Counting, Sorting and Distributed Coordination

Luis Galárraga
Saarland University

February 6, 2010

Abstract

A first glance at some basic problems in Computer Science like counting and sorting suggest they are inherently sequential; however throughtout the time, several methods and algorithms have been designed in order to solve these problems in a highly parallel way through a systematic flow of coordination tasks among several threads and a set of suitable data structures. This article is a brief overview of the most important techniques for distributed coordination in counting and sorting. For all the methods explained; some hints for implementation, runtime analysis, concurrent performance measures and correctness proofs will be provided.

1 Introduction to Distributed Coordination

Counting and sorting are very basic tasks in software development. In the particular case of sorting, there is a well known theory which unfortunately is formulated assuming a sequential execution. In a multiprocessor environment where multiple threads can be running in different processors, this theory has to be reformulated in order to take advantage of new system architecture trends. But before describing the methods, it is important to define some basic concepts in concurrent execution theory that will help to evaluate the effectiveness and efficiency of the proposed solutions.

1.1 Memory contention

Defined as a scenario in which many threads in different processors try to access the same memory location. This is not a problem when reading, as data can be cached in every processor, but if different processors try to write to a shared (meaning cached by more than one processor) memory location at the same time, cache coherence protocols normally coerce invalidation of the other copies of the data which is clearly inefficient in terms of bus utilization and leads to degradation of program's performance. Concurrent algorithms are designed to avoid memory contention as much as possible.

1.2 Quiescent Consistency

It is said that an object is quiescent in certain moment if it does not have any pending method call, which means there is not any method which has been called but not returned yet. *Quiescent consistence* principle says that the state of any quiescent object should be equivalent to some sequential order of the completed method calls until that moment and that methods separated by a period of quiescence should take time in their real time order. This principle might seem weak, but we will see later that it is enough for some applications. We say a method is total if it is defined for every object state otherwise it is partial. Quiescent consistency is also non-blocking: any call to a total method can always be completed [1].

1.3 Sequential Consistency

This principle states that object method calls should appear to happen in a one-at-a time sequential order, which is the program order. Method calls happening in different threads might be reorder by sequential consistency, but program order in every thread must be fulfilled. As quiescient consistency, sequential consistency is non-blocking [1].

1.4 Linearizability

[1] introduces the concept of a concurrent *history*, as a finite sequence of method invocations and responses belonging to different threads. A history H is sequential if its first element is an invocation and each invocation, except possibly the last, is inmediately followed by a matching response. We define $complete(H)$ as the largest subsequence of H consisting of invocations that have returned. The idea behind *linearizability* is that every *concurrent history* is equivalent, in the following sense, to some sequential history. The basic rule is that if one method call precedes another, then the earlier call must have taken effect before the later call. By constrast, if two method calls overlap, then their order is ambiguous, and we are free to order them in any convenient way. Every linearizable execution is sequentially consistent, but not viceversa.

1.5 Performance and robustness in concurrent objects

Two concepts widely used to evaluate the performance and robustness of a concurrent implementation are latency and throughput. We define *latency* as the time it takes an individual method call to complete whereas *throughput* is the overall or average rate at which a set of method calls complete. For example, real-time applications might care more about latency, and databases might care more about throughput.

2 Distributed Counting

The best example to illustrate the importance of distributed counting are shared counters, where shared means, they can be read and written by several threads concurrently. They have lots of applications like the implementation of concurrent or thread-safe data structures, for instance linked lists, stacks or queues, which depending on the implementation, might require the increment and decrement of some "size" field, everytime an item is added or removed. Other example can be an environment in which some cpu-intensive task has been splitted among several units (in our case threads in different processors, but we propose a general formulation) and we desire a granular real-time feedback for the progress of the whole operation. Units will have to modify concurrently some shared counter as they move along.

