7 Parallel Algorithms and Techniques

Question: What do parallel algorithms look like?

Question: How do parallel algorithms differ from sequential algorithms?

This chapter introduces some general principles of parallel algorithm design. We will consider a few case studies to illustrate broad approaches to parallel algorithms. As already discussed in Chapter 5, the underlying goal for these algorithms is to pose the solution into parcels of relatively independent computation, with occasional interaction. In order to abstract the details of synchronization, we will assume the PRAM or the BSP model to describe and analyze these algorithms. It is a good time for the reminder that getting from, say, a PRAM algorithm to one that is efficient on a particular architecture requires refinement and careful design for a particular platform. This is particularly true when 'constant time' concurrent read and write operations are assumed. Concurrent read and writes are particularly inefficient for distributed-memory platforms, and are inefficient for shared-memory platforms as well. It requires synchronization of processors' views of the shared memory, which can be expensive.

Recall that PRAM models focus mainly on the computational aspect of algorithm, whereas practical algorithms also require close attention to memory, communication, and synchronization overheads. PRAM algorithms may not always be practical, but they are easier to design than those for more general models. In reality, PRAM algorithms are only the first step towards more practical algorithms, particularly on distributed-memory systems.

Parallel algorithm design often seeks to maximize parallelism and minimize the time complexity. Even if the number of actually available processors is limited, higher parallelism translates to higher scalability in practice. Nonetheless, the work-time scheduling principle (Section 3.5) indicates that low work complexity is paramount for fast execution in practice. In general, if the best sequential complexity of solving the given problem is, say, $T_o(n)$, we would like the parallel work complexity to be $O(T_o(n))$. It is a common algorithm design pattern to assume up to $T_o(n)$ processors and then try to minimize the time complexity. With maximal parallelism, the target time complexity using $T_o(n)$ processors is $O(1)$. This is not always achievable, and there is often a trade-off between time and work complexity. We then try to reduce the work complexity to $O(T_o(n))$, without significantly increasing the time complexity. Sometimes we start by assuming an even higher number of processors, which is not $O(T_o(n))$, *e.g.*, $T_o^2(n)$. This is not a practical algorithm on its own, but it can sometimes be useful as a subroutine executed on small subsets of the input.

Once the algorithm is ready, we rely on the work-time scheduling principle to manage its execution on the available hardware. A caveat: the work-time scheduling principle is designed for PRAM algorithms, and very much like PRAM algorithms, it focuses on the computational aspect of the solution. Directly applying this principle to map a PRAM algorithm onto a limited number of processors does not always exhibit the best performance. For example, in the context of the BSP model, communication overheads are lower if the virtual processors that inter-communicate substantially are mapped onto the same physical processor. For shared-memory machines, different PRAM algorithms can lead to different synchronization overheads. This can make an algorithm that is apparently faster on paper actually slower in practice. Hence, even though the theoretical algorithm may naturally suggest a task decomposition, it may need to be adjusted to account for the hardware architecture.

While the focus of this chapter is on parallel algorithmic style of thinking and ab-initio design, it also presents a few cases of opportunistically finding inherently parallel steps in known sequential algorithms.

We have already discussed the reduction algorithm in Section 1.6, which is an example of a parallel algorithm organized as a binary tree of computations. This is an oft-occurring paradigm, where each processor performs some computation independently of others, creating a partial result. The processors then combine the partial results pair-wise into the final result, going up a tree. In its simplest form, this paradigm yields a work complexity of $O(n)$ and a time complexity of $O(\log n)$.[1] It works well for problems that have an $O(n)$ sequential solution. We begin with an example slightly more complex than the addition algorithm of Section 1.6. It demonstrates the common algorithmic technique called the *Binary Tree computation*, which is a special case of the well-known Divide and Conquer paradigm.

[1] At the risk of being repetitive, the base of log is assumed to be 2, unless explicitly listed.

7.1 *Divide and Conquer: Prefix-Sum*

Formally, the prefix-sum of a list of data items: $d_i, i \in 0..n$ is another list $s_i, i \in 0..n$ such that

$$s_i = \sum_{j=0}^{i} d_j,$$

where $a..b$ indicates the range a to b, both inclusive. A sequential algorithm to compute the prefix-sum is simply:

Listing 7.1: Sequential Prefix-Sum

```
// Compute in s the prefix-sum of d
s[0] = d[0]
for(i=0; i<=n; i++) {
  s[i] = s[i-1] + d[i]
}
```

This is an efficient sequential algorithm – it takes $O(n)$ steps, and no algorithm may take fewer steps asymptotically. However, each step depends on the previous computation, precluding any meaningful parallelism.

Prefix-sum is a good example of problems where an efficient sequential algorithm does not admit parallelism, but a fresh parallel design affords significant parallelism. Trying to factor out and reuse common computation is an important tool in sequential algorithm design. That is precisely what causes the dependency, however. Instead, a parallel algorithm is designed to subdivide a problem into independent parts, even if those parts repeat some computation.

For the prefix-sum problem, the main question is how to compute $s[i]$ without the help of $s[i-1]$. An extreme way to break the dependency is as follows.

Listing 7.2: Trivial Parallel Prefix-Sum

```
// Compute in s the prefix-sum of d
forall processor i in 0..n
  s[i] = d[i]
  for(j=0; j<=i; j++) {
      s[i] += d[j]
  }
```

This method is not particularly useful. The slowest processor now takes the same time that the single processor algorithm takes. The total time remains $O(n)$. The work complexity unnecessarily jumps to $O(n^2)$, well short of the optimal $O(n)$. This is not surprising, as there were too many dependencies broken above: $O(n^2)$, in fact. The computation at each iteration depends transitively on all earlier iterations.

For some problems, it may be possible to break dependencies entirely at a small cost. For most, there is a trade-off between the parallelism obtained and the work complexity achieved. There are two general approaches to break such a chain of dependencies: top-down or bottom-up. The first subdivides a long dependency chain into smaller chains, breaking the dependency of each chain on other chains. For example, it may be possible to carry out a partial computation of $s[i]$ independently of $s[i-1]$, say, for some i. This is then followed by a subsequent step to compute the final $s[i]$ in parallel for different i. The second – bottom-up – approach computes in parallel partial results in small groups of i, followed by processing the dependency chain on these partial group-results. This yields a smaller version of the original problem, and hence a smaller chain. The following examples explain these approaches.

Parallel Prefix-Sum: Method 1

We first break the dependency chain only between $s[\frac{n}{2}]$ and $s[\frac{n}{2}+1]$, allowing $s[\frac{n}{2}+1]$ to start to be computed before $s[\frac{n}{2}]$ is available. This, in turn, breaks the dependency of all $s[i], i > \frac{n}{2}$, on any $s[i], i \leq \frac{n}{2}$.

Listing 7.3: Partial Prefix-Sum: breaking dependency

```
// Compute in s the prefix-sum only of d[1+n/2..n]
s[1+n/2] = d[1+n/2]
for(i=2+n/2; i<=n; i++) {
  s[i] = s[i-1] + d[i]
}
```

Clearly, this does not compute the full prefix-sum, but a partial one. We will see that this works because it is simple to compute the full prefix-sums from this partial prefix sum later. The partial computation above can proceed in parallel with the computation of the prefix-sum for values of $i \leq \frac{n}{2}$. It is clear that the work complexity of computing the prefix-sum of the first half of s and that of the partial prefix-sum of the second half are equal to each other. Once both halves are computed, the full prefix-sum for $i > \frac{n}{2}$ remains to be computed. However, now that the correct value of $s[\frac{n}{2}]$ is known, the partial sums can be completed in a single parallel step in $O(1)$ time with $O(n)$ work as follows,

Listing 7.4: Partial Prefix-Sum: breaking dependency

```
// Update prefix-sum of the second half of s
forall processor p in (1+n/2)..n
  s[p] += s[n/2]
```

as long as $s[\frac{n}{2}]$ is accessible in parallel to all processors (of Listing 7.4).

By itself, the trick above does not lead to an improved complexity, because the prefix-sum still needs to be computed for each half, within which the dependencies remain unbroken. One sequential problem has been subdivided into two sequential problems, each of a smaller size. Using the divide and conquer paradigm, one can divide the problem recursively until the size of the remaining problem reduces to 1 (or a constant number). The recurrence relations for the time and work complexity, respectively, are then:

$$t(n) = t\left(\frac{n}{2}\right) + O(1)$$
$$W(n) = 2W\left(\frac{n}{2}\right) + O(n)$$

with $t(1) = O(1)$ and $W(1) = O(1)$. Hence, $t(n) = O(\log n)$ and $W(n) = O(n \log n)$, which is not optimal. This analysis assumes that $s[\frac{n}{2}]$ can be accessed by all processors concurrently. This analysis holds for CREW PRAM model and other models allowing concurrent read. Exclusive read, as in EREW PRAM, or a broadcast of $s[\frac{n}{2}]$ to all $\frac{n}{2}$ virtual processors, as in BSP, would require additional time. (See Exercise 7.2.)

Parallel Prefix-Sum: Method 2

An alternate way to break the dependencies is to break many of them at one go. For example, we might break all dependencies between odd and even indexes.

Listing 7.5: Partial Prefix-Sum: odd-even separation

```
1 // Compute in s the prefix-sum of d
2 in parallel
3   Recursively Compute prefix-sum considering only even index i
4   Recursively Compute prefix-sum considering only odd index j
5 forall processor p in 1..n
6   s[p] += s[p-1]
```

Once the two partial prefixes are known, they can derive the full sum quickly from each other, as on line 6 above. This variant of top-down interleaved decomposition has the same time and work complexity as the previous block decomposition. (See Exercise 7.3.) However, it does not suffer from the bottleneck of broadcasting $s[\frac{n}{2}]$ to the entire second half. Please note that the recursive structure of the overall algorithm (as of all the algorithms in this section) is similar to that of Method 1.

Parallel Prefix-Sum: Method 3

Let us discuss the bottom-up approach next.

Listing 7.6: Partial Prefix-Sum: odd-even separation

```
// Compute in s the prefix-sum of d
forall processor p, 0<p≤n and odd p
  s[p] = d[p]+d[p-1] // Pre sum pairs
```

```
Recursively compute prefix-sum on odd indexes of s
forall processor p, 0<p≤n
  s[p] += s[p-1]  // Post sum
```

The first *forall* sums the input values in pairs. The next step recursively computes the prefix-sum on these pair-wise sums, which are $\frac{n}{2}$ in number. This recursion will also proceed in the same bottom-up fashion. The second *forall* computes the final prefix-sum from the prefix-sum of the pairs. The structure of the algorithm is depicted in Figure 7.1. In Method 3, the dependencies are removed by reducing the input set first (in parallel, of course). The recurrence relations for

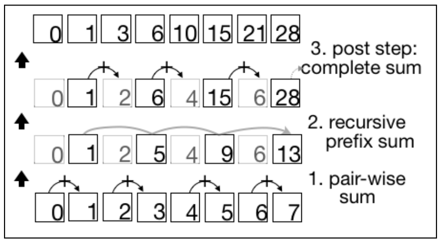


Figure 7.1: Recursive Prefix-Sum. Each row of boxes denotes the state of the array s after each step. The numbers in the bottom row constitute the input values. The first and last steps are each fully parallel and completed in $O(1)$ by $O(n)$ processors.

time and work complexity for this algorithm are

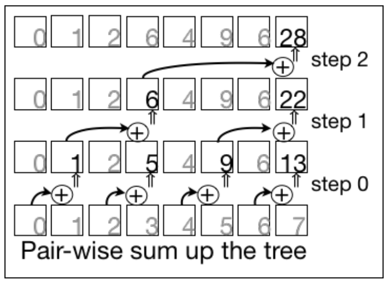
$$t(n) = t\left(\frac{n}{2}\right) + O(1)$$
$$W(n) = W\left(\frac{n}{2}\right) + O(n)$$

This yields the optimal work complexity of $O(n)$, while retaining the time complexity of $O(\log n)$. An unrolling of the recursive statement shows that the structure of the solution is similar to the binary-tree based computation of Section 1.6: a reduction going up the tree followed by the completion step going down the tree, as shown in Listing 7.7 and Figure 7.2.

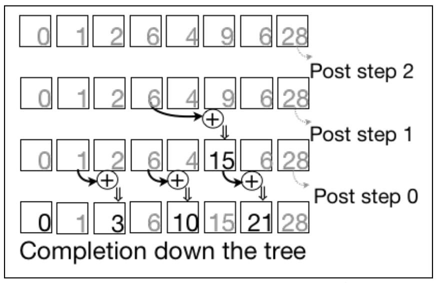
Listing 7.7: Partial Prefix-Sum: odd-even separation

```
// Compute in s the prefix-sum of d
for step = 0..(logn - 1)
  j = 2^step
  forall processor i in 1..(n/j)
    s[2*j*i-1] += s[j*(2*i-1)-1] // Reduce up the tree
for step = (logn - 1)..0
  j = 2^step
  forall processor i in 1..(j-1)
    s[j*(2*i+1)-1] += s[2*j*i-1] // Complete down the tree
```

Figure 7.2(a) shows the steps of the upward reduction pass, and Figure 7.2(b) shows the downward completion pass. Each level is shown as a row of boxes, which depict the values in the array after each step. In the upward pass, pairs are summed at each step, with the number of such sums halving at each step. The sum so produced at the last step of the upward pass is the sum of all values, which is also the prefix-sum $s[n]$. In the downward pass, the prefix-sum values are computed at each level from the prefix-sum evaluated at the level above. The bottom-most level then computes the final prefix-sum.



