, e.g., is not
  - while the parallel algorithm can compute chunks of the expression using any partitioning scheme, the chunks must eventually be combined in left-to-right order.

Parallel Prefix Computation & Packet Routing

- **Parallel Prefix Computation.**
- Same assumptions as in the preceding paragraph,
- A parallel prefix computation is defined as simultaneously evaluating all of the prefixes of the expression $x_0 \circ x_1 \circ \ldots \circ x_{n-1}$
  - i.e., x_0, $x_0 \circ x_1$, $x_0 \circ x_1 \circ x_2$, . . . , $x_0 \circ x_1 \circ \ldots \circ x_{n-1}$.
  - The *i*th prefix expression is $s_i = x_0 \circ x_1 \circ \ldots \circ x_i$.
- **Packet Routing.**
  - A packet of information resides at Processor *i* and must be sent to Processor *j.*
  - The problem is to route the packet through intermediate processors, if needed, such that it gets to the destination as quickly as possible.
  - The problem becomes more challenging when multiple packets reside at different processors, each with its own destination.
  - In this case, the packet routes may interfere with one another as they go through common intermediate processors.
  - When each processor has at most one packet to send and one packet to receive, the packet routing problem is called *one-to-one communication* or *1–1 routing.*

Broadcasting & Sorting

- **Broadcasting.**
  - Given a value *a* known at a certain processor *i,* disseminate it to all *p* processors as quickly as possible, so that at the end, every processor has access to, or "knows," the value.
  - This is sometimes referred to as *one-to-all communication.*
  - The more general case of this operation, i.e., one-to-many communication, is known as *multicasting.*
  - From a programming viewpoint, we make the assignments *xj*: = *a* for 1 ≤*j* ≤*p* (broadcasting) or for *j* in *G* (multicasting), where *G* is the multicast group and *xj* is a local variable in processor *j.*
- **Sorting.**
  - Rather than sorting a set of records, each with a key and data elements, we focus on sorting a set of keys for simplicity.
  - Our sorting problem is thus defined as: Given a list of *n* keys *x*0, *x*1, . . . , *xn*–1, and a total order ≤on key values, rearrange the *n* keys as *xi* non-descending order.
  - We consider only sorting the keys in converted, in a straightforward manner, to one for sorting the keys in nonascending order or for sorting records.

Algorithms for linear array – semigroup computat.

- Linear interconnection
- A special case of semigroup computation, namely, that of maximum finding.
  - Each of the *p* processors holds a value initially and our goal is for every processor to know the largest of these values.
  - A local variable, max-thus-far, can be initialized to the processor's own data value.
  - In each step, a processor sends its max-thus-far value to its two neighbors.
  - Each processor, on receiving values from its left and right neighbors, sets its max-thus-far value to the largest of the three values, i.e., max(left, own, right).
  - In the worst case, $p - 1$ communication steps (each involving sending a processor's value to both neighbors), and the same no three-way comparison steps, are needed.
  - This is the best one can hope for, given that the diameter of a *p* processor linear array is $D = p - 1$ (diameter-based lower bound).
- For a general semigroup computation,
  - the processor at the left end of the array (the one with no left neighbor) becomes active and sends its data value to the right (initially, all processors are dormant or inactive).
  - On receiving a value from its left neighbor, a processor becomes active, applies the semigroup operation o to the value received from the left and its own data value, sends the result to the right, and becomes inactive again.
  - The wave of activity propagates to the right until the rightmost processor obtains the desired res.
  - The computation result is then propagated leftward to all processors. In all, $2p - 2$ communication steps are needed.

Algorithms for linear array – parallel prefix

- Assume that the ith prefix result obtained at the ith processor, $0 \leq i \leq p - 1$.
- The general semigroup algorithm described in the preceding paragraph
  - performs a semigroup computation first and
  - then does a broadcast of the final value to all processors.
  - Thus, we already have an algorithm for parallel prefix computation that takes $p - 1$ communication/combining steps.
- A variant of the parallel prefix computation, in which Processor i ends up with the prefix result up to the $(i - 1)$th value, is sometimes useful.
  - This *diminished prefix computation* can be performed just as easily if each processor holds onto the value received from the left rather than the one it sends to the right
- Thus far, we have assumed that each processor holds a single data item.
  - Extension of the semigroup and parallel prefix algorithms to the case where each processor initially holds several data items is straightforward.
- In a parallel prefix sum computation with each processor initially holding two data items, the algorithm consists of
  - each processor doing a prefix computation on its own data set of size n/p (this takes $n/p - 1$ combining steps),
  - then doing a diminished parallel prefix computation on the linear array as above ($p - 1$ communication/combining steps), and
  - finally combining the local prefix result from this last computation with the locally computed prefixes (n/p combining steps).
- In all, $2n/p + p - 2$ combining steps and $p - 1$ communication steps are required.