2.1 The classical approach

We define a counter as an object that holds an integer value and provides a *getandIncrement* method, which returns the value of the integer and then increments it. When we are talking about threads synchronization and mutual exclusion, the first idea coming into our head are *locks*. A shared counter can be easily implemented with a lock-based strategy, so any thread wanting to modify the counter acquires the lock, modify the counter and then releases the lock. If any other thread has locked the counter, we have to wait until the lock is free. Even though there are multiple implementations for locks, some better than others, all of them suffer from memory contention in some degree and thus their throughput is not optimal. As we stated previously, it can be avoided or reduced with some other techniques.

2.2 Software Combining Trees

Combining trees are designed to solve the problem of many threads requesting to increment a shared counter at "more or less" the same time, generating a high memory contention. The idea behind is simple: some threads become responsible of gathering the increments of other threads, combine them, increment the shared value and then propagate the results. Suppose we have p threads, so we construct a balanced binary tree with k levels, with:

$$k = min\{j|2^j \geq p\} \tag{1}$$

So our tree has 2^{k-1} leaves. Now each thread is assigned a leaf, and at most two threads share a leaf. If one thread wants to modify the counter, it has to traverse the tree from its leave to the root and in each level might combine its value with some other thread. We will illustrate the process with one example.

During thread interactions, the state of nodes change. For the purpose of this illustration and to provide a clue of the implementation we define the following states for the nodes:

- IDLE: Initial state of all nodes, except the root. If a thread has reached the root it means it can finally modify the counter. As soon as a thread arrives to an IDLE node, it changes its status to FIRST.

- FIRST: An earlier thread has recently visited this node, and will return to look for a value to combine. We will call this guy, the *master* thread, whereas a second thread reaching the node is the *slave*. It is worth remarking that a thread can be the master in one node, and slave in another.
- SECOND: A second thread has visited this node and has stored its increment value, waiting for the master to combine.