(a) Upward pass



(b) Downward pass

Figure 7.2: Prefix-Sum algorithm in two passes up and down a binary tree. The values in the dark font are the ones updated at that step. Hence, the number of active processors at each step are indicated by the number of values in the dark font in that row.

Having found an efficient algorithm to reduce the dependency in the prefix-sum, we will soon see that prefix-sum can, in turn, be used as a subroutine to break similar dependencies in other problems. Prefix-sum is more generically called a *Scan*, which has no connotation of summing. *Exclusive scan* is defined as a scan where the i^{th} element is not included in the i^{th} result, *i.e.*,

$$s_i = \sum_{j=0}^{i-1} d_j.$$

Scan algorithms can easily be modified to compute exclusive scan. This is left as an exercise (Exercise 7.4).

7.2 *Divide and Conquer: Merge Two Sorted Lists*

We take another example of a Divide and Conquer algorthm. Merging of two sorted lists into a single sorted list is an important tool for sorting and other problems. (For this discussion, assume sorting is in increasing order.) The sequential algorithm to merge two lists, say, *list*1 and *list*2, comprising n elements each, is as follows:

Listing 7.8: Sequential Merge

```
i = j = 0
while(i < n and j < n)
  if(list1[i] < list2[j])
    output list1[i++]
  else
    output list2[j++]
while(i < n)
  output list1[i++]
```

```
while(j < n)
  output list2[j++]
```

Let the output list be called *list*3. This algorithm takes $O(n)$ steps. It is inherently sequential because only after the result of the comparison of the pair $list1[i]$ and $list2[j]$ is known at line 3 in an iteration, that the pair to compare in the next iteration is determined.

Parallel Merge: Method 1

We first consider breaking this dependency similarly to the previous section. The standard binary subdivision of *list*1 and *list*2 into two halves each, followed by the merger of each pair does not yield two independent sub-problems. However, for each half, it is easy to determine the block of the other list that it needs to merge with, so that the recursive sub-problems do become independent.

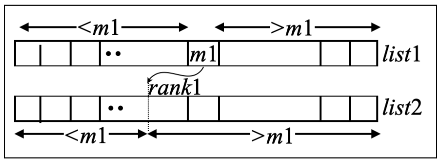


Figure 7.3: Recursive Parallel Merge 1: $m1$ is the middle element of *list*1. *rank*1 is the positions of $m1$ in *list*2. Elements smaller than $m1$ in each list can be merged with each other, the remaining can be merged with each other. $m1$ need not participate in either merge.

Let the *rank* of element x in *list*, *i.e.*, $Rank(x, list)$, be the number of elements in *list* that are less than x.[2] Let us use the shorthand *rank*1 for $Rank(m1, list2)$, the rank of $m1$ in *list*2. $m1$ here is the middle element of *list*1. A single processor can find *rank*1 through binary search for $m1$ in *list*2 in $O(\log n)$ time. This means, *rank*1 elements of *list*2 are smaller than $m1$, just as $\frac{n}{2}$ elements of *list*1 are smaller than $m1$. The remaining elements are greater than $m1$ (see Figure 7.3). Hence, the two smaller sets of elements are the smallest $rank1 + \frac{n}{2}$ elements of *list*3. This implies that the first part of *list*3 can be obtained by merging the first $\frac{n}{2}$ elements of *list*1 with the first *rank*1 elements of *list*2. The second part of *list*3 can be obtained by merging the remaining elements of *list*1 with the remaining elements of *list*2. These are two independent merge sub-problems. The lists to merge need not have the same length any more. In the following listing, we assume list to have $n1$ and $n2$ elements, respectively. The merged list is produced in *list*3.

[2] Note that this usage of the term is slightly different from that in Chapter 6.

Listing 7.9: Parallel merge 1: Top Down dependency breaking

```
ni = max(n1, n2)
nj = min(n1, n2)
listi = longer of(list1, list2) // either if n1 == n2
listj = the other list
ranki = Search listi[n/2] in listj
in parallel
```

```
  list3[0..ni/2+ranki-1] =
      Merge listi[0..ni/2-1] with listj[0..ranki-1]
  list3[n/2+rank..n1+n2] =
      Merge listi[ni/2..ni] with listj[ranki..nj]
```

There is a shortcoming of the subdivision described above. If $ranki \ll \frac{ni}{2}$, the second merger possibly has too much work remaining – one list has $\frac{ni}{2}$ elements but the other may have as many as nj elements of $listj$. The recurrence relations for the algorithm in Listing 7.9 for the EREW PRAM model is:

$$t(n) \leq t\left(\frac{3n}{4}\right) + \log n$$
$$W(n) \leq W\left(\frac{3n}{4}\right) + W\left(\frac{n}{4}\right) + \log n$$

This implies $t(n) = O(\log^2 n)$ and $W(n) = O(n)$. In practice, given a certain number of, say, shard-memory processors, one would allocate more processors to the larger subproblem. Of course, one might consider subdividing into more balanced sub-problems. (See Exercise 7.12.)

Parallel Merge: Method 2

The other manner of breaking dependency in the Prefix-sum case can also be employed for merging. Note that merging is equivalent to finding the ranks of all elements of $list1$ in $list2$ and all elements of $list2$ in $list1$. The final rank in $list3$ of, say, element $list1[i]$ is $i + Rank(list1[i], list2)$.

Let $Ranklist(list_x, list_y)$ denote the list of the ranks $Rank(list_x[i], list_y)$ for all elements $list_x[i]$ of $list_x$. Once $Ranklist(list1, list2)$ is given as the list of ranks $Rank1$ and $Ranklist(list2, list1)$ is given as the list $Rank2$, we can compute $list3$ as follows:

Listing 7.10: Merge by rank

```
in parallel
  forall processor p, 0 ≤ p < n
    list3[p+Rank1[p]] = list1[p]
  forall processor q, 0 ≤ q < n
    list3[q+Rank2[q]] = list2[q]
```

In the first step of the rank computation algorithm, we recursively merge only the even positioned elements of $list1$ with the even positioned elements of $list2$. In other words, we find the rank of $list1[i]$ for even values of i in $list2_e$. We will use the shorthand $list_e$ to denote the even sublist, meaning $list_e[j] = list[2j]$, for $0 \leq j < \frac{n}{2}$. We do not need to create a separate $list_e$, but instead use the term to restrict consideration to the even indexes of *list*. Once $Ranklist(list1_e, list2_e)$ and $Ranklist(list2_e, list1_e)$ are known, we use them to find $Ranklist(list1, list2)$ and $Ranklist(list2, list1)$. We will use the shorthand $Rank_e1$ for the rank of elements of $list1_e$ in $list2_e$, and $Rank_e2$ for those of elements of $list2_e$ in $list1_e$. Figure 7.4

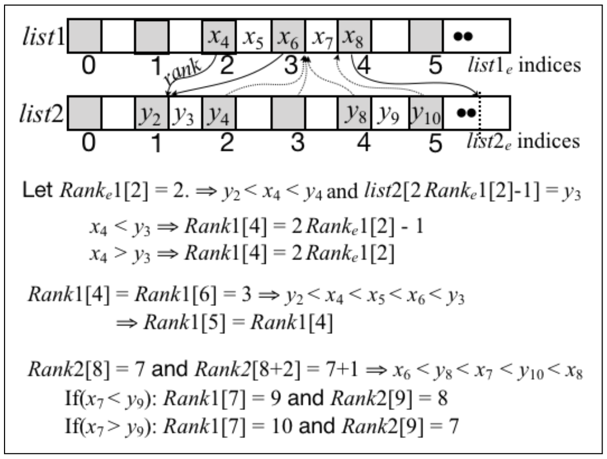

list1
x_4 x_5 x_6 x_7 x_8 ••
0 1 2 3 4 5 $list1_e$ indices
rank
list2
y_2 y_3 y_4 y_8 y_9 y_{10} ••
0 1 2 3 4 5 $list2_e$ indices
Let $Rank_e1[2] = 2. \Rightarrow y_2 < x_4 < y_4$ and $list2[2\,Rank_e1[2]\text{-}1] = y_3$
$x_4 < y_3 \Rightarrow Rank1[4] = 2\,Rank_e1[2] - 1$
$x_4 > y_3 \Rightarrow Rank1[4] = 2\,Rank_e1[2]$
$Rank1[4] = Rank1[6] = 3 \Rightarrow y_2 < x_4 < x_5 < x_6 < y_3$
$\Rightarrow Rank1[5] = Rank1[4]$
$Rank2[8] = 7$ and $Rank2[8+2] = 7+1 \Rightarrow x_6 < y_8 < x_7 < y_{10} < x_8$
If($x_7 < y_9$): $Rank1[7] = 9$ and $Rank2[9] = 8$
If($x_7 > y_9$): $Rank1[7] = 10$ and $Rank2[9] = 7$

Figure 7.4: Recursive Parallel Merge 2: the value of list1$[i]$ is x_i, and that of $list2[j]$ is y_j. $list1_e$ comprises the even positions of *list*1, and $list2_e$ comprises the even positions of *list*2. $Rank_e1$ and $Ranke_e2$ are the rank lists with respect to $list1_e$ and $list2_e$.

demonstrates how to compute *Rank*1 and *Rank*2 from $Rank_e1$ and $Rank_e2$. $Rank_e1$ and $Rank_e2$ may be computed recursively, or similarly to the bottom-up variant of the parallel prefix-sum algorithm. Say, $Rank_e1[i] = r_e$ is the rank of element $list1[2i]$ in $list2_e$. The figure shows this element as x_{2i}. This implies that r_e elements of $list2_e$ are smaller than x_i, meaning $list2[0..2r_e - 1]$ are smaller than x_i and $list2[2r_e] > x_i$. Hence, $Rank(x_i, list2)$ is also $2r_e$ if $x_i < list2[2r_e - 1]$ and $2r_e + 1$ otherwise. (Note that $list2[2r_e - 1]$ is not included in $list2_e$, and hence it was not compared in the recursive merging of $list1_e$ and $list2_e$.) Ranks of all elements of $list1_e$ in *list*2 and those of elements of $list2_e$ in *list*1 can be computed in this manner in parallel with each other, taking $O(1)$ time under CREW PRAM model. Thus, the work complexity to compute $Ranklist(list1_e, list2)$ and $Ranklist(list2_e, list1)$, given $Ranklist(list1_e, list2_e)$ and $Ranklist(list2_e, list1_e)$ is $O(n)$. Concurrent read is required because, say, $Rank(x_{i+1}, list2_e)$ may also be r_e. In that case, $list[2r_e - 1]$ would be required in the computation of both $Rank(x_{i+1}, list2)$ and $Rank(x_i, list2)$.

We next compute the ranks of the odd-index elements of *list*1 and *list*2. These are at $list1[2i + 1]$ for index i of $list1_e$, and similarly for *list*2. (Note that $2i + 1$ may reach beyond the end of *list*1; these edge effects are easy to handle, but we ignore them here for simplicity of description. One may assume the value ∞ at such indexes.) $Rank1[2i + 1]$, which is a shorthand for $Rank(list1[2i + 1], list2)$, can be computed from $Rank1[2i]$ and $Rank1[2i + 1]$, which are known from the previous step. Recall that *list*1 is sorted, and hence $list1[2i] < list1[2i + 1] <$

$list1[2i+2]$. Suppose $Rank1[2i]$ and $Rank1[2i]$ are equal; call them r. This means $Rank1[2i+1]$ must also be r, since $list2[r-1] < list1[2i]$ and $list2[r] > list1[2i+2]$ and hence $list2[r-1] < list1[2i+1] < list2[r]$. We may not always be so lucky though. For example, in Figure 7.4, $Rank(x_6, list2)$ is 3, and $Rank(x_8, list2)$ is much higher, say n. To find $Rank(x_7, list2)$, we must find which elements in the range $y_3..y_{n-1}$ are smaller than x_7. A binary search would find that index but would take too long; we seek an $O(1)$ algorithm.

Realize, however, that we already know the ranks of elements $Rank(y_j, list1)$ for all even j, $3 \leq j < n$. These ranks are all either 7 or 8 in the example. In fact, we are looking for the index k such that $Rank(y_j, list1) = 7$ for $j \leq k$ and $Rank(y_j, list1) = 8$ for $j > k$. In this example, $k = 8$. Thus k can be computed in $O(1)$ time if processor j for each even value of j checks if $Rank(y_j, list1) + 1$ equals $Rank(y_{j+1}, list1)$. The processor – and there is exactly one – that finds it true may now compute $Rank(y_j + 1, list1)$ as well as $Rank(x_7, list2)$ in this example, and more generally $Rank(x_{r1}, list1)$, where $r1$ is $Rank(y_j, list1)$. $Rank(x_{r1}, list2)$ needs to be computed only if $r1$ is even. This is detailed in Listing 7.11.