Algorithms for linear array – packet routing & broadcasting

- ***Packet Routing.***
  - To send a packet of information from Processor *i* to Processor *j* on a linear array, we simply attach a *routing tag* with the value *j* – *i* to it.
  - The sign of a routing tag determines the direction in which it should move (+ = right, – = left) while its magnitude indicates the action to be performed (0 = remove the packet, nonzero = forward the packet).
  - With each forwarding, the magnitude of the routing tag is -- by 1. Multiple packets originating at different processors can flow rightward and leftward in lockstep, without ever interfering with each other.
- ***Broadcasting.***
  - If Processor *i* wants to broadcast a value *a* to all processors, it sends an rbcast(*a*) (read r-broadcast) message to its right neighbor and an lbcast(*a*) message to its left neighbor.
  - Any processor receiving an rbcast(*a*) message, simply copies the value *a* and forwards the message to its right neighbor (if any).
  - Similarly, receiving an lbcast(*a*) message causes *a* to be copied locally and the message forwarded to the left neighbor.
  - The worst-case no. communication steps for broadcasting is *p* – 1.

Algorithms for linear array – sorting

- If the key values are already in place, one per processor, then an algorithm known as *odd–even transposition* can be used for sorting.
  - A total of p steps are required.
  - In an odd-no. step, odd-no. processors compare values with their even-no. right neighbors.
  - The two processors exchange their values if they are out of order.
  - Similarly, in an even-numbered step, even-numbered processors compare–exchange values with their right neighbors.
  - Worst case: largest key value in P0 and must move all the way to the other end of the array.
    - This needs $p - 1$ right moves.
  - One step must be added because no movement occurs in the first step.
  - Of course one could use even–odd transposition, but this will not affect the worst-case time complexity of the algorithm.
- Note that the odd–even transposition algorithm uses p processors to sort p keys in p compare–exchange steps.
- In most practical situations, the number n of keys to be sorted (the *problem size*) is greater than the number p of processors (the *machine size*).
- The odd–even transposition sort algorithm with n/p keys per processor is as follows.
  - First, each processor sorts its list of size n/p using any efficient sequential sorting algorithm.
  - Next, the odd–even transposition sort is performed as before, except that each compare–exchange step is replaced by a merge–split step in which
    - the two communicating processors merge their sublists of size n/p into a single sorted list of size $2n/p$ and
    - then split the list down the middle, one processor keeping the smaller half and the other, the larger half.

Algorithms for binary trees

- ***Semigroup Computation.***
  - Each inner node receives two values from its children (if each of them has already computed a value or is a leaf node),
  - applies the operator to them, and passes the result upward to its parent.
  - After ceil(log2 *p*) steps, the root processor will have the computation result.
  - All procs are then notified of the result through a broadcasting operation from the root.
  - Total time: 2ceil(log2*p*) steps.
- ***Parallel Prefix Computation.***
  - Can be done optimally in 2ceil(log 2 *p)* steps
  - Consists of an upward propagation phase followed by downward data movement.
  - The upward propagation phase is identical to the upward movement of data in semigroup computation.
  - At the end of this phase, each node will have the semigroup comp.res.for its subtree.
  - The downward phase is as follows.
    - Each processor remembers the value it received from its left child.
    - On receiving a value from the parent, a node passes the value received from above to its left child & the combination of this value and the one that came from the left child to its right child.
  - The root is viewed as receiving the identity element from above and thus initiates the downward phase by sending the identity element to the left & the value received from its left child to the right.
  - At the end of the downward phase, the leaf processors compute their respective res.

Algorithms for binary trees - sorting

- We can use an algorithm similar to bubblesort that
  - allows the smaller elements in the leaves to “bubble up” to the root processor first,
  - thus allowing the root to “see” all of the data elements in nondescending order.
  - the root then sends the elements to leaf nodes in the proper order.
- Before describing the part of the algorithm dealing with the upward bubbling of data, let us deal with the simpler downward movement.
  - This downward movement is easily coordinated if each node knows the number of leaf nodes in its left subtree.
  - If the rank order of the element received from above (kept in a local counter) does not exceed the number of leaf nodes to the left, then the data item is sent to the left.
  - Otherwise, it is sent to the right.
- Note: implicitly assumes that data are to be sorted from left to right in the leaves.
- The upward movement of data in the above sorting algorithm can be accomplished:
  - Initially, each leaf has a single data item and all other nodes are empty.
  - Each inner node has storage space for 2 values, migrating upward from its left & right subtrees.

```
if you have 2 items
then do nothing
else if you have 1 item that came from the left (right)
        then get the smaller item from the right (left) child
        else get the smaller item from each child
```

- The above sorting algorithm takes linear time in the number of elements to be sorted.
- A more efficient sorting algorithm can be developed?
  - The answer, unfortunately, is that we cannot do fundamentally better than the above.