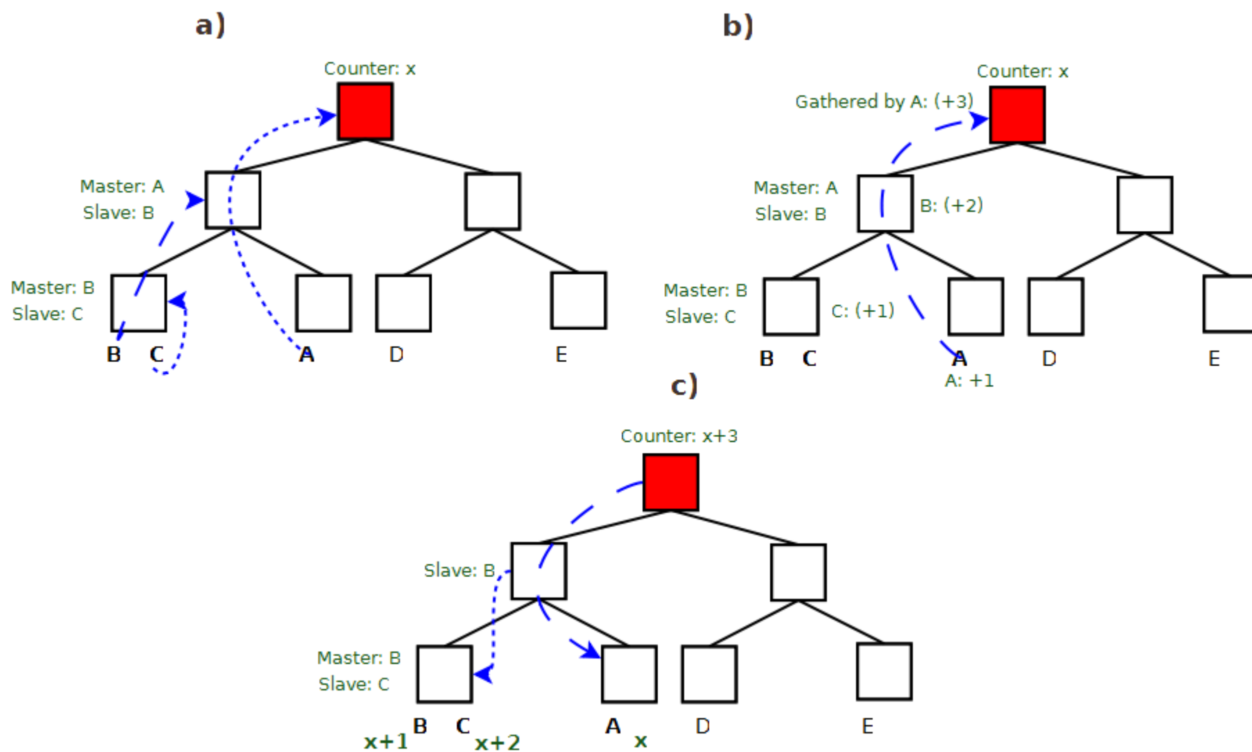


Figure 1: The three stages of the combining trees algorithm.

Suppose threads B and C in figure 1, decide to modify the counter "more or less" at the same time. B manages to start its ascent to the root by the changing the status of its leaf node from IDLE to FIRST, to indicate it will be the active thread there. Concurrently, A starts its ascent to the root changing its leaf node from IDLE to FIRST too. Later, when C attempts to start the ascent and realizes its leaf node is marked as FIRST, it changes the status from FIRST to SECOND, to avoid the master (B in this case) to modify the counter without considering its increment, then locks the node and stops the ascent. Suppose thread A reaches second level before B, so B becomes the slave changing the state from FIRST to SECOND, locking the node and stopping the ascent.

As general rule: as soon as thread is forced to stop (it reached either a node in FIRST state or the root), it starts its combining phase which consists in gathering the values of all the possible slaves that might have appeared in its way. It again traverses the tree from its leaf to its stop node. If a node is in FIRST state, it means there is no slave there. If it is in SECOND state, it might be locked, which means the slave is busy combining with its slaves. In that case we

have to wait. Once the node is free, the master can combine its value with its slave.

Continuing with our example, suppose thread A reaches the root in its first ascent. It inmediately starts the gathering process which implies a second traverse of the tree, depicted in figure 1 part b). When reaches the root again, it is ready to modify the counter. As some other thread may be in the same stage of the process, the node may could be locked. If that is the case, we have to wait and proceed with the modification once the node is available.

The final part of the process is to distribute the changes, which consists in telling our slaves that we are done. This process is also recursive: our slaves will do the same with their slaves until all nodes who participated in the combining process are acknowleged about the modification. This is depicted in figure 1 part c).

Finally some remark. Suppose the counter held some value x before our run. Thread A included B's increment which in the same way included C's increment. Assume increments of one unit, so after A's modification, our counter holds $x+3$. When A reports the value to B, it provides $x+1$ (remember this operation must retrieve in some way the old value). When B receives A's acknowledgment, it proceeds to report to its slave C, providing $x+2$ as value. Thread A will return x, the value it read from the counter. The linearization order of the *getandIncrement* calls by different threads is determined by the order in which they stopped in the first tree traverse.

2.2.1 Analysis of Software Combining Trees

Unlike lock-based approaches, combining trees suffer from latency. In a lock based-approach, each increment takes $O(1)$ time whereas in a combining tree it takes $O(\log(p))$ time; however they have been proved to achieve better throughput because of the decrease in memory contention. If our p threads start a request at more or less the same time, the operation ends in only two accesses to the shared resource. The main disadvantage of combining trees, is their sensitivity to changes in the concurrency rate. There may be cases in which a thread fails in combining inmediately. Using our previous example, suppose thread C decides to start its first traverse just after B's combining phase started (the second traverse). In that case, C will not be able to combine and will have to wait until its leaf node returns to state IDLE. That scenario is not desired at all and if concurrency (defined as the rate of thread requests) is pretty low, it will be frequent and lead to a degradation of performance.