Listing 7.11: Parallel merge 2: "Bottom Up?" dependency breaking

```
Merge(list1_e, list2_e) // Create Rank1_e and Rank2_e
in parallel // First compute rank of even elements from Rank_e
  forall processor p, 0 <= p < n and p%2==0
    if(list2[Rank1[p]+1] < list1[p])
      Rank1[p] = 2*Rank1[p] + 1
    else
      Rank1[p] = 2*Rank1[p]
  forall processor q, 0 <= q < n and q%2==0
    if(list1[Rank2[q]+1] < list1[q])
      Rank1[q] = 2*Rank1[q] + 1
    else
      Rank1[q] = 2*Rank1[q]
in parallel // Now compute the ranks of odd elements
  forall processor p, 0 <= p < n and p%2==0
    if(Rank1[p] == Rank1[p+2])
      Rank1[p+1] = Rank1[p]
    else if(Rank1[p]+1 == Rank1[p+2]) and Rank1[p]%2 == 1
      if(list2[Rank1[p]] > list1[p+1])
        Rank2[Rank1[p]] = p+2
      else
        Rank2[Rank1[p]] = p+1
  forall processor q, 0 <= q < n and q%2==0
    if(Rank2[q] == Rank2[q+2])
      Rank2[q+1] = Rank2[q]
    else if(Rank2[q]+1 == Rank2[q+2]) and Rank2[q]%2 == 1
      if(list1[Rank2[q]] > list2[q+1])
        Rank1[Rank2[q]] = q+2
    else
      Rank1[Rank2[q]] = q+1
```

The recurrence relations for the algorithm in Listing 7.11 for the CREW PRAM model is:

$$t(n) \leq t\left(\frac{n}{2}\right) + O(1)$$
$$W(n) \leq W\left(\frac{n}{2}\right) + O(n)$$

This implies $t(n) = O(\log n)$ and $W(n) = O(n)$. Work is optimal, but can time complexity be improved? Let us investigate.

Parallel Merge: Method 3

Recall that the main task is to compute the ranks of every element of $list1$ and $list2$ in each other. Each rank can be potentially computed independently of the other ranks. One natural way to partition this task is to subdivide one of the lists, say $list1$, into sublists $list1_m, 0 \leq m < P$, for a given P. Employing $P = n$ processors, each processor may complete its 'merger' in O($\log n$) time by performing a binary search for the singleton element of $list1_m$ in $list2$. *Partitioning* a problem into sub-problems to solve is a common parallel algorithm design technique.

Listing 7.12: Parallel Merge 3

```
forall processor p in 0..n-1
  Rank1[p] = find list1[p] in list2
forall processor q in 0..n-1
  Rank2[q] = find list2[q] in list1
```

This algorithm also performs concurrent reads, as all binary searches proceed in parallel. The time complexity remains $O(\log n)$, since the $2n$ rank computations are all independent of each other, and each performs a binary search through a list of n elements. The work complexity, however, increases to $O(n \log n)$, which is sub-optimal. While this algorithm is simpler in structure than the previous one, the performance is worse. This suggests that processors replicate too much computation, and we should try to factor out some repeated computation.

A closer inspection indicates that $Rank(list1_m, list2) \leq Rank(list_{m+1}, list2)$. Separate binary searches for $list1_m$ and $list1_{m+1}$ disregard this relationship, each proceeding independently of the other. On the other hand, we do not want the search for $list1_{m+1}$ to wait until that for $list1_m$ is complete. On the other side, trying to reduce the time complexity further by performing faster searches for $list1_m$ may require multiple processors per search, leading to an even higher work complexity. We will see that such inexpensive algorithms may be usable on small sub-problems. Let us explore this further through the following digression.

Aside: Parallel P-ary Search

As a sidetrack, consider searching of element x in a sorted $list$ using P processors. Let's say we want to find $Rank(x, list)$. Extending binary search, we subdivide $list$ into $P+1$ blocks, with $n_P = \frac{n}{P}$ elements in each block. (The last block may have fewer elements.) In effect, $list_P[i] = list[n_p * i]$. Processor p determines if $list_P[p-1] \leq x < list_P[p]$. This condition is true for at most one value of p, given that elements in $list$ are unique. If the condition does not hold for any processor p, it implies $x > list_P[P-1]$, *i.e.*, x lies in the last block. In $O(1)$ time with $O(P)$ work, we thus determine the block of $list$ in which x may lie. We recursively allow all P processors to find the rank of x in that block next. This extends the sequential binary search into a P-ary search.

Listing 7.13: P-ary search

```
// Find rank of x in range L..R with P processors
if x < list[L]
  return Rank(x) = L
n_p = (R-L+1)/P
if(n_p <= 1) // Terminate recursion. No need to subdivide.
  if x > list[R]
    return Rank(x) = R
  forall processors p in 1..(R-L)
    if list[L+(p-1)] < x < list[L+p]
      return Rank(x) = L+p
    return success // Some other processor will complete Rank
if x > list[L+P*n_p]
  with P processors: Search in range {L+p*n_p}..R
else
  forall processor p in 1..P-1
    if list[L+(p-1)*n_p] < x < list[L+p*n_p] // Test for equality to find x
      with P processors: Search in range {L+(p-1)*n_p}..{L+p*n_p}
    // x not in this processor's block. Return.
```

Each processor performs $O(1)$ comparisons per invocation of the function above. There are $O(\log_P n)$ invocations. Hence, the total time complexity is $O(\log_P n)$, and the work complexity is $O(P \log_P n)$. This is $O(\log n)$ only for a constant P. In general, this means that the efficiency with P processors is proportional to $O(\frac{\log P}{P})$. Nonetheless, this algorithm scales up to $P = n$, and takes time $O(1)$ with n processors, which equals what brute-force search would take with n processors.

The P-ary search algorithm above demonstrates one other algorithmic technique. It generalizes the binary-tree computation structure, which parallel merge method 3 uses, for example: it recursively considers the even indexes, reducing the problem size by half at each level. Some problems are amenable to partitioning into more than two sub-problems at a time, allocating an appropriate number of processors to each sub-problem. Many of these partitioning problems do not require any post-recursion operation – each sub-problem simply generates a known subset of the solution. The P-ary search is one such example.

Applying partitioning to the merging problem, we may select every k^{th} element of $list1$ into $list1_k$, meaning $list1_k[i] = list1[ik]$. A large value of k reduces the size of the recursive sub-problem. On the other hand, a large k also leave a large number of ranks remaining to be computed after the sub-problem is solved.

Suppose $k = \sqrt{n}$. $list1_k$ and $list2_k$ are each of size $\sqrt{n}$. As a result, we can find the rank of each element of $list1_k$ in $list2$ using $\sqrt{n}$ processors for each search:

Listing 7.14: Parallel Merge 4: $\sqrt{n}$ subdivision

```
// Rank √n elements of list1 in list2
rootn = √n
forall processor p in 0..rootn-1
  with rootn processors P-ary Search list1[p*rootn] in list2[0..n-1]
```

$\sqrt{n}$ processors can find $Rank(list1_k[i], list2)$ for any i in $O(1)$ time using $O(\sqrt{n})$ work. Since there are $\sqrt{n}$ elements in $list1_k$, n processors can compute all of $Ranklist(list1_k, list2)$ in $O(1)$ time, with $O(n)$ work. We can similarly compute $Ranklist(list2_k, list1)$ in $O(1)$ time, with $O(n)$ work. This seems good; except much work remains – we do not yet know the ranks of $(n - \sqrt{n})$ elements of each list. We can compute these ranks by recursively solving smaller merge problems. See the illustration in Figure 7.5 to understand how.

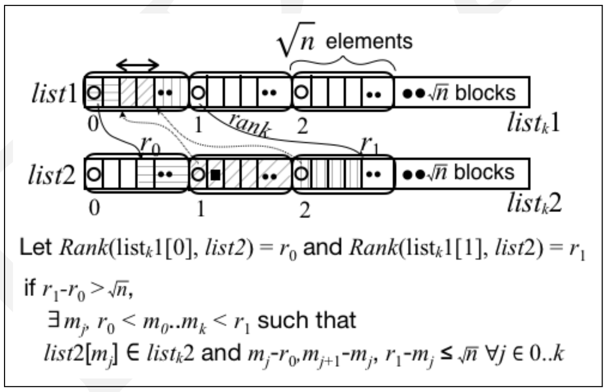

$\sqrt{n}$ elements
$list1$
●● $\sqrt{n}$ blocks
0
1
rank
2
$list_k1$
r_0
r_1
$list2$
●● $\sqrt{n}$ blocks
0
1
2
$list_k2$
Let $Rank(list_k1[0], list2) = r_0$ and $Rank(list_k1[1], list2) = r_1$
if $r_1 - r_0 > \sqrt{n}$,
$\exists m_j$, $r_0 < m_0..m_k < r_1$ such that
$list2[m_j] \in list_k2$ and $m_j - r_0, m_{j+1} - m_j, r_1 - m_j \leq \sqrt{n}\ \forall j \in 0..k$

Figure 7.5: Recursive Parallel Merge 3: Each list is subdivided into blocks of $\sqrt{n}$ elements. The rank of the first element of each block, depicted as circles, is computed first. These ranks help subdivide the merging problem into up to $2\sqrt{n}$ smaller merging problems. Three of these sub-problems are highlighted – the first pair with horizontal hatching, the second with oblique, and the third with vertical hatching.

Consider two consecutive elements of $list1$ that are included in $list1_k$, say $list1[i\sqrt{n}]$ and $list1[(i+1)\sqrt{n}]$. The ranks of these elements in $list2$ are known after the P-ary search. Call them r_i and r_{i+1}. We know that ranks of all elements $list1[x]$, where $i\sqrt{n} < x < (i+1)\sqrt{n}$, are also in the range $r_i..r_{i+1}$. This means that we can decompose the merger into smaller mergers: Merge $list1[i\sqrt{n}..(i+1)\sqrt{n}-1]$ with $list2[r_i..r_{i+1}-1]$. This sub-problem can be large if $r_i \ll r_{i+1}$.

However, in that case, just as in parallel merge method 2, the range $list2[r_i..r_{i+1}]$ contains elements from $list2_k$ whose rank in $list1$ are known. Those elements delineate blocks with no more than $\sqrt{n}$ elements each. Moreover, their ranks in $list1$ are not more than $\sqrt{n}$ apart, as they all lie in the range $(i\sqrt{n})..((i+1)\sqrt{n}-1)$. This ensures that we may now independently merge pairs of blocks of $list1$ and $list2$, respectively. The number of such pairs is at most $2\sqrt{n}$ as at least one block of each pair has $\sqrt{n}$ elements. Thus the recurrence relation for complexity is:

$$t(n) = t(\sqrt{n}) + O(1)$$
$$W(n) = \sqrt{n}W(\sqrt{n}) + O(n)$$

This means that $t(n) = O(\log\log n)$ and $W(n) = O(n\log\log n)$. This $W(n)$ is not optimal, even if the time complexity is now lower. A subtle point to note: each recursive sub-problems computes the ranks only with respect to its block of elements. For example, in Figure 7.5, the recursive sub-problem computes the rank of $list2[k+1]$, the element shown as $\blacksquare$, in the part of $list1$ marked by $\leftrightarrow$. If this computed rank is $srank$, $Rank(list2[k+1], list1)$ is $srank + Rank(list2[k], list1)$.

Parallel Merge: Method 4

The last algorithmic technique we discuss in this section reduces the work complexity of an algorithm with high work complexity but low time complexity by combining it with another that has low work complexity. The main idea is to create sub-problems that are small enough that the first algorithm's work complexity does not have a limiting effect on the overall work complexity.

Let us see how this works for the merging problem. Here, we use the faster algorithm to merge $list1_k$ and $list2_k$, carefully selecting k. Note that algorithm Merge Method 3 merges two lists of size n each using $O(n\log\log n)$ work. If $list1_k$ and $list2_k$ have $\frac{n}{k}$ elements each, they can be merged using $O(\frac{n}{k}\log\log\frac{n}{k})$ work. This amounts to $O(n)$ if we choose $k = \log\log n$. In particular, it implies that if $list1$ and $list2$ have n elements each, $list1_k$ and $list2_k$ can be merged in $O(\log\log n)$ time using $O(n)$ work.

Given $Ranklist(list1_k, list2_k)$ and $Ranklist(list2_k, list1_k)$ computed recursively, we can now compute $Ranklist(list1_k, list2)$ and $Ranklist(list2_k, list1)$ also in $O(\log\log n)$ time using $O(n)$ work (k

remains $\log\log n$). Recall that if $Rank(list1_k[i], list2_k)$ is r, $list1_k[i] = list1[ik]$ lies between $list2_k[r-1]$ and $list2_k[r]$, *i.e.*, between $list2[(r-1)k]$ and $list2[rk]$. There are only $k-1$ elements between $list2[(r-1)k]$ and $list2[rk]$, and hence a single processor can locate $list1_k[i]$ in $O(k)$ steps. $\frac{n}{k}$ processors can, in parallel, compute the ranks of $\frac{n}{k}$ elements of $list1_k$ in $list2$. In parallel with these processors, $\frac{n}{k}$ processors can compute $Ranklist(list2_k, list1)$ in $O(k)$ time.