Algorithms for 2d meshes

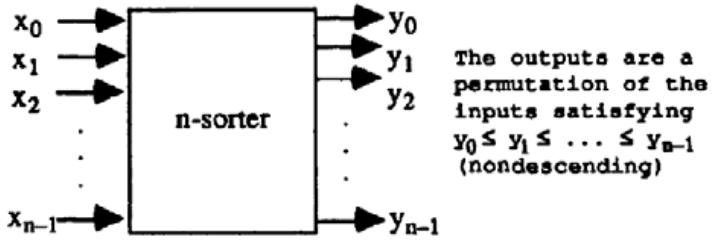
- ***Semigroup Computation.***
  - Do the semigroup computation in each row and then in each column.
  - For example, in finding the maximum of a set of *p* values, stored one per processor,
    - the row maximums are computed first and made available to every processor in the row.
    - Then column maximums are identified.
  - The same process can be used for computing the sum of *p* numbers.
  - Note that for a general semigroup computation with a noncommutative operation, the *p* numbers must be stored in row-major order for this algorithm to work correctly.
- ***Parallel Prefix Computation.***
  - Can be done in three phases, assuming that the processors (and their stored values) are indexed in row-major order:
    1. do a parallel prefix computation on each row,
    2. do a diminished parallel prefix computation in the rightmost column, and
    3. broadcast the results in the rightmost column to all of the elements in the respective rows and combine with the initially computed row prefix value.
  - For example, in doing prefix sums,
    - first-row prefix sums are computed from left to right.
    - the processors in the rightmost column hold the row sums.
    - A diminished prefix computation in this last column yields the sum of all of the preceding rows in each processor.
    - Combining the sum of all of the preceding rows with the row prefix sums yields the overall prefix sums.

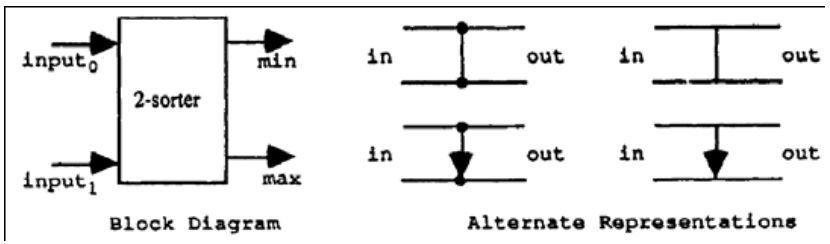
Algorithms for 2d meshes

- ***Broadcasting.***
  - In two phases:
  1. broadcast the packet to every processor in the source node's row and
  2. broadcast in all columns.
  - If multiple values are to be broadcast by a processor, then the required data movements can be pipelined, such that each additional broadcast requires only one additional step.
- ***Sorting.***
  - the simple version of a sorting algorithm known as *shearsort.*
  - The algorithm consists of ceil(log 2*r*)+ 1 phases in a 2D mesh with *r* rows.
  - In each phase, except for the last one, all rows are independently sorted in a snakelike order:
    - even-numbered rows 0, 2, … from left to right,
    - odd-numbered rows 1, 3, . . . from right to left.
  - Then, all columns are independently sorted from top to bottom.
  - In the final phase, rows are independently sorted from left to right.
  - The shearsort algorithm needs compare exchange steps for sorting in row-major order.

Sorting Network

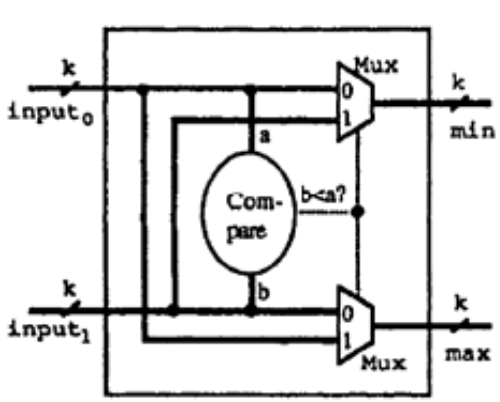
- Circuit-level designs for parallel processing are necessarily problem-specific or special-purpose.
- A sorting network is a circuit that receives n inputs, and permutes them to produce n outputs, such that the outputs satisfy $y_0 \leq y_1 \leq y_2 \leq \ldots y_{n-1}$.
- Refer to such an n-input n-output sorting network as an *n-sorter*
- Just as many sorting algorithms are based on comparing and exchanging pairs of keys, we can build an n-sorter out of 2-sorter building blocks.
- A 2-sorter compares its two inputs (call them *input*0 and *input*1) and orders them at the output, by switching their order if needed, putting the smaller value, min(*input*0, *input*1), before the larger value, max(*input*0, *input*1).