Finally we propose a brief analysis of some possible variants of the algorithm. Suppose we use an n-ary tree instead. Can this modification improve the overall performance? A shallow and intuition-based analysis might suggest that yes, but we have to pay a price. The original problem scales bad if the concurrency is low, which means the probability for two threads to start a request within a small time difference is small. If now n threads share a leaf, this probability is smaller and in a low concurrency environment implies a stronger performance penalty. Other enhancements for this technique are oriented to modify the time a thread has to wait for the arrival of some other to combine. Literature suggest fixed waiting time is not robust because high variance in request arrival rate reduces the combining rate. Surprisingly, evidence suggests to increase waiting times when memory contention is high.

2.3 Counting Networks

Combining trees are a robust option for concurrent counting under some conditions. Additionally, they are a linearizable alternative. However, not all applications require linearizable counting. Counter-based Pool implementations require only quiescently consistent counting. All that matters is that counters produce no duplicates and no omissions. Based on this argument, we start the introduction of *Counting Networks*, for which we provide a bottom-up definition and construction.

2.4 Balancers

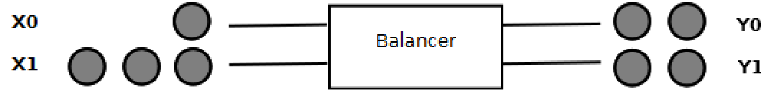


Figure 2: A balancer. Tokens arrive at arbitrary times on arbitrary input lines and are redirected to ensure that when all tokens have exited the balancer, there is at most one more token on the top wire than on the bottom one.

A balancer is a switch with two input and output wires as shown in figure 2. Tokens arrive on the balancer's input wires at arbitrary times and emerge on their output wires at some later time. The balancer is like a toggle. Given a sequence of tokens, it sends one to the top output wire and one to the bottom. If we imagine it as an object, during quiescent periods a balancer must follow this property:

Let x_0 and x_1 the number of tokens that respectively arrive on a balancer's top and bottom input wires, and y_0 and y_1 the number of tokens that emerge on output wires. During quiescent periods:

$$x_0 + x_1 = y_0 + y_1 \tag{2}$$

2.5 Balancing networks

A *balancing network* is constructed by connecting some balancers' output wires to other balancers' input.

We generally say a balancing network has width w if it has w input wires x_0, x_1,...,x_{w-1} and output wires y_0, y_1,..,y_{w-1}. On the other hand, the depth of a balancing network is defined as the maximum number of balancers a token can traverse starting from any input wire. A balancing network is quiescent if:

$$\sum_{i=0}^{w-1} x_i = \sum_{i=0}^{w-1} y_i \tag{3}$$

Let $m = \sum_{i=0}^{w-1} y_i$, we say a balancing network (during a quiescent period) satisfies the step property if $\forall i, 0 <= i < w; y_i = \lceil \frac{m-i}{w} \rceil$. Furthermore if a balancing network satisfies the step property, it is a called a counting network.

It is important to mention that if our network satisfies the step property, it can be easily adapted to count the number of tokens that have traversed the network. The step property simply says that if the number of tokens that traversed the network, is divisible by w, for every output wire, y_i must be the same, but if a new token arrives, then it must emerge on output wire y_0, next one in y_1 and so on.

2.5.1 Bitonic Counting Networks

A 2k-merger is a balancing network which is described recursively as follows:

If $k = 1$ the 2-merger is a single balancer with two inputs x_0, x_1 and two outputs y_0, y_1. If $k > 1$, it has two input sequences x, x' of width k and two k-mergers. The input for the first merger is compounded by the even subsequence

x_0, x_2,...,x_{k-2} of x and odd subsequence x'_1, x'_3,...,x'_{k-1} of x'. In the same way, the second merger is fed with the odd subsequence of x and the even subsequence of x'. Finally, the outputs of the first merger are sent as top inputs of a set of 2k balancers. The second merger outputs are connected to the bottom wires of the mentioned balancers. A 2k-merger is designed in such a way that if sequences x, x' fulfill the step property (nothing is postulated about its union), its output also do so.