Similar to Merge Method 3, we now have $2\frac{n}{k}$ pairs of lists to merge, each with no more than k elements. With $k = \log\log n$, each of these mergers can be completed by a single processor in $O(\log\log n)$ time, requiring $O(n)$ total work.

Thus the total time complexity of the optimal merge algorithm is $O(\log\log n)$, and its work complexity is $O(n)$ on CREW PRAM. This is work-time optimal. Time complexity of any work-optimal PRAM algorithm to merge two sorted lists with n elements is $\Omega(\log\log n)$. In fact, the lower bound to merge sorted lists on an EREW PRAM is $\Omega(\log n)$. [3].

[3] T. Hayashi, K. Nakano, and S. Olariu. Work-time optimal k-merge algorithms on the pram. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):275–282, 1998

7.3 Accelerated Cascading: Find Minima

This section demonstrates a technique called *Accelerated Cascading*, which is designed to first reduce the depth of the computation tree, leading to an algorithm with lower time complexity at the cost of increased work complexity. That algorithm can then be combined with a work-efficient algorithm, which may have a slightly higher time complexity. It works by recursively partitioning the problem into sub-problems. It is a generalization of the binary tree computation structure and partitioning, except the number of sub-problems is not two (or a fixed number), but a function of the problem size itself. For example, one may partition a problem of size n equally into $\sqrt{n}$ sub-problems at each level.

We will use accelerated cascading to solve the problem of finding the minima of an unsorted list of values. (Assume these values are comparable to each other.) The regular binary-tree structure works well for this problem:

Listing 7.15: Parallel Minima

```
// Find the minima of list of n elements in O(log n) time
forall processor p in 0..(n-1)
  minima[p] = list[p]
for step in 1..log n
  forall processor p in 0..(n-1), p %(2^{step}) == 0
    minima[p] += minima[p+2^{step-1}]
```

Listing 7.15 requires $O(\log n)$ time and $O(n)$ work. Work complexity is optimal, as the best sequential algorithm is $O(n)$. Does there exist an algorithm with lower time complexity? Quite like the merge algorithms we discussed in the previous section, we can try to check for each element $list[i]$ if it is the minima. If constant time-common write is allowed (as in CRCW PRAM), $n-1$ processors can in parallel determine in $O(1)$ time if $list[i]$ is the minima.

Listing 7.16: Parallel Minima

```
// Find if list[i] is the minima
smallerthan[i] = false
forall processor p in 0..(n-1), p != i
  if(list[i] > list[p]) // Found a smaller element
    smallerthan[i] = true
```

smallerthan is a variable more than one processors may attempt to write simultaneously. All three versions of CRCW PRAM model support this operation – any writing processor writes `true`, and the algorithm is correct if any of the writes succeeds. Note that these writes cannot complete in $O(1)$ time in the EREW PRAM model, nor the BSP model.

To complete the algorithm in parallel for all indexes i, the list *smallerthan* contains for each i, whether any element of *list* is smaller than $list[i]$. This would take a total of $O(n^2)$ work, and $O(1)$ time. If $smallerthan[i]$ is `false`, $list[i]$ is the minima. The following listing produces this minima value in $O(1)$ time with $O(n)$ additional work.

Listing 7.17: Parallel Minima

```
// Produce the minima in smallerthan
forall processor p in 0..(n-1)
  if(smallerthan[i] == false)
    minima = index[i]
```

This algorithm, call it *Minima*0, works for Common-CRCW PRAM if the minima is unique, *i.e.*, only one element is strictly less than all others.

Let us try applying the technique discussed in the previous section to combine this non-optimal $O(n^2)$ work algorithm with the optimal $O(n)$ algorithm in Listing 7.15, to improve the time complexity. Recall that Merge Method 3 has a work complexity of $O(n \log\log n)$, slightly higher than the optimal work complexity $O(n)$. The non-optimal minima finding algorithm is $O(n^2)$, significantly higher than the optimal $O(n)$. To maintain the bound of $O(n)$ work, we would have to partition *list* into blocks of size $\sqrt{n}$. This two step algorithm would find the minima of each block first, in parallel with each other. This requires $O(n)$ work per block. With $\sqrt{n}$ blocks, the total work is $O(n\sqrt{n})$, with the time remaining $O(1)$. In the second step, the minima of the $\sqrt{n}$ block minima can be computed by repeating the same algorithm. The second step requires $O(1)$ time and performs $O(n)$ work. Let us call this algorithm *Minima*1.

*Minima*1 computes the minima of a list containing n elements in $O(1)$ time with $O(n^{1+\frac{1}{2}})$ work. *Minima*1 was derived by employing the more work-expensive algorithm on smaller blocks of the data. Can we reduce this further by re-applying the same idea? The answer is yes. We call this general technique *accelerated cascading*.

Suppose we employ *Minima*1 on blocks of size $\sqrt{n}$. The first step requires $n^{\frac{1}{2}}$ parallel invocations of *Minima*1 on blocks of size $n^{\frac{1}{2}}$ each. The second step finds the minima of the $n^{\frac{1}{2}}$ block-minima, again using *Minima*1. The resulting total work is $n^{\frac{1}{2}}n^{\frac{3}{4}} = n^{1+\frac{1}{4}}$. The time taken is that in two invocations of *Minima*1. This is the *Minima*2 algorithm.

This could go on. After k successive operations, the work complexity achieved is $n^{1+\frac{1}{2^k}}$. But, can this really go on indefinitely? Let us take a closer look. If we use *Minima*2 on the original problem, we require running $\sqrt{n}$ instances of *Minima*1 in parallel, each on a block of $\sqrt{n}$ elements. Each instance of *Minima*1, in turn, divides its $\sqrt{n}$ elements into $\sqrt{\sqrt{n}}$ blocks of size $\sqrt{\sqrt{n}}$ each. This looks like the binary tree algorithm structure, except the number of sub-problems created at each level is not a fixed 2. Rather, it is the square-root of the size of the problem at that level. Each of those sub-problem's size is also the square-root of the levels' problem size.

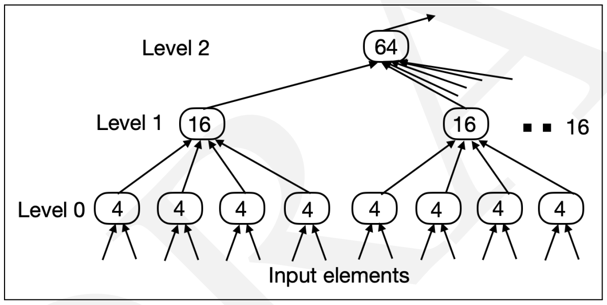


Figure 7.6: Find Minima with Accelerated Cascading. The numbers in the boxes are the number of processors used to compute the minima of the output of the level below, one element per arrow.

We continue this recursion until the problem size is less than 2 (or a higher constant). The number of levels in this recursion is $O(\log\log n)$. Since we can find minima on common-CRCW PRAM in $O(1)$ time, we know the $O(\log\log n)$ levels can each be computed in $O(1)$ given sufficient processors at each level. Note that there are $\frac{n}{2}$ computation nodes at step 0 (leaf level) and 1 node at the root. In general, there are $\frac{n}{2^{2^l}}$ computation nodes at level l, with 2^{2^l} elements processed per node. Given that (n^2) work is required to find the minima of n items, $(2^{2^l})^2$ work is required to find the minima of (2^{2^l})

items at each node on Common-CRCW PRAM. This adds up to $O(n)$ work at each level.

This leads to a total time complexity of $O(\log\log n)$ and a total work complexity of $O(n\log\log n)$. We will call this algorithm the fast minima method. Now that we have an algorithm requiring $O(n\log\log n)$ work, we can resort to the technique from the previous section.

Divide *list* into $\frac{n}{\log\log n}$ blocks with $\log\log n$ elements per block. We can compute the minima of each block in $O(\log\log n)$ time sequentially. One processor block add up to $O(\log\log n)$ work. We next apply the fast minima algorithm on the $\frac{n}{\log\log n}$ block minima, taking $O(\log\log n)$ time and $O(n)$ work. That is work-optimal and has a better time complexity than the algorithm with the basic binary tree structure.

7.4 *Recursive Doubling: List Ranking*

Solutions to List ranking in this section and Euler tour and Connected components in the next sections demonstrate the parallel algorithmic technique known variously as *Pointer Jumping* or *Recursive Doubling*. It is particularly useful for traversal of paths in lists and graphs.

Such traversal starts at a "root node" and follows pointers until a specific node, or the end of the path, is encountered. For example, to find connected components in a graph, one may perform a breadth-first or a depth-first search starting at some arbitrary node, labeling all reached nodes with the starting node's label. The main idea is to start exploring paths from all nodes in parallel, later *Short-circuiting* the paths that have already been explored.

Let us consider the linked-list ranking problem as a simple example. The linked-list ranking problem is to compute the link-rank of all nodes. The link-rank of a node is the number of links one must traverse to reach that node from the first node. The linked-list is not known to be sorted in any order. A sequential solution is as follows:

Listing 7.18: Sequential Linked-list ranking

```
// Find the rank of all nodes. headnode is the first node of a linked list
current = headnode
rank = 0
while(current != NULL) {
  current.rank = currentrank
  current = current.next
  currentrank = currentrank + 1
}
```

If the only way to access the nodes is by following *next* references starting at *headnode*, no parallelism is available. Instead, a parallel algorithm requires a different data structure, one that allows direct access to different nodes. Consider *nodelist*, a list of references to nodes in an arbitrary order, *i.e.*, $nodelist[i]$ for $i \in 0..n$ is a node in the liked list, and $nodelist[i].next$ does not necessarily refer to $nodelist[i+1]$. Now consider the following parallel algorithm:

Listing 7.19: Parallel Linked-list ranking

```
// Find the rank of all n nodes. headnode is the first node of a linked list
forall processor p in 1..(n-1)
  if(nodelist[p] == headnode)
    nodelist[p].rank[p] = 0
  else
    nodelist[p].rank[p] = 1
  skipnext[p] = nodelist[p].next
for step = 0..log(n-1)
  forall processor p in 0..n
    if(skipnext[p] != NULL)
      nodelist[skipnext[p]].rank = nodelist[skipnext[p]].rank + nodelist[p].rank
      skipnext[p] = skipnext[skipnext[p]]
```

skipnext is initially a copy of the next reference of each node. This copy is required because we later modify this reference to short-circuit certain nodes, and do not want to destroy the original linked-list. There are $\log n$ steps in the main loop of Listing 7.19. At each step, all processors update the currently estimated rank of its next node with reference to its own rank's current estimate. Then, each processor short-circuits its *next* reference by 'jumping' it to its next node's *next* reference.

Figure 7.7 demonstrates this jumping algorithm. At step i, the ranks of nodes with ranks $0..2^i - 1$ are known. (Try proving this by induction on i.) Thus, there are $\log n$ steps taken by each of the n processors. The time complexity is $O(\log n)$, and the work complexity is $(n \log n)$.

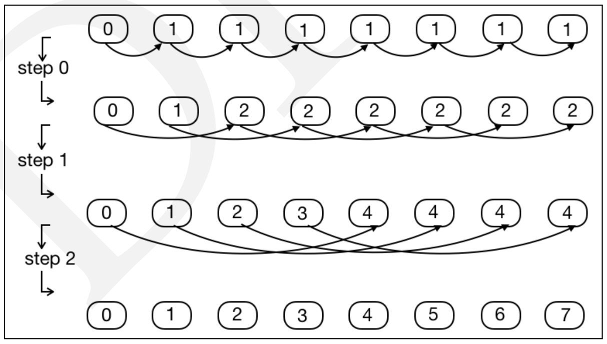


Figure 7.7: Parallel Linked-list ranking. *nextlist* is depicted by arrows, the values of *rank* are shown for every index after each step. These nodes are drawn in the order of the linked-list and not the order of the indexes.

7.5 *Recursive Doubling: Euler Tour*

List ranking and pointer jumping are useful in graph traversal as well. Often graph traversal is simply a way to reach all graph vertices or edges. A parallel graph representation allows processors direct access to any vertex or any edge. Breadth-first or Depth-first traversal is not necessary in such a context. In other settings, the path taken by graph traversal is meaningful. Such traversal may appear to be sequential by nature. However, they may not truly be sequential. For example, in breadth-first traversal, all children of a node may be traversed in parallel with each other.