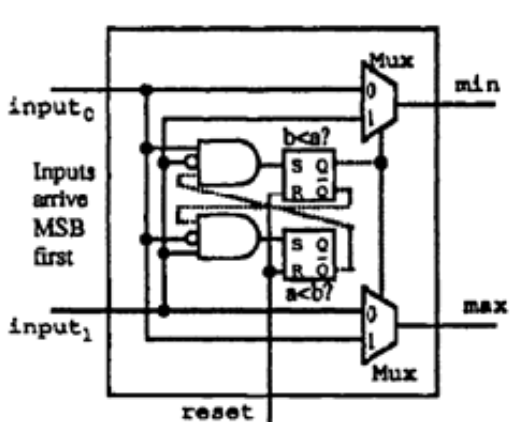




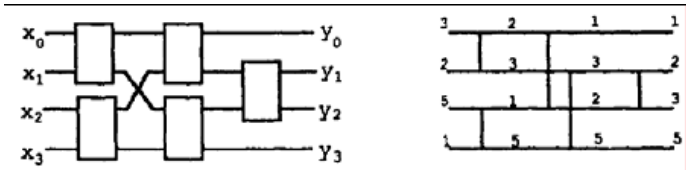
Hardware realization of a 2-sorter

- If we view the inputs as unsigned integers that are supplied to the 2-sorter in bit-parallel form, then the 2-sorter can be implemented using a comparator and two 2-to-1 multiplexers, as shown in left fig.
- When the keys are long, or when we need to implement a sorting network with many inputs on a single VLSI chip, bit-parallel input becomes impractical in view of pin limitations.
- Right fig. also depicts a bit-serial hardware realization of the 2-sorter using two state flip-flops.
  - The flip-flops are reset to 0 at the outset.
  - This state represents the two inputs being equal thus far. The other two states are 01 (the upper input is less) and 10 (the lower input is less).
  - While the 2-sorter is in state 00 or 01, the inputs are passed to the outputs straight through.
  - When the state changes to 10, the inputs are interchanged, with the top input routed to the lower output and vice versa.





Correctness of a n-sorter



- Fig depicts a 4-sorter built of 2-sorter building blocks: block diagram and the schematic representation.
- The schematic representation of the 4-sorter shows the data values carried on all lines when the input sequence 3, 2, 5, 1 is applied.
- How do we verify that the circuit is fact a valid 4-sorter?
- The *zero–one principle* allows us to do this with much less work.
  - An *n* -sorter is valid if it correctly sorts all 0/1 sequences of length *n.*
- Using the zero–one principle, the correctness of the 4-sorter can be verified by testing it for the 16 possible 0/1 sequences of length 4.
  - The network clearly sorts 0000 and 1111.
  - It sorts all sequences with a single 0 because the 0 “bubbles up” to the top line.
  - Similarly, a single 1 would “sink down” to the bottom line.
  - The remaining part of the proof deals with the sequences 0011, 0101, 0110, 1001, 1010, 1100, all of which lead to the correct output 0011.

Best n-sorter?

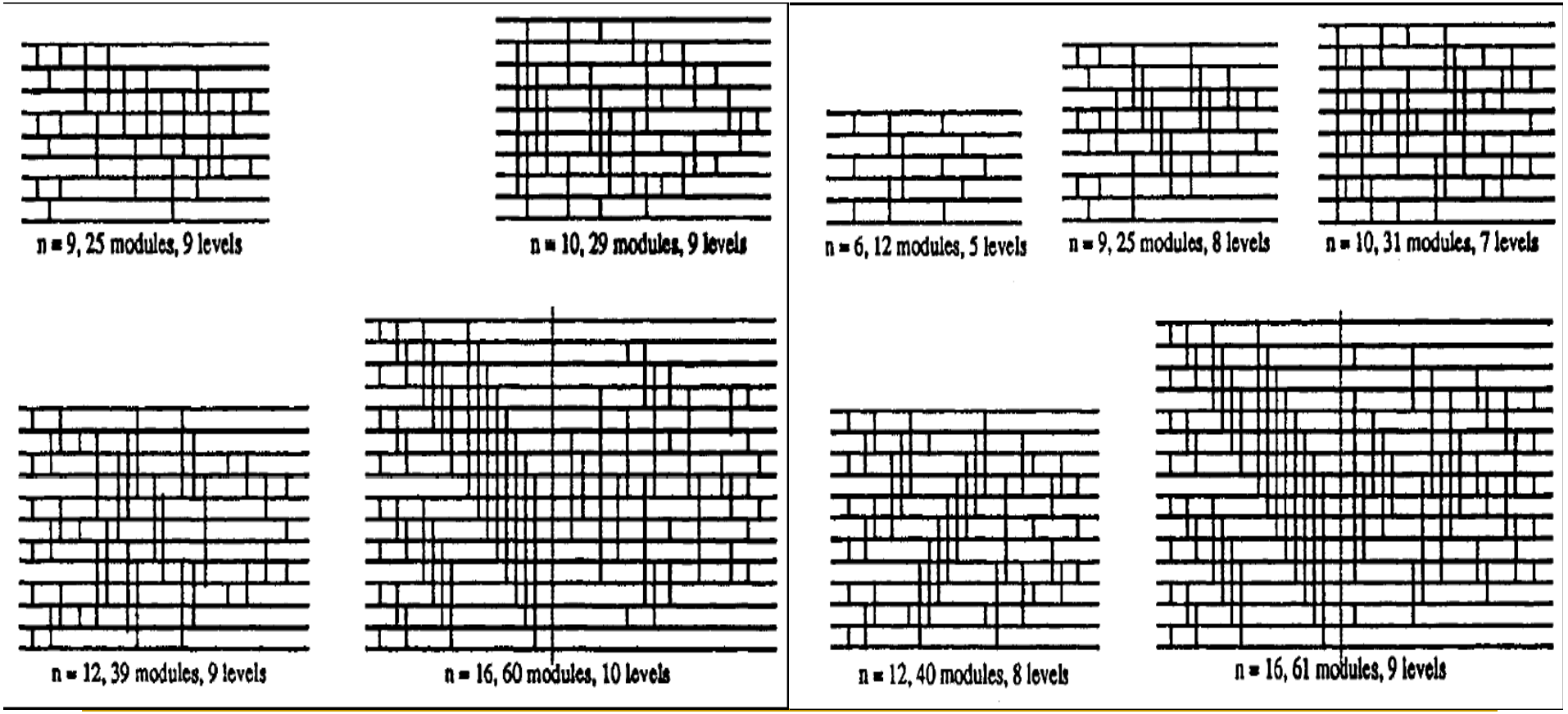
- what we mean by “the best *n*-sorter”?
- Cost: the total number of 2-sorter blocks used in the design
- Delay: the number of 2-sorters on the critical path from input to output