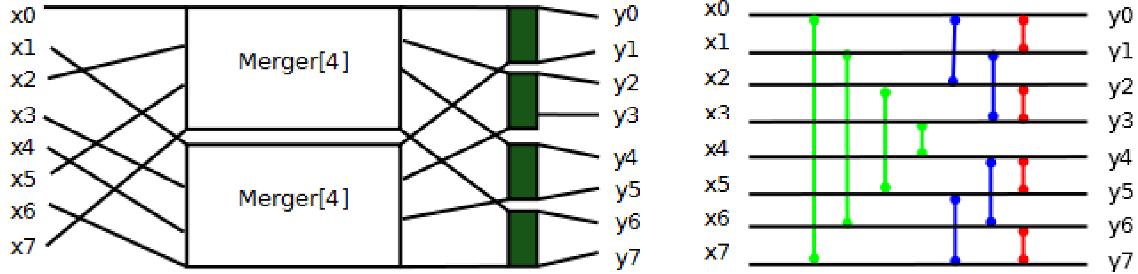
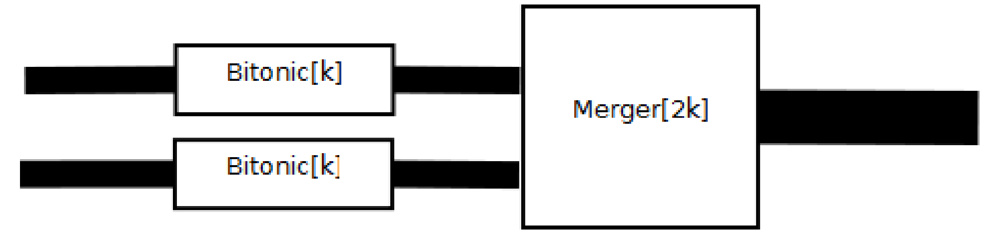


Figure 3: On the left-hand size we see the logical structure of a 8-Merger network, compounded by two 4-Merger blocks and an additional layer of balancers. On the right-hand side, we see the physical layout of the network. The $\log 8 = 3$ different layers of balancers are color coded.

A 2k-bitonic counting network is defined in the same way (k is a power of two):

If $k = 1$, a 2-bitonic counting network is defined a single balancer. If $k > 1$ it is compounded by two k-bitonic whose output wires are used as inputs for a 2k-merger.



2.5.2 Correctness of Bitonic Counting Networks

Lemma 1 *If a sequence x has the step property, then so do all its subsequences.*

We claim 2k-bitonic is a counting network and provide a brief sketch of the inductive proof which shows this balancing network satisfies the step property. We will define an auxiliary variable j, such that $k = 2^{j-1}$ and apply induction on j. If $j = 1$, $k = 1$; our 2-bitonic network consists of a single balancer which by definition satisfies the step property. Recall the construction of a 2k-bitonic network ($j + 1$, $k = 2^j$), compounded by two k-bitonic (j, $k = 2^{j-1}$) networks whose outputs are sent to a 2k-merger. According to our induction hypothesis our k-bitonic networks outputs follow the step property, so the proof must be focused in showing the 2k-merger satisfies the step property given the preconditions. Lemma 1 is recalled in the development of the proof.

2.5.3 Periodic Counting Networks

Let x to be a sequence of indices, A level i chain of x is a subsequence of x whose indices have the same i low-order bits. For example, the subsequence x^E of entries with even indices is a level 1 chain, as is the subsequence x^O of entries with odd indices. The A-cochain of x, denoted as x^A, is the subsequence whose indices have the two low-order bits 00 or 11. For example, the A-cochain of the sequence x_0, x_1,...,x_7 is x_0, x_3, x_4, x_7. In the same way the B-cochain, denoted as x^B, of a sequence is defined as the subsequence of indices whose low-order bits are 01 and 10.

A 2k-block network is defined as follows:

For $k = 1$, it consists of a single balancer. For $k > 1$, it is compounded by two k-blocks. The first block is fed with the inputs corresponding to the A-cochain x^A whereas the second block takes the B-cochain x^B subsequence as depicted in picture 4. Finally the outputs of the blocks are the top and bottom inputs respectively, for a set of 2k balancers.

A 2k-periodic (k is a power of 2) counting network is constructed from $\log 2k$ 2k-block networks. The outputs of the every block are sent in the same order to the next item in the chain. We claim that 2k-periodic network satisfies the step property.