In this section, we consider a depth-first traversal, particularly of a binary tree. This traversal is also called an *Euler tour* of the binary tree. It proceeds as follows:

Listing 7.20: Euler tour of a binary tree

```
// traverse a tree whose root is given
return if root == null
pre-visit(root)
traverse(left-subtree)
in-visit(root)
traverse(right-subtree)
post-visit(root)
```

The functions *pre-visit*, *in-visit*, and *post-visit* are application dependent. The order in which *pre-visit* is called on nodes is called the *pre-order*. Similarly, *in-visit*'s and *post-visit*'s orders are called *in-order* and *post-order*, respectively. This order is inherently sequential, but it does not have to be. For example, the position of node *i* in *in-order*, *i.e.*, its rank, may be computed in parallel with other node's ranks. We will determine the rank by turning the binary tree structure into a veritable list, which encodes the traversal order. As Figure

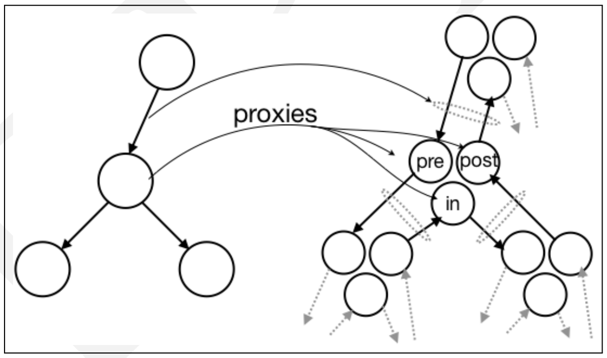


Figure 7.8: Treating a binary tree (left) as a list (right). Proxies for a node and and an edge are shown by the curved arrows.

7.8 demonstrates, we divide each internal node into three proxies, and each edge into two. The node proxies are labeled *pre*, *in*, and *post*, depending on their position in the traversal. This need not be a physical separation of the data structure, it is simply a logical view of the same nodes. If 'parent' references are maintained, those references can implicitly double as proxies. This transformation converts a tree into a list. The rank of (the proxy) nodes within this list is the Euler tour positions of each node. The time and work complexity are similar to that of list ranking – $O(\log n)$ and $O(n \log n)$, respectively.

If only, say, the *in-order* rank is required. The list ranking algorithm does not count the proxies marked *pre* and *post* by simply setting the initial values of rank to 0 for these nodes (see Listing 7.19).

7.6 Recursive Doubling: Connected Components

Let us next see how to use pointer jumping to derive a simple algorithm to find connected components in an undirected graph. (The basic idea also applies to directed graphs.) Let us assume that graph G is given as a list of edges *edgelist*, where i^{th} edge $edgelist[i]$ is a pair (u, v), where u and v are integers identifying two vertices, respectively. Let us call the number of edges m, and the number of vertices n.

The goal of the problem to find connected components is to assign a label $label[u]$ to each vertex u, which identifies its connected component. If vertex u has a path to vertex v, they are in the same connected component. If there is no such path, they are in different components. In other words, for all edges (u, v), $label[u] = label[v]$. Further, no such edge (u, v) may exist that $label[u]$ is different from $label[v]$. Vertices in different components have different labels. (Can you make labels be a component ID? See Exercise 7.19.)

Our first algorithm uses this property. It iteratively relabels each vertex until all the vertices of a connected component have the same label. For each edge (u,v), it assigns the same label to u and v. Of course, this must proceed in a controlled manner. Otherwise, labels may continue to change indefinitely. One way to control relabeling is to enforce asymmetry: a vertex's label is given to its neighbor only if that neighbor's label is larger.

Listing 7.21: Compute Connected Components I

```
forall processor p in 0..n
  label[p] = p
Repeat until any label changes
  forall processor p in 0..m
    (u,v) = edgelist[p]
    if(label[u] > label[v])
      label[u] = label[v]
```

Note that processors for two edges incident on a vertex u may both write two different values to $label[u]$ in the same step. An arbitrary-CRCW PRAM would allow any one of these writes to succeed. Listing 7.21 works under this model. It terminates with a correct labeling. The relabeling stops only when all edges have the same label on both its vertices. Since all vertices start with unique labels, and no edge exists between any pair of vertices in two different connected components, no such pair may have the same label. Also, if vertices u and v in the same component have different labels, it means at least two adjacent vertices on the path u to v have different labels. However, no two vertices connected by an edge are allowed to have different labels by the algorithm above.

How many steps are required before labels converge? At each edge where there is no convergence yet, the label of higher-labeled vertex reduces by one. The initially smallest labeled vertex of each component never changes its label. Call that vertex the *root* of the component. The root's label is taken by its immediate neighbors first and then by their neighbors until it diffuses through the entire component. This process might suggest that the root's label reaches the entire components in $O(\mathcal{P})$ steps , where $\mathcal{P}$ is the maximum length of the path from the root to any vertex in its component. This is not strictly true because root's label does not necessarily reach its neighbor u in one step, since another edge's processor may succeed writing its value in $label[u]$. Note, however, that the label can only reduce to a neighbor's label. Hence, in at most $d(u)$ steps, $label[u]$ changes to $label[root]$, where $d(u)$ is the degree of u. This is true for any vertex v along the path from the root. Hence, the total complexity is $O(\mathcal{P} + d(G))$, where $d(G)$ is the degree of the graph. Since all m processors may be active for all steps, the total work complexity is $O(m(\mathcal{P} + d(G)))$.

Ignoring the effect of the degree, the progress of label along different paths from the root appears to be similar to list ranking. It is reasonable to expect a similar pointer jumping would take time logarithmic in the length of the path. The difference here is that the graph is not a linear structure like a list, and we need to determine which way to jump. The labels we generate impose a direction to jump. Let the label-tree be formed by directed label-edge from vertex u to vertex $label[u]$. This is a forest in general, and in the beginning, each vertex is an isolated tree with the vertex's label set to itself.

In the next algorithm, processors are associated with graph edges. For simplicity, we associate processors for both edges (u, v) and (v, u). Each processor attempts to merge two adjacent label-trees corresponding to its associated edge (if certain conditions are met). A pair of label trees T_1 and T_2 are said to be adjacent if there is an edge (u, v) such that $u \in T_1$ and $v \in T_2$. In the label tree, we call vertex v the parent of vertex u if $label[u]$ is v. u is a root if $label[u]$ equals u; the root is its own parent. Further, a label-tree is called a star if all its vertices have the same label, that of its root. A star indicates a connected component. If no star is adjacent to any other, the algorithm terminates.

Listing 7.22: Compute Connected Components

```
1  // Set label(u) == label(v), iff u and v are in the same component. Graph has m edges, n vertices
2  forall processor p in 0..n
3    label[p] = p
4  forall processor p in 0..m
5    active[p] = true
6  while(there is an active processor) {
7    forall p in 0..m, active[p] == true
8      (u,v) = edgelist[p]
9      if inStar(u) and label[u] > label[v]  // See Listing 7.23
10       label[label[u]] = label[v] // Hook star's root to the smaller root
11     if inStar(u) and label[u] != label[v]
12       label[label[u]] = label[v] // Hook star's root to the other if not hooked on line 10
13     if not inStar(u)
14       label[u] = label[label[v]] // Pointer jumping
15     else
16       active[p] = false
17 }
```

The processor for edge (u, v) in the listing above first checks if u is a part of a star. If it is a part of a star, it attempts to hook its parent, *i.e.*, its root, to the parent of neighbor v. v need not be in a star. The processor accomplishes hooking by relabeling u's parent to that of v's. This makes u's root the child of v's parent. Of course, if v is also a part of a star, processor (w, u) may simultaneously hook v's root to u's root if v and w belong to the same star. Asymmetry must be imposed to prevent a cycle, just as in Listing 7.21. Hence, u hooks its root to v's root only if v's label is smaller than u's label.

The listing above ensures that all stars that can be hooked to another tree are indeed hooked, even if they fail to hook on line 9 due to *symmetry breaking*. The reason a star may fail to hook on line 9 is that all its adjacent stars may have higher value roots. If a star $S1$ remains a star at line 11, it may hook to any adjacent tree. That adjacent tree must not be a star at 11, because any star that is hooked ceases to be a star. Further, any star $S2$ adjacent to $S1$ at line 9 could not remain unhooked on line 11 because $S2$ did have at least one adjacent star with a smaller root: $S1$.

If a star has no adjacent tree, it does not hook. In that case, that star is one of the graph's final connected components. See Figure

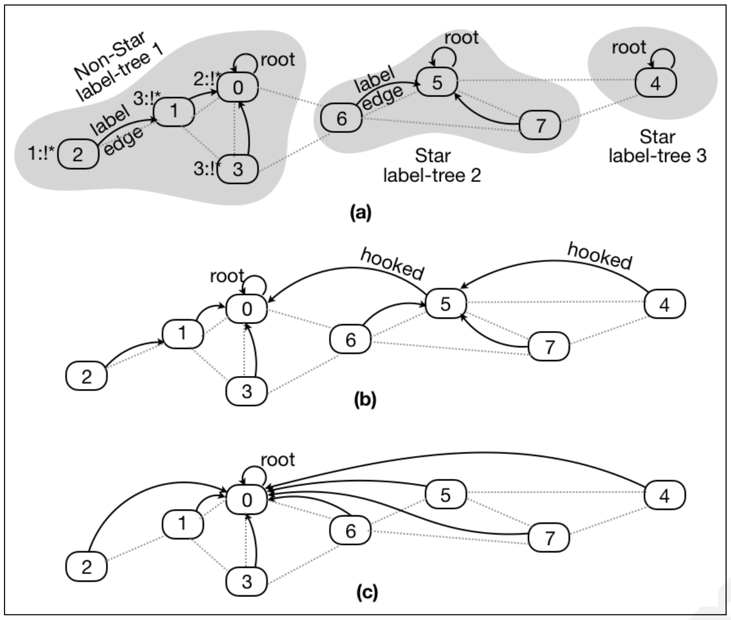


Figure 7.9: Connected component computation. The number within the oval is the vertex identifier. Dark arcs indicate the labels. For example, *label*[2] is 1 in part (a). This part shows three label trees, of which the first one is not a star, and the other two are stars. Graph edges are shown in lighter dashed lines. Part (b) shows the hooking of two stars. Part (c) shows one step of pointer jumping, which yields a single connected component with vertex 0 as the root.

7.9 for an illustration. The vertex identifiers are shown in the ovals. Figure 7.9(a) shows the state of the algorithm at some step, when there are three label-trees. The left-most tree is not a star, and the other two are. Edges connect tree 2 to both tree 1 and tree 3; it is adjacent to both trees. The processor associated with edge $(7, 4)$ attempts to hook tree 2 to tree 3, while the one associated with edge $(6, 3)$ attempts to hook it to tree 1. In arbitrary-CRCW PRAM, one of the writes succeeds on line 9. Let's say the second one. On that line, tree 3 does not hook to tree 2 because tree 3 already has a smaller label. Instead, it hooks to tree 2 on line 11. Note that before that line, tree 2 becomes a part of tree 1. There is a single tree remaining after the two hooks, and it is not a star. A single step of pointer jumping on line 14 turns the tree into a star, which is the final connected component. The algorithm terminates in the next iteration.

How does the algorithm terminate, though? Since common write is allowed, processors may set a shared variable *anyactive*[4] to `false` at the beginning of every iteration on line 6. Every active processor then sets *anyactive* to `true` at the end of the iteration. If no processor sets *anyactive*, it remains `false`, and all processors terminate. The other step that is not detailed in Listing 7.22 is how to determine if vertex u is a part of a star. This step is described below.

[4] We dispense with the $ suffix for shared variables in this chapter; variables are shared by default in PRAM

Listing 7.23: Compute if vertex w is a part of a star

```
// inStar: True if all vertices in u's current component have the same label.
star[u] = true // All vertices are in lock-step
if label[u] != label[label[u]] // u's parent is not a root
```

```
4    star[u] = false
5    star[label[label[u]]] = false // u's grandparent is also not a star
6  star[u] = star[label[u]]  // If its parent was marked non-star, u is not star
```

We seek to determine in parallel for every node if it is a part of a star. A root that has any grandchild is not the root of a star. Hence, vertex u whose parent is not the root (a root's label is its own identifier) is a proof that the tree it belongs to is not a star. Such a vertex marks the grandparent as not a star. At the same time, root's child-less children may remain marked stars. However, if the root has even a single grand-child, it is marked non-star on line 5. This is the evidence for child-less children of the root to be marked star on line 6. Figure 7.9(a) demonstrates this. The processor for edge, say, $(2, 1)$ marks node 2 as non-star first on line 4. This processor next marks the grandparent of node 2, *i.e.*, node 0, non-star on line 5. Finally, node 3 is marked non-star by the processor for edge $(3, 0)$ on line 6.

The algorithm terminates in time $O(\log \mathcal{P})$. The distance of each node in a non-star tree to its root halves in each iteration (except that of the root and its children). Once the tree becomes a star, it must hook to another tree with a new root, and the distances continue to halve. Since all m processor may remain active until the end, the work complexity is $O(m \log \mathcal{P})$.

Sometimes it may be necessary to count the number of connected components and assign contiguous identifiers instead of a root's label. This can be easily achieved by using a prefix-sum in $O(\log n)$ time using $O(n)$ work.