1) Minimizing cost x delay would be appropriate.

- If we can redesign a sorting network so that it is 10% faster but only 5% more complex, the redesign is deemed to be cost-effective and the resulting circuit is said to be *time-cost-efficient* (or at least more so than the original one.

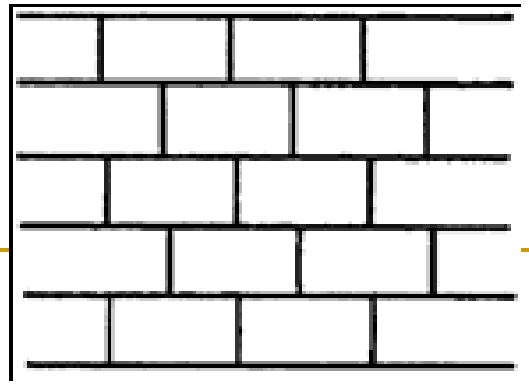
Examples:

- Low cost sorting networks
- Fast sorting networks



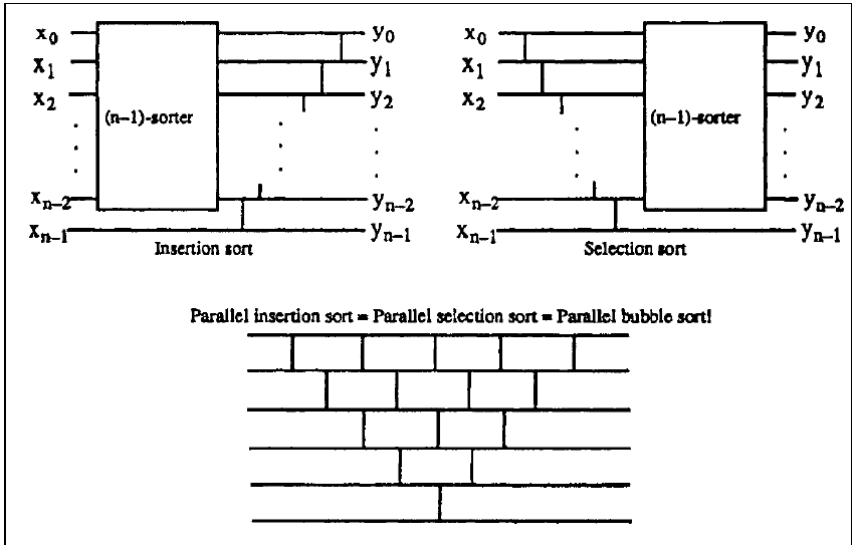
General best n-sorter?

- lowest-cost designs are known only for small *n* and as yet there is no general method for systematically deriving low-cost designs
- The fastest possible designs are also known only for small *n;* time-cost-efficient sorting networks are even harder to come by.
- There are many ways to design sorting networks, leading to different results with respect to the figures of merit.
  - For example, Fig. shows a 6-sorter whose design is based on the odd–even transposition sorting algorithm discussed in connection with sorting on a linear array of processors.
    - This “brick wall” design offers advantages in terms of wiring ease (because wires are short and do not cross over).
    - It is quite inefficient as it uses *n* x floor(*n*/2) modules and has *n* units of delay.
    - Its cost x delay product is $O(n^3)$.



Sorting network by recursive design?