2.5.4 Correctness proof

The correctness proof for the 2k-periodic counting network is also inductive in $2k$ and lies on proving 2k-block network. Again, Lemma A is quite useful in this task.

2.5.5 Analysis of Bitonic and Periodic Counting networks

A complexity analysis for the depth of Bitonic and Periodic Counting Networks in terms of layers, implies to solve the following recurrences which lead to a complexity $O(\log^2 2k)$:

$$DMerger(2k) = DBlock(2k) = \begin{cases} 1; k = 1 \\ 1 + 2 * DMerger(k); k > 1 \end{cases}$$

$$DBitonic(2k) = \begin{cases} 1; k = 1 \\ DMerger(2k) + 2 * DBitonic(k); k > 1 \end{cases}$$

$$DPeriodic(2k) = \begin{cases} 1; k = 1 \\ \log 2k * DBlock(2k); k > 1 \end{cases}$$

The most important applications of counting networks come from the implementation of highly concurrent data structures for threads synchronization like shared counters, producer/consumer buffers and barriers. If contention is sufficiently high, the use of counting networks have been proven to benefit throughput.

In order to measure the performance of a counting network, it is important to introduce the concept of saturation. The network saturation S is defined to be the number of tokens n present in the network divided by the number of balancers. For bitonic and periodic networks $S = \frac{2n}{wd}$, where $w = 2k = width$ and $d = depth$. If $S > 1$, the network is oversaturated, and undersaturated if $S < 1$. When a network is oversaturated, its throughput is dominated by per-balance contention, as we have more threads than balancers. On the other hand, throughput in a undersaturated network is governed by network depth which means it is possible to maximize throughput with a convenient choice

of width and depth parameters. If we analyze latency, it is easy to see that this property is exclusively influenced by network depth.

A balancing network does not need to be follow the step property to be useful. A *threshold network* of width w is a balancing network with input sequence x and output sequence y, such that the following holds:

In any quiescent state, $y_{w-1} = m$ if and only if $mw <= \sum x_i < (m + l)w$.

Informally, a threshold network can detect each time w tokens have passed through it. A counting network is a threshold network, but not viceversa. Both the 2k-block and 2k-merger networks are threshold networks and are normally used in the implementation of *Barriers*.

Finally, a *Busch-Mavronicolas* or *BM counting network* is compounded by $w = 2^k$ inputs; and $t = p * w$ outputs, for any $p > 1$. It has more outputs than inputs which has proven to provide more robustness to high contention. Experiments comparing all these counting networks suggest that BM networks perform better than bitonic and periodic under most conditions, but bitonic design outperforms all others under conditions of minimal contention [2].

3 Distributed Sorting

In this section, two techniques for parallel sorting will be discussed and contrasted in the same way as for counting networks. As a promise, many concepts from previous section will be reused.

3.1 Sorting Networks

A *comparator* is to a comparison network which a balancer is to a balancing network and their structure is similar with one big difference: comparators are synchronous which means they output values only when both inputs have arrived. A comparison network is compounded by many comparators interconnected in a convenient arrangement like in counting networks. We say a *comparison network* with input values x_0, x_1,...,x_{w-1} is a sorting network if output values are equal to input values but sorted in descending order.

The central fact in sorting networks theory is that balancing and comparison networks are *isophormic*.

Lemma 2 *If a balancing network counts, then its comparison counterpart also does.*

Lemma 3 *If a sorting network sorts every input sequence of 0s and 1s, then it sorts any sequence of input values.*

The complete proof for lemmas 1 and 2 can be found at [3].

3.1.1 Implementation of a Bitonic Sorting Network

We have chosen a Bitonic Sorting network for illustration purposes. The network can be represented as a collection of d layers of $\frac{w}{2}$ comparators. For this purpose, a table of size $(\frac{w}{2})d$ is defined in such a way that entry (i, l) contains two numbers that describe which two wires meet in balancer i, layer l.