Listing 7.24: Relabel Connected components contiguously

```
// Assign in id[p] the connected component identifier for each node p
forall p in 0..(n-1)
  if(label[p] == p)
    id[p] = 1
  else
    id[p] = 0
exclusive prefix-sum (id)
forall p in 0..(n-1)
  id[p] = id[label[p]]
```

7.7 Pipelining: Merge-sort

We will next discuss several sorting algorithms, each of them designed to demonstrate an algorithmic technique. Parallel merge-sort derives directly from parallel merge discussed earlier. Parallel radix-sort is efficient for sorting integer-based elements. Parallel quick-sort and sample-sort are also efficient for general cases.

We begin with an application of Pipelining in Merge-Sort.

Basic Merge-Sort

Recall from Section 7.2 that merge can be completed in $O(\log\log n)$ time with $O(n)$ work on CREW PRAM. Merge-sorting a list of n comparable elements begins by merging $\frac{n}{2}$ pairs of singletons, followed by $\frac{n}{4}$ pairs of 2-element lists, and so on until the last step merges 1 pair of $\frac{n}{2}$-element lists. It proceeds as follows:

Listing 7.25: Relabel Connected components contiguously

```
1 // Sequentially Merge-sort a list with n element
2 for step = 0 to ceil(log(n))-1  // Assume n is a power of 2
3   numpair = $2^{\log(n)-step-1}$
4   listlen = $2^{step}$ // Adjust last pair's len if n is not a power of 2
5   for pair = 0 to numpair-1
6     p0 = pair*2*listlen
7     merge list[p0..p0+listlen-1], list[p0+listlen..p0+2*listlen-1]
```

In this listing, the loop on line 5 can clearly be parallelized. This is demonstrated in Figure 7.10, where steps proceed upward from the bottom. Step i computes level $i+1$ from level i. Level 0 comprises one element per list. Merging different pairs at each step can proceed independently of each other. If we assume $\frac{n}{2}$ processors for the entire sort, these processors can all be allocated to one step at a time. At step i, $\frac{n}{2^{i+1}}$ pairs of lists are merged. The two lists in each pair at level i have 2^i elements each. Using 2^i processors per pair-merge adds up to $\frac{n}{2}$ total processors at each step.

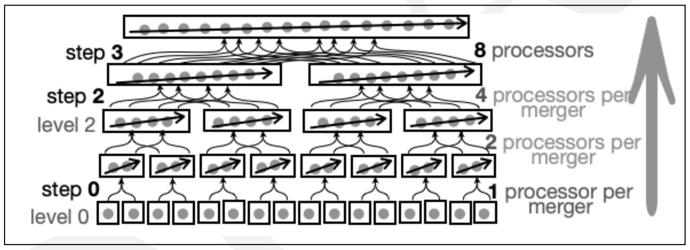


Figure 7.10: Merge-tree for Merge-sort. Steps proceed bottom-up. The number of merges halves at each level, the sizes of lists to merge (and produced) double at each level. The number of processors per merge operation also doubles at each step, ensuring that the total number of processors is constant across steps.

On the other hand, it appears that the steps of the loop on line 2 must proceed sequentially. After all, the lists at level i are not available until step $i-1$ is complete. That would imply that $\log n$ sequential iterations of line 2 are required. Step i requires $O(\log i+1)$ time and $O(n)$ work overall. This results in a total time complexity of $O(\log n \log\log n)$ and work complexity of $O(n\log n)$ to merge-sort on CREW PRAM. The time complexity on EREW PRAM is $O(\log^2 n)$. This is a work-optimal algorithm. Is it also time-optimal? Not so, it turns out. We introduce the algorithmic technique called *Pipelin-*

ing that improves the parallelism, and hence the time complexity, without increasing the work complexity.

Pipelining amounts to incrementally performing otherwise sequential steps by decomposing them into sub-steps. Sub-steps can be performed on a part of the input, without waiting for the entire input. Such pipeline is possible if the steps also produce the output incrementally, a part at a time. Merging algorithms described in Section 7.2 satisfy this general requirement. Recall, for example, that Merge Method 2 merges two sorted lists by first ranking the even elements and then using the results to rank the odd elements. Thus it does not need the values of the odd elements at the start. Consequently, it produces the rank of the even elements first, and then the ranks of the odd elements later. Further, the entire algorithm is recursively applied, as illustrated in Figure 7.11 (for level 3 of Figure 7.10). The part of the list that is processed exponentially evolves to the entire list.

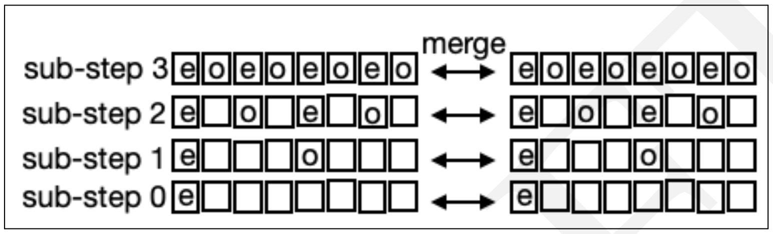


Figure 7.11: Active sublists at each merger sub-step are marked. The even positioned elements of the active sublist are marked e, and the odd positioned ones are o. The odd ones are the newly activated elements at that sub-step. Thus, the number of active elements doubles at each sub-step. First, the ranks of the already active members are updated with respect to the other updated sublist. Then, the ranks of the newly active members are computed (see Listing 7.11).

In general, since the lists merged at level i comprise 2^i elements each, there are $\log 2^i$ sub-steps required at that level. At sub-step 0, only one element of each list is active: the $0th$ element. In sub-step 1, the first and the middle elements are active. They are, respectively, the even and the odd elements at that sub-step. For each merger of two lists, a sub-step computes the ranks of each list's active elements with respect to the active elements of the other list. In sub-step j, 2^j elements are active, doubling with each sub-step. The rank of all active elements at sub-step j can be computed in $O(1)$ from the ranks computed in sub-step $j-1$ (see Merge Method 2).

That opens up the possibility of pipelining sub-steps. The newly active members required at sub-step j of level i may be produced by level $i-1$ any time before sub-step j of level i, and not necessarily before its sub-step 0. The pipeline would be perfect if those active members are produced by level $i-1$ at sub-step $j-1$. However, this is not quite how Merge Method 2 proceeds. It does not guarantee, for example, that the middle element of the merged list it produces for the level above is known after sub-step 1 (or, any early sub-step). On the other hand, it expects the middle elements of its two input lists to be available at its sub-step 1.

An adjustment to the merge algorithm is necessary to pipeline it. In particular, we relax the strict requirement of the order in which the elements of the merged list must be produced, and rather use them at the next level in the order they are actually produced. The main question we need to answer is: what partial results to produce at each sub-step and how to use them at the next level. The answer will also indicate when level $i+1$ can begin after level i begins, and when it completes. Also note that pipelining implies that multiple levels have active computation simultaneously. Work complexity analysis and processor allocation must account for this. In the non-pipelined version of the merge-sort algorithm, only one level is active at a time, possibly simplifying processor allocation.

Pipelined Merges

We discuss the pipelined merge-sort algorithm[5] next. For simplicity, we will assume that n, the number of elements to sort, is a power of 2. The algorithm works as well for non-power of 2 cases. The algorithm itself is only slightly more complex than the one in Listing 7.25, but its analysis is somewhat more cumbersome.

[5] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4): 770–785, 1988b

Let us agree on a terminology first. Sub-steps are in the context of each merge-sort step. We will use the term *tick* to denote an algorithm-wide sub-step numbering. Each node of the merge-tree maintains an evolving list. This list is initially empty at all nodes except the leaves, and completes when it contains all elements in the subtree of that node (see Figure 7.10). At each tick, each active node incrementally merges some of the data produced earlier by its children into its evolving list. A node becomes active and starts merging two ticks after elements start to appear in its children's lists. Three ticks after the children complete their lists, the parent also completes its merger and deactivates. The activation and deactivation happen level by level, from the leaf level to the root.

Thus level i activates at tick $2i+1$, and completes at tick $3i$. This means that 3 ticks after level i is complete, level $i+1$ also completes. Thus there are $O(\log n)$ ticks. We will see that the algorithm takes a constant time per tick. The incremental merger is similar to Merge Method 2, except a more general sublist is processed at each tick. We will refer to the final list produced by node x by $L(x)$. The two children of a node at level i are nodes at level $i-1$. We call these children $x.left$ and $x.right$, respectively.

In a slight abuse of notation, we will use the level i in place of node, *i.e.*, $L(i), L(i.left)$, or $L(i.right)$. Since the operation at all nodes of a level is similar, our algorithmic description does not distinguish between them. Read $L(i)$ as the list produced by a generic node at level i. We do need to refer to a node's children; hence, $i.left$ and $i.right$. We also need to account for the evolution of the list at each node. We will denote the current list at any node at level i at tick t – meaning just before tick t – by $L(i,t)$. $L(i,t+1)$ is computed at tick t by merging $L(i.left,t)$, and $L(i.right,t)$ with the help of $L(i,t)$. $L(i,t)$ = null for $t < 2i$ and $L(i,t) = L(i)$ for $t \geq 3i$. $L(0) = L(0,0)$ is the initial list at level 0. Each leaf node contains one element from the full list to be sorted (see Figure 7.10).

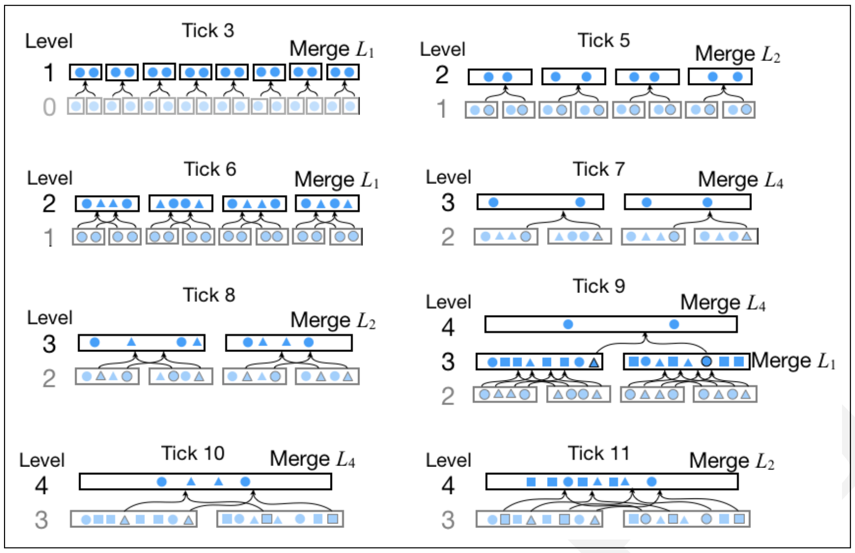


Figure 7.12: Pipelined merge: ticks 3 to 11. Ticks 0-2 and 4 have no mergers and are skipped. The state of the active levels are shown. Inactive levels whose lists are used by active levels are also shown, but greyed out. The sublisting of the children is indicated with L_1, L_2, or L_4. Children's elements selected in the sublists are shown with a dark outline. The elements added to a level at successive ticks are shown as circles, triangles, and squares, respectively. At tick 9, both levels 3 and 4 are active. Note that the state of a level's lists is shown at the end of each tick. For example, level 4 processors at tick 9 merge $L_4(3,8)$, the sublists of the children's lists produced at tick 8, and shown in the tick 8 figure.

At tick t, the 2^{i-1} processors assigned to a node at an active level i compute

$$L(i,t+1) = Merge(L_k(i.left,t), L_k(i.right,t)) \text{ using } L(i,t) \qquad (7.1)$$

where L_k is the sublist of L, taking every k^{th} element starting at $L[k-1]$. The value of k is 4 during the evolution of the children's lists. Once the children's lists complete, k is set to 4, 2, then 1 at the next three ticks. After that third tick, the parent's list is complete, since it produces a merger of the complete lists of its two children. Note that setting k to 1 means that all elements of the children are included in L_k. Specifically, in equation 7.1:

$$\begin{aligned} k &= 4 \text{ if } t < 3i-1 \\ k &= 2 \text{ if } t = 3i-1 \\ k &= 1 \text{ if } t > 3i-1 \end{aligned} \qquad (7.2)$$

The progression of ticks is demonstrated in Figure 7.12.

The important step of the actual merger at each tick at each level remains to be discussed. The Merge on equation 7.1 is completed in $O(1)$ time using $O(j)$ work, where j is the number of elements in $L(i,t)$. This is due to the choice of L_k: old elements at a level are well interspersed among the new elements, even if they do not faithfully alternate like odd and even. We need another digression to explain this formally.

Aside: *c*-cover Merging

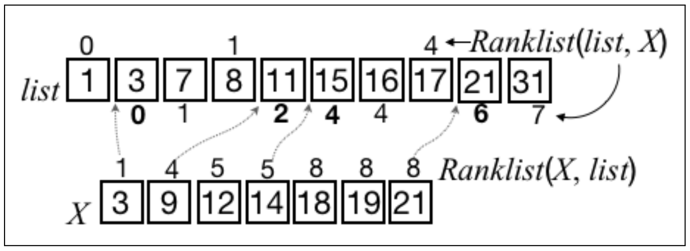
A sorted list L' is called a c-cover of list L if two consecutive elements e_1 and e_2 of L' have at most c elements of L between them. In other words, there are c or fewer elements e in L such that $e_1 < e < e_2$. If e_2 is the first element of L', we consider $e_1 = -\infty$, and if e_1 is the last element, we let $e_2 = \infty$.