- One way to sort n inputs is to sort the first $n - 1$ inputs, say, and then insert the last input in its proper place.
- Ex:
  - insertion sort (Fig. left)
  - Selection sort (Fig. right) – bubblesort
- Inefficient:
  - cost $O(n^2)$
    (> $O(n \log n)$ by splitting)
  - delay $O(n)$
    (> $O(\log n)$ by slitting)



Sorting network based on insertion or selection sort + recursivity

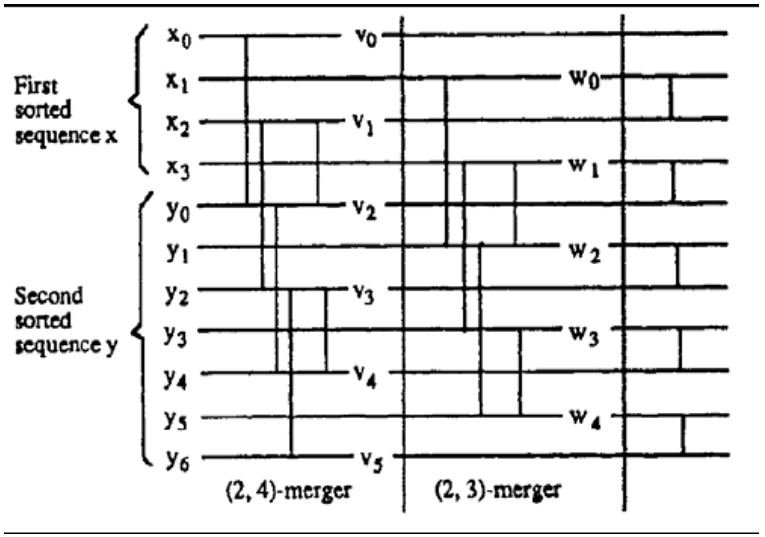
Practical sorting networks: Batcher's netws

- Suboptimal: Cost x delay = $O(n \log^4 n)$
- But Batcher's sorting networks are quite efficient.
- Attempts at designing faster or less complex networks for specific values of n have yielded only marginal improvements over Batcher's construction when n is large.
- Examples:

1. Sorting based on even-odd merging technique
2. Sorting based on bitonic sequences

Odd-even merging technique

- An (*m* ,*m′*)-merger is a circuit that merges two sorted sequences of lengths *m* and *m′* into a single sorted sequence of length *m* + *m′*.
- Let the two sorted sequences be $x_0<=x_1<=\ldots x_{m-1}$, $y_0<=y_1<=\ldots y_{m'-1}$
- The odd–even merge is done by merging the even- and odd indexed elements of the two lists separately :
  - $x_0,x_2,\ldots x_{2ceil(m/2)-2}$ and $y_0,y_2,\ldots y_{2ceil(m'/2)-2}$ are merged to get $v_0,v_1,\ldots v_{ceil(m/2)+ceil(m'/2)-1}$
  - $x_1,x_3,\ldots x_{2ceil(m/2)-1}$ and $y_1,y_3,\ldots y_{2ceil(m'/2)-1}$ are merged to get $w_0,w_1,\ldots w_{ceil(m/2]+ceil(m'/2)-1}$
- If we now compare–exchange the pairs of elements $w_0:v_1$, $w_1:v_2$, $w_2:v_3$,.... the resulting sequence $v_0w_0v_1w_1v_2w_2\ldots$ will be completely sorted.
- Note that v_0, which is known to be the smallest element overall, is excluded from the final compare–exchange operations
- Ex: final (4, 7)-merger in Fig. uses 16 modules and has a delay of 4 units

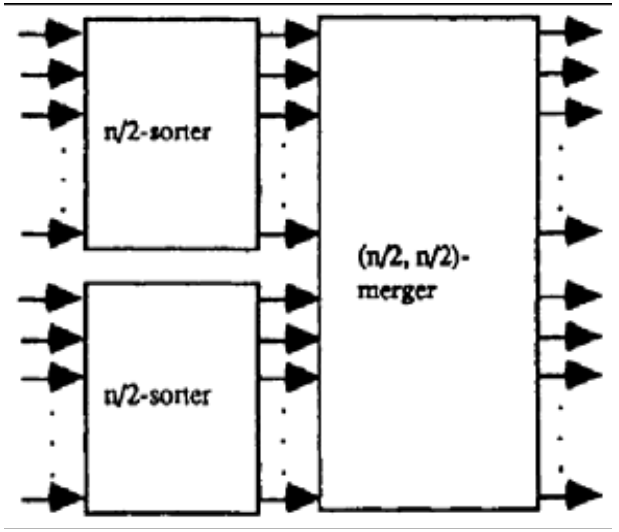


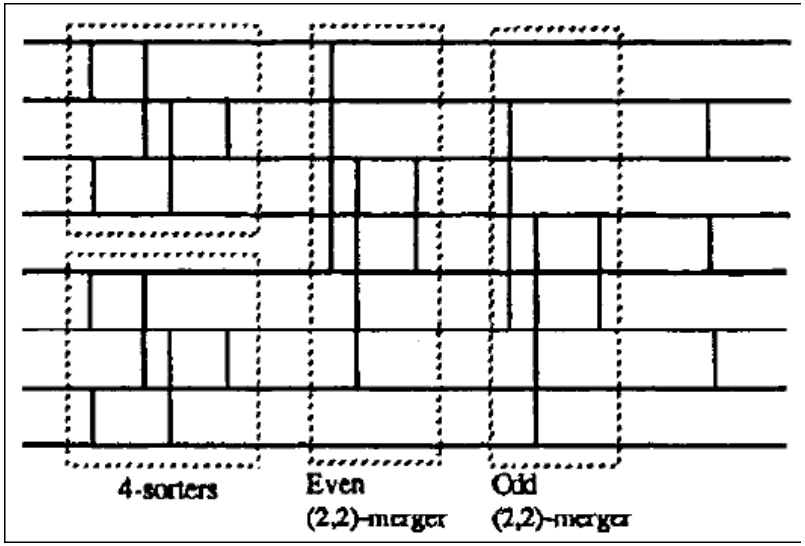
The three circuit segments,
separated by vertical dotted lines,
correspond to a (2, 4)-merger for even-indexed inputs,
a (2, 3)-merger for odd-indexed inputs,
and the final parallel compare–exchange operations.