Listing 1: Implementation of parallel sorting using a Bitonic sorting network in Java.

```
1  public class BitonicSort{
2      static final int[][][] bitonicTable = ...;
3      static final int width = ...; //counting network width
4      static final int depth = ...; //counting network depth
5      static final int p = ...; //number of threads
6      static final int s = ...; // a power of 2
7      Barrier barrier;
8      .....
9      public <T> void sort(Item<T>[] items) {
10         int i = ThreadID.get();
11         for(int l = 0; l < depth; l++){
12             barrier.await();
13             for(int j = 0; j< s; j++){
14                 int north = bitonicTable[(i*s) + j][l][0];
15                 int south = bitonicTable[(i*s) + j][l][1];
16                 if(items[north].key < items[south].key){
17                     Item<T> temp = items[north];
18                     items[north] = items[south];
19                     items[south] = temp;
20                 {
21             }
22         }
23     }
24 }
```

Assume for simplicity we wish to sort an array of $2 * p * s$ elements, where p is the number of threads and $p * s$ is a power of two. The network has $p * s$ comparators at every layer. Each thread emulates the work of s comparators in every round (defined by counter l in outer loop). In each round a thread performs s comparisons in a layer of the network, switching the array items if necessary. Note that in every layer the comparators join different wires, so no two threads attempt to exchange the items of the same entry avoiding synchronization when swapping them; however if some thread finished his work at round l it must wait until all threads are done. That is the purpose of the *Barrier* object. A Barrier for p threads is a synchronization mechanism which provides an *await* method which does not return until all threads have called it. The Barrier itself is a nice application of balancing networks theory. For a detailed explanation of a Barrier implementation in Java, read [4].

3.2 Performance and Robustness of Sorting Networks

If we take our BitonicSort as an example, we can easily conclude that it sorts the input in $O(s\log^2 p)$ where s is a constant which depends on the input size and determines how many sequential computations will be executed in every round. We claim that sorting networks are suitable for small data sets where the cost of accessing the items is not expensive (e.g they are in main memory). If the data set is extremely large and do not fit in main memory, accessing an item can become extremely expensive. In those cases, we need to keep as much locality of reference as possible. An algorithm like BitonicSort where an item is accessed by different threads throughout the rounds (generating a boost of cache misses at the start of every new round), can be simply too expensive. But all are not bad news, there are some other techniques like Sample Sorting, that address this problem efficiently.

3.3 Sample Sorting

As stated for the analysis of sorting networks, algorithms based on this mechanism are appropiate for small data sets that reside in memory. For sets which do not fit in main memory, our goal is to minimize the number of threads that access a given item through randomization. In that sense Sample Sorting can be viewed as a generalization of Quicksort algorithm, but instead of having one pivot, we have $p-1$, where p is the number of threads. A pivot or splitter is just an index we use to split the set into p subsets or buckets. The algorithm has three steps:

1. Threads choose $p-1$ splitter keys to partition the data set into p buckets. The splitters are implemented, in such a way that all threads can read them very quickly. Notice this step requires a consensus among all threads.
2. Each peer sequentially processes its $\frac{n}{p}$ items, moving each item to the right bucket. A binary search is performed over the array of splitters. Formally a key i belongs to bucket j if $data[i] > splitters[j]$, where $0 <= j <= p-1$, $0 <= i <= n$. If $data[i] < splitters[0]$, then it belongs to bucket 0.
3. Each thread sorts the items in its bucket using any sequential sorting algorithm like Quicksort.

As a further clarification, a synchronization mechanism like Barriers must be used so that all threads are always in the same stage of the process.

The efficiency of the algorithm relies mostly on steps 1 and 3. The optimal scenario is when each thread ends with a bucket of the same size, however if the selection of splitters is not optimal, the imbalance produces an extra delay because of the synchronization enforcement. The splitters are selected using random sampling. Each thread picks randomly s samples (s is a small number like 32 or 64) from its $\frac{n}{p}$ corresponding subset. This sums up $s*p$ samples which are then sorted using a parallel algorithm like BitonicSort. Finally each thread reads $p-1$ splitters in positions s, $2s$,...,$(p-1)s$. This sampling process reduces the effects of an uneven distribution among the $\frac{n}{p}$ size data sets accessed by the threads.