Two sorted lists $list1$ and $list2$ with n elements each may be merged in $O(1)$ time using $O(n)$ work if $Ranklist(X, list1)$ and $Ranklist(X, list2)$ are given and X is a 4-cover for both $list1$ and $list2$. We first "invert" X's ranks by computing $Ranklist(list1, X)$ and $Ranklist(list2, X)$. Listing 7.26 shows how to compute $Ranklist(list, X)$. given $Ranklist(X, list)$. Since we know $r = Rank(X[i], list)$ for elements of X, we also know $Rank(list[r], X)$. We only need to derive the ranks of the other elements near $list[r]$ in $list$. An example is shown in Figure 7.13, explaining the steps of Listing 7.26.

Listing 7.26: *c*-cover merge

```
// Compute rankx = Ranklist(list, X) given xrank = Ranklist(X, list),
forall processor p in 0..(n-1) // n elements in list
  rankx[p] = -1 // Unfilled value
forall processor p in 0..(|X|-1) // |X| is the number of elements in X
  if(i == 0 or xrank[i-1] != xrank[i])
    if (list[xrank[p]] == X[p]) // Do not include X[p] in rank of list[xrank[p]]
      rankx[xrank[p]] = p
    else // Included X[p] is rank of list[xrank[p]]
      rankx[xrank[p]] = p+1
  if (p < n-1 and xrank[p+1] > xrank[p]+1) // Also rank the next element of list
    rankx[xrank[p]+1] = p+1
forall processor p in 0..(n-1)
  if (rankx[p] == -1) // Not filled yet
    for i = p-1 to p-3 // Look up to 3 steps to the left for the first filled rank
      if(xrank[i] != -1)
        rankx[p] = rankx[i]
        exitloop
    if(i < 0) // No filled rank found (xrank[-1] is effectively 0)
      xrank[p] = 0;
```

Like our prior assumption, the elements within each list remain unique. However, X may have some elements that appear in $list$ and others that don't. Since we define $r = Rank(x, list)$ as the number of elements in $list$ that are strictly less than x, we must differentiate between these two cases. If $x == list[r]$, all elements to the left of x in X are less than $list[r]$, *e.g.*, $X[6] = 21$ in Figure 7.13. On the other hand, the rank of $X[1] = 9$ is 4. $list[4]$ is not equal to 9. This means that all the elements to the left of 9 in X, as well as 9, are less than $list[4]$. These cases are differentiated on line 6 of the listing. In either

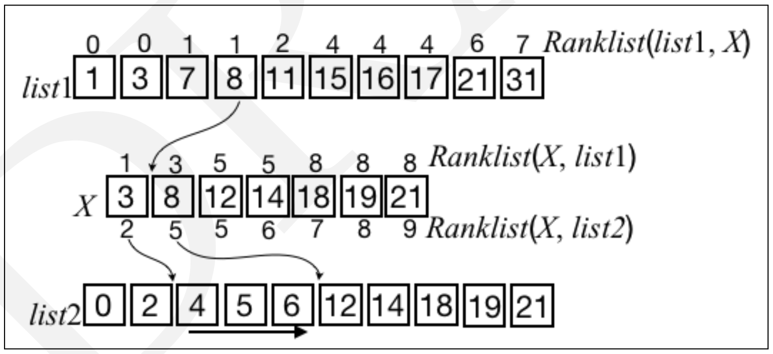


0 1 4 ←Ranklist(list, X)
list 1 3 7 8 11 15 16 17 21 31
0 1 2 4 4 6 7
1 4 5 5 8 8 8 Ranklist(X, list)
X 3 9 12 14 18 19 21

Figure 7.13: Compute $Ranklist(list, X)$ given $Ranklist(X, list)$ in $O(c)$ time, if X is a c-cover of $list$. The example shows integer-valued lists X and $list$. The values $Ranklist(X, list)$ are shown above X. These ranks indicate the position of the corresponding elements of X in $list$. The elements at those positions of $list$ are ranked in X at line 7 or 9 of Listing 7.26. These ranks are shown below $list$ in bold font. The ranks computed on line 11 are shown in light font. Finally, the ranks of the remaining elements of $list$ are computed at line 16 and are shown above $list$.

case, $list[r+1]$ is greater than x. $list[r+1]$ is also less than the element to the right of x in X, as long as that element has a rank different from that of x. Hence, $Rank(list[r+1], X)$ is one more than the index of x in X. Remember that two consecutive elements of X may have the same rank, for they have 0 elements of $list$ between them. For example, the ranks of $X[2]$ and $X[3]$ are both 5. The listing lets the processor associated with the last of these equal elements of X to set the inverse rank (on line 5).

The processor associated with x also sets the ranks of $list[r]$ and $list[r+1]$ on line 11. All the elements to the right of $list[r+1]$ that do not get a reverse rank in the previous step also have ranks equal to that of $list[r+1]$. For example, $r = 1$ for $X[0]$. $list[r+1] = list[2]$ has rank 1 (one more than the index of $X[0]$). The rank of the element to the right of $X[0]$ has rank $s = 4$ in $list$, meaning $list[s-1] = list[3]$ is definitely less than $X[1]$. However, all such elements to the right of $list[r+1]$ until index $s-1$ are greater than $X[0]$. Hence, they all must have the same rank as that of $list[r+1]$. This is completed on line 16 in Listing 7.26. This loop iterates at most c steps if X is a c-cover of $list$, as $s - r \le c$.



0 0 1 1 2 4 4 4 6 7 Ranklist(list1, X)
list1 1 3 7 8 11 15 16 17 21 31
1 3 5 5 8 8 8 Ranklist(X, list1)
X 3 8 12 14 18 19 21
2 5 5 6 7 8 9 Ranklist(X, list2)
list2 0 2 4 5 6 12 14 18 19 21

Figure 7.14: Compute $Ranklist(list1, list2)$ given $Ranklist(list1, X)$ and $Ranklist(X, list2)$ in $O(c)$ time, if X is a c-cover of $list2$. The example shows integer-valued lists $list1, X$ and $list2$. The values $Ranklist(X, list1)$ are shown above X, and $Ranklist(X, list2)$ below it. $Ranklist(list1, X)$ is shown above $list1$. $Rank(list[3], X)$ is 1. Given that $Rank(X[0], list2)$ is 2 and $Rank(X[1], list2)$ is 5, $2 \le Rank(list1[3], list2) < 5$. Hence, $Rank(list1[4], list2)$ is 2+ the number of elements of $list2[2..4]$ that are less than $list1[3]$. There are at most c such elements, in general.

$Ranklist(list1, X)$ and $Ranklist(list2, X)$ can be computed in parallel with each other. They both take $O(1)$ time and perform $O(n)$ for a constant c is a constant. Recall that we are interested in computing

$Ranklist(list1, list2)$ and $Ranklist(list2, list1)$. If the rank of element i of $list1$, $Rank(list1[i], X)$ is $xrank1$, $X[xrank1 - 1] < list1[i]$, and $X[xrank1] \geq list1[i]$. All elements in $list2$ less than $X[xrank1 - 1]$ are definitely less than $list1[i]$. There are $Rank(X[xrank1[i - 1]], list2)$ of those. Additionally, some elements between the positions $Rank(X[xrank1[i - 1]], list2)$ and $Rank(X[xrank1[i]], list2)$ in $list2$ may also be less than $list[i]$. See Figure 7.14 for an example. $xrank1 = Rank(list1[3], X)$ is 1. $rankx2a = Rank(X[xrank1], list2)$ is 2. This means that the first 2 elements of $list2$ are all less than $list1[3]$. $rankx2b = Rank(X[rank1 + 1], list2) = 5$ implies that $list2[rankx2b] \geq list1[4]$. Some elements between index $rankx2a$ and $rankx2b$ may be smaller than $list1[3]$, but there may be at most c such elements; 3 in this example.

Hence, one processor allocated to position 3 of $list1$ may compute its rank in $O(c)$ time. If c is a constant, the work complexity is $O(n)$. This can be done in EREW PRAM with a bit of care. More than one element of $list1$, e.g., $list1[2]$ and $list1[3]$, may have the same rank in X: $xrank1 = 1$. The processors assigned to find their respective ranks in $list2$, both read $Rank(X[1], list2)$. Since there are at most c such elements in $list1$, they may be serialized to eliminate any need for concurrent read. This would require the processor assigned to every position to determine its serial order. One way is to count the number of elements to the left of its position in $list1$ that have the same rank as its own. We skip those details here. The following is a simpler CREW PRAM version.

Listing 7.27: Transitive c-cover merge

```
// Compute rank1 = Ranklist(list1, list2)
//   given rankx1 = Ranklist(list1,X), xrank2 = Ranklist(X, list2),
forall processor p in 0..(n-1) // n elements in list1
  i = xrank2[rankx1[p]-1]
  while(i < min(n, xrank2[rankx1[p]]))
    i = i+1
  rank1[p] = i
```

The listing above indicates that these ranks can be computed transitively: $Ranklist(L1, L3)$ can be computed from $Ranklist(L1, L2)$ and $Ranklist(L2, L3)$. Further, if $L1$ and $L2$ have a c-cover relationship, and so do $L2$ and $L3$, $Ranklist(L1, L3)$ can be computed in constant time with linear work. We say that two lists have a c-cover relationship if one of them is a c-cover of the other.

4-cover Property Analysis

We can now discuss the pipelined merger of equation 7.1. The goal is to find and use c-covers for each merger. We will see that the list in a node at the end of a tick is a 4-cover for the merger at its next tick.

$\diamond 1$: $L_k(i, t-1)$ is a 4-cover of $L_k(i, t)$,
and $Ranklist(L_k(i, t-1), L_k(i, t))$ is known before tick t.

$\diamond 2$: $L(i, t)$ is 4-cover of $L_k(i.left)$,
$L(i, t)$ is 4-cover of $L_k(i.right, t)$,
and $Ranklist(L(i, t), L_k(i.left, t))$ and
$Ranklist(L(i, t), L_k(i.left, t))$ are known before tick t.

Property $\diamond 2$ ensures that $L_k(i.left, t)$ and $L_k(i.right, t)$ can be merged in $O(1)$ time with $O(|L(i, t+1)|)$ work at tick t using 4-cover merger. We need property $\diamond 1$ to find the Ranklists of $\diamond 2$. We will prove the following property, of which Properties $\diamond 1$ and $\diamond 1$ are corollaries:

$\diamond *$: If $L_4(i, t)$ has m elements in some range $l..r$, $L_4(i, t+1)$ has no more than $2m$ in that range. This would imply that if a and b are two consecutive elements of $L_4(i, t)$, meaning it has 2 elements in the range $a..b$, $L_4(i, t+1)$ has no more than 4. That would guarantee that $L_4(i, t)$ is 4-cover of $L_4(i, t+1)$ for any i and t.

The statement above can be proven by induction on i. Note that once the lists at a node's children are complete, the next three ticks at the parent are similar to three steps of Merge Method 2. Every $2^2 th$, then every $2^1 th$, and finally every $2^0 th$ element of the children's lists are merged. $\diamond *$ holds trivially in these cases. The following proof, hence, focusses on the ticks when the children lists are not yet complete when sublists are formed by every $4th$ element. We refer to the elements selected in the sublist for merger as samples. The non-selected elements will be called non-samples.

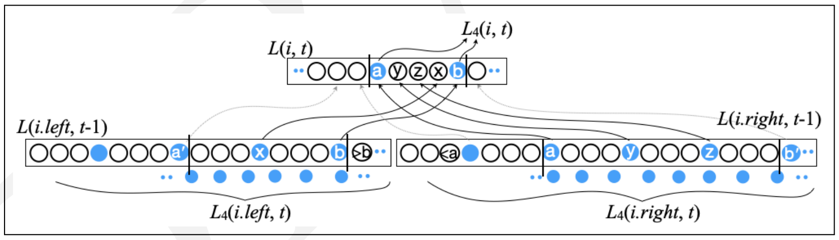


Figure 7.15: $L_4(i, t+1)$ is 4-cover of $L_4(i, t)$. Node i and its children are shown. Filled circles depict samples. Non-filled ones are non-samples. Elements a and b are two consecutive elements in $L_4(i, t)$. Hence, 5 elements of $L(i, t)$ are in the range $a..b$. Some of these 5 come from the left and the others from the right child's sublist. In these sublists, any sample in the range $a..b$ must be within the vertical bars. In particular, a' is definitely less than a, and b' is definitely greater than b. There are 3 samples in the left child in $a'..b'$, and 4 in the right. If no more than 6 and 8 elements appear after tick t in children's sublists, $L(i, t+1)$ may not contain more than 14 elements in $a..b$. Hence, at most 4 can be in $L_4(i, t+1)$.