Each of the smaller mergers can be designed recursively
For example, a (2, 4)- merger consists of two (1, 2)-mergers for even- and odd-indexed inputs, followed by two parallel compare– exchange operations.

Recursive sorter based on odd-even merging

- Armed with an efficient merging circuit, we can design an *n*-sorter recursively from two *n*/2-sorters and an (*n*/2, *n*/2)-merger, as shown in Fig. 1
- The 4-sorter from previous slide is an instance of this design:
  - it consists of two 2-sorters
  - followed by a (2, 2)-merger
  - in turn built from two (1, 1)-mergers and a single compare–exchange step.
- A larger example, corresponding to an 8-sorter: Fig. 2.
  - 4-sorters are used to sort the first and second halves of the inputs separately,
  - The sorted lists then merged by a (4, 4)-merger composed of an even (2, 2)-merger, an odd (2, 2)-merger, and a final stage of three comparators.



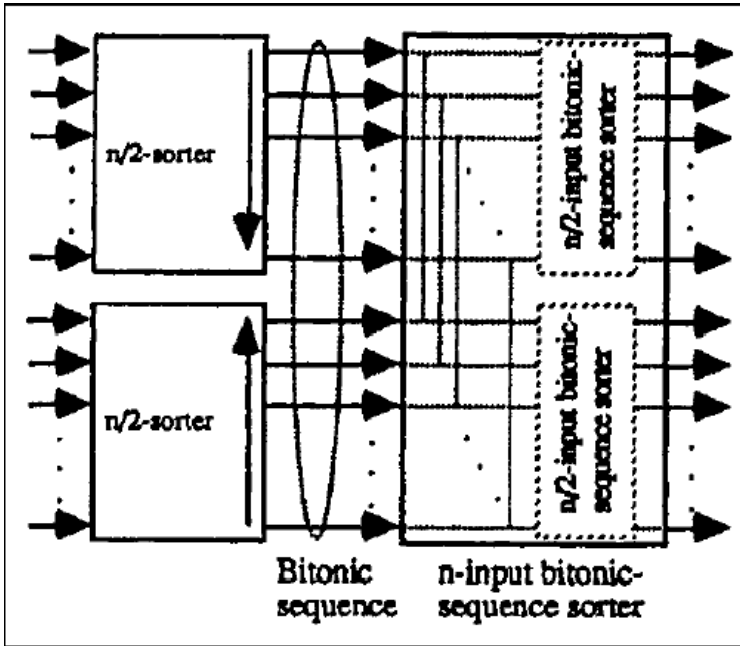


Delay and costs of Batcher's sorting netw.

- Batcher's (*m, m*) even–odd merger, when *m* is a power of 2, is characterized by the following delay and cost recurrences:
  - $C(m)=2C(m/2)+m-1=(m-1)+2(m/2-1)+4(m/4-1)+\ldots = m \log m + 1$
  - $D(m)=D(m/2)+1=\log m+1$
  - Cost x delay = $O(m \log^2 m)$
- Batcher sorting networks based on the even–odd merge technique are characterized by the following delay and cost recurrences:
  - $C(n)=2C(n/2)+(n/2)(\log(n/2))+1=n (\log n)^2/2$
  - $D(n)=D(n/2)+\log(n/2)+1=D(n/2)+\log n=\log n (\log n+1)/2$
  - Cost x delay = $O(n \log^4 n)$