3.4 Performance and Complexity of Sample Sorting

We will consider each step of the algorithm. Step 1 corresponds to the random sampling which takes $O(\log^2 p)$ under the assumption of a Bitonic sort. Step 2 implies to move $\frac{n}{p}$ items to their right buckets, performing a binary search every time. It means $O(\frac{n}{p}\log p)$. Step 3 depends on the sequential algorithm used. Assuming a comparison-based efficient algorithm, it leads to $O(\frac{n}{p}\log\frac{n}{p})$. Finally the dominating term is the third one.

Our complexity analysis have done several assumptions which are valid in most scenarios. If the item's key size is known and fixed, we can use algorithms like Radixsort which can sort in linear time. If that is the case, the assymptotic behaviour improves in a considerable way. Furthermore if there are enough information about the probability distribution of the input set, sampling can be skipped and we could get the values from the probability mass of the keys avoiding the run of the Bitonic Sort and hence a round of expensive accesses to items.

3.5 Some other alternatives to Sample Sorting

Sample Sorting is one in a long list of alternatives for paralell sorting. There are some other algorithms like *Flash Sorting*, *Parallel Merge Sorting*, *Parallel Radix Sorting* which have proven to work efficiently too. Last one catches our attention because it is not comparison-based and can sort in linear time under the right conditions. Lots of effort have been devoted in enhancing the original algorithm, from its sequential version to parallel versions with load balancing. Assume an input of n integer keys which are represented as sequences of b bits. The algorithm sorts the input in $\frac{b}{g}$ rounds where $g < b$ and $b \mod g = 0$. In the first round, we consider the least g significant bits of the inputs, so that all the keys having the same g least significant bits are located in a "bucket", leading to a sequence 2^g buckets holding our keys. The next round considers the next g least significant bits of the keys and repeats the process, but keeping the order of the keys and the buckets in the previous round. The parallel version of the algorithm, splits the input set in $\frac{n}{p}$ subsets, one per each processor. Then every processor, runs the sequential algorithm over the subset, generating 2^g local buckets. The variants of the algorithm differ in the redistribution process for the next round. Load Balanced Parallel Radix Sort [5] and Partitioned Paralell Radix Sort [6] are remarkable; the first for focusing in processor load balance and the second for minimizing the communication inherent to redistribution phase after every round.

4 Conclusion

Nowadays, it is highly probable for any piece of software to rely intensively, at some stage, on counting and sorting operations for which we have provided several consistent parallel options that reduce memory contention and do an efficient utilization of the resources taking into account details of the system architecture like caches and shared buses. Although the examples provided in this report are focused in shared-memory environments, these algorithms can be easily ported to message-passing architectures where bandwidth utilization becomes crucial.

References

[1] M. Herlihy and N. Shavit, "Concurrent objects," in *The Art of Multiprocessor Programming*, pp. 45–69, Burlington, USA: Elsevier Inc., 2008.

[2] E. N. Klein, C. Busch, and D. R. Musser, "An experimental analysis of counting networks," *Journal of the ACM*, pp. 1020–1048, September 1994.

[3] J. Aspnes, M. Herlihy, and N. Shavit, "Counting networks," *Journal of the ACM*, 1994.

[4] M. Herlihy and N. Shavit, "Barriers," in *The Art of Multiprocessor Programming*, pp. 397–415, Burlington, USA: Elsevier Inc., 2008.

[5] A. Sohn and Y. Kodama, "Load balanced parallel radix sort," in *ICS '98: Proceedings of the 12th international conference on Supercomputing*, (New York, NY, USA), pp. 305–312, ACM, 1998.

[6] S.-J. Lee, M. Jeon, D. Kim, and A. Sohn, "Partitioned parallel radix sort," *J. Parallel Distrib. Comput.*, vol. 62, no. 4, pp. 656–668, 2002.