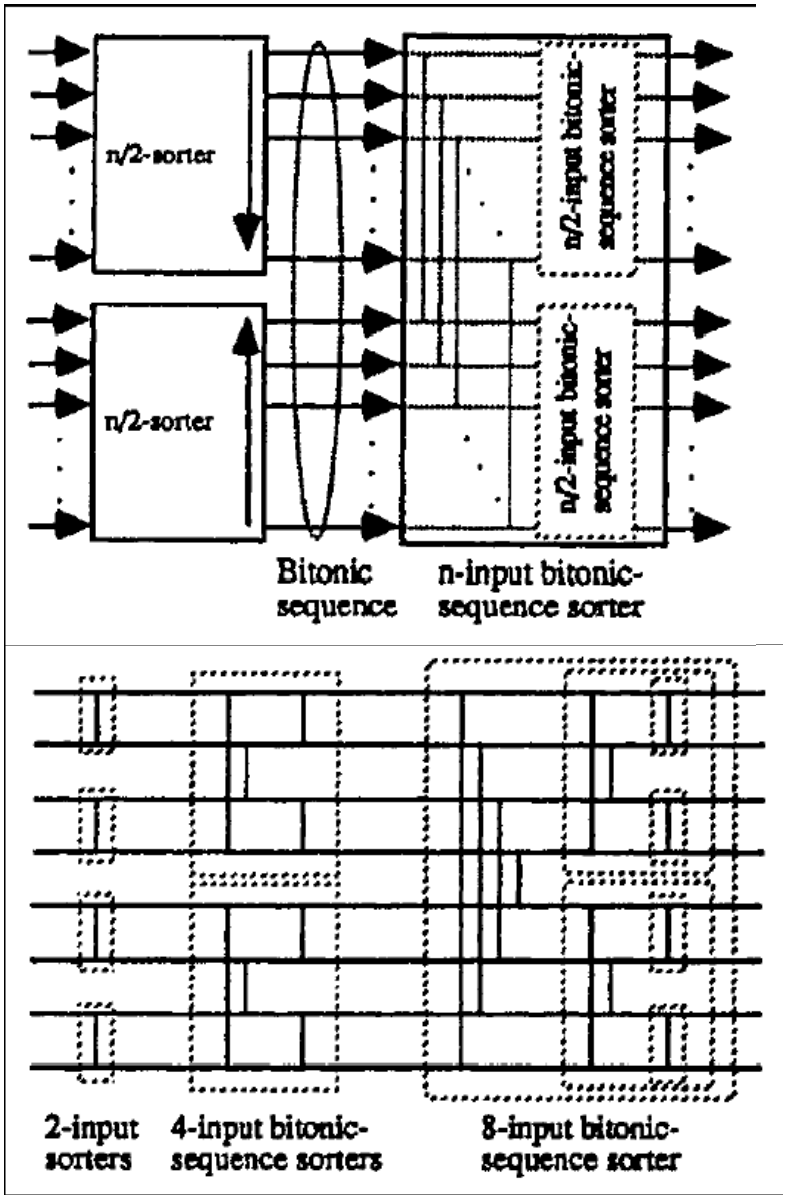
Bitonic sequences

- A bitonic sequence is defined as one that
  - "rises then falls" $x_0 <= x_1 <= \ldots <= x_i >= x_{i+1} >= x_{i+2} >= \ldots >= x_{n-1}$,
  - "falls then rises" $x_0 >= x_1 >= \ldots >= x_i <= x_{i+1} <= x_{i+2} <= \ldots x_{n-1}$)
  - or is obtained from the first two categories through cyclic shifts or rotations.
- Examples include
  - 1 3 3 4 6 6 6 2 2 1 0 0 Rises then falls
  - 8 7 7 6 6 6 5 4 6 8 8 9 Falls then rises
  - 8 9 8 7 7 6 6 6 5 4 6 8 The previous sequence, right-rotated by 2
- Batcher observed that if we sort the first half and second half of a sequence in opposite directions, as indicated by the vertical arrows in Fig, the resulting sequence will be bitonic and can thus be sorted by a special bitonic-sequence sorter.
- It turns out that a bitonic-sequence sorter with *n* inputs has the same delay and cost as an even–odd (*n*/2, *n* /2)-merger.
- Therefore, sorters based on the notion of bitonic sequences (bitonic sorters) have the same delay and cost as those based on even–odd merging.



Bitonic-sequence sorter

- A bitonic-sequence sorter can be designed based on the assertion that:
  - If in a bitonic sequence, we compare-exchange the elements in the first half with those in the second half, as indicated by the dotted comparators in Fig.1,
  - Then
    - each half of the resulting sequence will be a bitonic sequence and
    - each element in the first half will be no larger than any element in the second half.
  - Thus, the two halves can be independently sorted by smaller bitonic sequence sorters to complete the sorting process.
- Note that we can reverse the direction of sorting in the lower *n*-sorter if we suitably adjust the connections of the dotted comparators in Fig. 1.
- A complete eight-input bitonic sorting network is shown in Fig. 2



Periodic balanced sorting networks

- A class of sorting networks that possess the same asymptotic delay and cost as Batcher sorting networks
- An *n*-sorter of this type consists of log *n* identical stages, each of which is a (log *n*)-stage *n*-input bitonic-sequence sorter.
- Thus, the delay and cost of an *n*-sorter of this type are $(\log n)^2$ and $n (\log n)^2/2$.
- Fig. shows an eight-input example
  - larger delay (9 vs. 6) and higher cost (36 vs. 19) compared with a Batcher 8-sorter
  - but offers the following advantages:

1. The structure is regular and modular (easier VLSI layout).
2. Slower, but more economical, implementations are possible by reusing the blocks
3. Using an extra block provides tolerance to some faults (missed exchanges).
4. Using two extra blocks provides tolerance to any single fault (a missed or incorrect exchange).
5. Multiple passes through a faulty network can lead to correct sorting (graceful degradation).
6. Single-block design can be made fault-tolerant by adding an extra stage to the block.