By induction, if the statement holds at the children before tick t, it also holds at the parent after tick t (*i.e.*, before tick $t+1$). The base case is easy to prove, as it holds trivially at all levels whose children are complete. Figure 7.15 illustrates the inductive step of the proof. The figure takes two *consecutive* elements of $L_4(i, t)$: a and b. (The same argument holds for non-consecutive elements.)

Consider a node at level i. By the inductive hypothesis, $L_4(i-1,t-1)$ has α elements in the range $a..b$, $L_4(i-1,t)$ has no more than 2α in that range.

Since $L_4(i,t)$ has every fourth element of $L(i,t)$, if $L_4(i,t)$ has m elements in the range $a..b$, $L(i,t)$ has $4m-3$ elements in that range. Some of these $4m-3$ come from the left child's sublist and the others from the right child's. Vertical bars in Figure 7.15 delineate the range $a..b$ in the children. There are fewer than $4m-1$ elements in the two children's sublists in the range $a..b$, we include two additional samples that bound the non-samples that may lie in the range $a..b$. Let s_1 samples come from the left sublist $L_4(i.left,t-1)$, and s_2 from $L_4(i.right,t-1)$. $s_1+s_2 \leq 4m-1$. By the inductive hypothesis, no more than $2s_1$ samples exist in $L(i.left,t)$ in the range $a..b$, and no more than $2s_2$ samples in $L(i.right,t)$. Since $L(i,t+1)$ is formed by merging $L_4(i.left,t)$ and $L_4(i.right,t)$, it cannot contain more than $2(s_1+s_2)=8m-2$ elements in the range $a..b$. This guarantees that its sublist $L_4(i,t+1)$ may not contain more than $2m$ elements in the range $a..b$, and $L_4(i,t+1)$ is a 4-cover of $L_4(i,t+1)$.

Thus, we can say that property $\diamond 1$ holds, and $L_k(i,t)$ is a 4-cover of $L_k(i,t+1)$ for all k in the pipelined merge progression. Since this applies to all levels, clearly $L_k(i.left,t-1)$ is a 4-cover of $L_k(i.left,t)$.

Further, considering that $L(i,t)$ was formed at tick $t-1$ by merging $L_k(i.left,t-1)$ and $L_k(i.right,t-1)$, we know that between any pair of consecutive elements of $L(i,t)$ there are no more than two elements of $L_k(i.left,t-1)$ and no more than two elements of $L_k(i.right,t-1)$. For example, consider consecutive elements z and x in $L(i,t)$ in Figure 7.15. There cannot be more than 2 elements in $L_k(i.left,t-1)$ in the range $z..x$. Were z and x both in $L_k(i.left,t-1)$, they would be its consecutive elements. If neither were in $L_k(i.left,t-1)$, it must not contain any element in the range $z..x$. If only one of z and x is in $L_k(i.left,t-1)$, it must be the only element in $z..x$. For example, in Figure 7.15, x is in $L_k(i.left,t-1)$, and z is not. In that case, the element immediately before x in $L_k(i.left,t-1)$, a' in this case, is definitely less than z, ensuring that there is only one element in $z..x$. Given, then, that the $L_k(i.left,t-1)$ contains no more than 2 elements in $z..x$, by property $\diamond *$, $L_k(i.left,t)$ contains no more than 4 in that range. Hence, $L(i,t)$ is a 4-cover of $L_k(i.left,t)$. Similarly, $L(i,t)$ is a 4-cover of $L_k(i.right,t)$. That proves property $\diamond 2$.

Merge Operation per Tick

At node i, at tick t, we merge $L_k(i.left,t)$ and $L_k(i.right,t)$. We know $L(i,t)$ is a 4-cover for both. If we have $Ranklist(L(i,t), L_k(i.left,t))$ and $Ranklist(L(i,t), L_k(right,t))$, we can directly invoke the c-cover merge algorithm described earlier. $Ranklist(L(i,t), L_k(i.left, t))$ can be transitively computed from $Ranklist(L(i,t), L_k(i.left, t-1))$ and $Ranklist(L_k(i.left, t-1), L_k(i.left), t)$ in constant time with linear work, as described next. $Ranklist(L(i,t), L_k(i.right, t))$ can be similarly computed.

If $L(i,t)$ is `null`, it means there were no elements in $L_k(i.left, t-1)$ and $L_k(i.left, t-1)$, and this is the first time a child's list contains k elements. Since $k \leq 4$, this merger can be done at each such node in $O(1)$ time with $O(1)$ work. In the general case, when $L(i,t)$ does exist, this means it was a merger of sublists $L_k(i.left, t-1)$ and $L_k(i.right, t-1)$, using $X = L(i, t-1)$ as 4-cover. This means we computed $Ranklist(L(i,t), L_k(i.left, t-1))$ and $Ranklist(L(i,t), L_k(i.right, t-1))$. We store them in lists *lrank*, and *rrank*, respectively, at node i. Also, since we use $L(i, t-1)$ as a cover for computing $L(i,t)$, we compute $Ranklist(L(i, t-1), L(i,t))$ (see c-cover merger algorithm earlier). This can provide, in $O(1)$ time with $O(L_k(i,t))$ work, ranks for their 4-covers: $Ranklist(L_k(i, t-1), L_k(i,t))$. We store them in *erank* at node i.

In Listing 7.28, $left.erank$ refers to the list *erank* of the left child and *right.erank* to that of the right child. Note that unlike Merge Method 2, the pipelined mergers cannot happen in-place. Multiple rank arrays are required. Some consolidation is possible, and the pipelined merger can be implemented on EREW PRAM. 4-cover merge must be implemented in a way that the *lrank*, *rrank*, and *erank* are read by all active processors first and then updated at the end. We skip a detailed presentation of that.

Listing 7.28: Pipe-lined c-cover merge

```
// Merge Lk(x.left), Lk4(x.right)), given lrank, rrank and erank
forall node x at level 2*tick+1..3*tick
  k = computek(level, tick) // See Eq 7.2
  Using |L(x.left)/k| processors // |L(x.right)|==|L(x.right)|, assume
    4-cover-merge(Lk(x.left), Lk(x.right, t), \
          lrank, rrank, x.left.erank, x.right.erank)
    Update rankl, rankr, erank
```

Many mergers happen in parallel in the pipelined scheme, each completing in $O(1)$ time. The nodes at level i of the merge-tree complete their processing at tick $3i$. This implies that all levels have completed after $3 \log n$ ticks, and the root node has the sorted result. The work at each level is also easy to count. Note that a level deactivates when complete. A level is complete when it receives all the elements of the list to be sorted. The lowest active level, call it l, receives all n elements in a span of 3 next ticks. This means that up to $\frac{n}{2}$ processors remain active at this level, for we merge two lists of size 2^l each using 2^l processors at each node. No level below level l is active. Levels above l all use $k = 4$. Thus level $l + 1$ uses up to $\frac{n}{8}$ processors, and so on, up to the highest active level, $\frac{3l}{2}$. This adds up to $O(n)$ active processors at any tick. Thus the total work complexity is $O(n \log n)$.

7.8 Application of Prefix-sum: Radix-Sort

Among sequential sorting algorithms, Radix-sort is known to be particularly efficient for sorting small integers. The main idea of the algorithm is to divide each element, rather the sorting key, into small parts. The algorithm iterates over parts, the least significant part first. Each part takes a fixed number of values. For example, a 32-bit integer naturally consists of 32 1-bit parts.

Suppose, in general, there are d parts, each taking D values. For each part, the entire list is divided into D buckets, which can be accomplished sequentially in $O(n)$ time for a list of size n. The sequential complexity to complete radix-sorting is $O(dn)$. Radix-sort relies on the different parts being sorted in a strict sequence. Hence, the only step that may be parallelized is the sorting of n items into D buckets.

Setting $d = 32$ for a 32-bit integer key, each of the d sorts amounts to a single prefix sum.

Listing 7.29: Parallel Radix-sort

```
1 // Radix-Sort list n-place
2 for i = 0 to 31
3   bsum = parallel prefix-sum bit[i] using $n$ processors // We count LSB as bit 0
4   forall processor p in 0..(n-1)
5     if(bit[i] of list[p] == 0)
6       rank[p] = p - bsum[p]
7     else
8       rank[p] = n - bsum[n-1] + bsum[p] - 1
9   list[rank[p]] = list[p]
```

This listing ensures that at iteration i, all elements with $bit[i] = 0$ move to the beginning of the list, and those with $bit[i] = 1$ move to the end. Additionally, they move in a stable way: two elements with $bit[i] = 0$ retain their order. Line 6 computes the rank for in the elements with bit 0. As seen in section 7.9, bit 0 piggy-backs on the prefix sum of bit 1. The number of elements with $bit[i] = 0$ before $list[p]$ is the total number of all elements before it, *i.e.*, p, less the ones with $bit[i] = 1$, *i.e.*, $bsum[p]$. Similarly, the number of elements with $bit[i] = 1$ before an element itself with $bit[i] = 1$ is simply its prefix sum minus 1. Of course, the elements with $bit[i] = 1$ must all be ranked after all those with $bit[i] = 0$. The total number of elements with $bit[i] = 0$ is $n - bsum[n-1]$. This is shown on line 8.

Since each prefix sum takes $O(\log n)$ time with $O(n)$ work, the total time complexity for radix-sorting d-part keys is $(d \log n)$, and the total work complexity is (dn).

Another version of Radix-sort iterates over the bits in the reverse order, most significant part first. For certain platforms, that version would be more suitable. In the most significant part first scheme, the elements with $bit[i] = 0$ and $bit[i] = 1$ are recursively divided into two buckets. Thus the first bit subdivides $list$ into two buckets, the next bit further subdivides each bucket into two, and so on. More importantly, each bucket can be sorted independently of the other buckets. Once a bucket becomes small enough, it may be sorted sequentially, while other processors continue to subdivide buckets. We leave those details as an exercise.

7.9 *Exploiting Parallelism: Quick-Sort*

The sequential quick-sort algorithm is organized quite like the sequential merge-sort algorithm, but there is no post-partition step. The quick-sort algorithm first partitions the given list $list1$ into two lists $list1s$ and $list1l$ in a way that all elements of $list1l$ are greater than every element of $list1s$. This generates two independent sub-problems that can be solved independently. This is top-down partitioning and follows a similar binary-tree structure as that of the merge-tree, but proceeds top to down. If we accept computing down the tree sequentially, the question that remains is: can we at least partition $list1$ in parallel. Prefix-sum is the answer.

Listing 7.30: Quick-sort Partitioning

```
// Partition list1[a..b]
pivot = random value in a..b
forall processor p in 0..(n-1)
  if(list1[p] < list1[pivot]) // Separate
    side[p] = 1
  else
    side[p] = 0
index = Parallel Prefix--sum(side) using n processors
pdest = index[n-1]
forall processor p in 0..(n-1)
  if(side[p] == 1)
    index[p] = index[p]-1
  else
    if(p == pivot)
      index[p] = pdest
    else if(p > pivot) // For large elements after the pivot position,
17        index[p] = pdest + p - index[p]
18      else
19        index[p] = pdest + p - index[p] + 1
20    list1[index[p]] = list1[p]
```

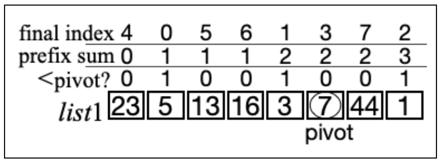

final index 4 0 5 6 1 3 7 2
prefix sum 0 1 1 1 2 2 2 3
<pivot? 0 1 0 0 1 0 0 1
list1 23 5 13 16 3 7 44 1
pivot

Figure 7.16: Partition a list. Suppose index 6 is chosen as the pivot. Determine which side of the pivot each element lies on. Then, perform a prefix-sum on the small side. The large side prefix-sum can be derived from the small side, which is shown in final index.

Partition for quick-sort amounts to asking for each element of *list*1, if its less than the pivot (call them *small*) or greater than it (call them *large*). The small and large sets are then sorted independently of each other. The pivot, if it is a part of *list*1, goes in the middle. This is trivially parallelizable with $O(1)$ time and $O(n)$ work. All processors must agree on the pivot. This requires concurrent read capability to complete in $O(1)$.

It is not sufficient to know which set an element belongs in. These sets must be formed as well. Quick-sort separates the two sets in-place. We can do the same in parallel in the PRAM model. We need to find non-conflicting indexes for elements to transfer to such that all the elements smaller than the pivot get smaller indexes than the rest. One can simply determine the rank of each element within its set. Line 8 in Listing 7.30 computes the prefix-sum of the list *side* and stores it in *index*. The elements less than the pivot have a 1 in *side*. Thus, $index[p]$ contains the number of small elements to the left of position p, plus 1 for itself. *index* provides a contiguous numbering for all small elements starting at 1: a small element at position p can be transferred to position $index[p] - 1$ without any conflict.

We similarly need to compute the prefix-sum for the large elements. A separate scan is not required because there are only two sets. The number of large elements to the left of position p is p minus the number of small elements to its left. However, the numbering for the large elements must start after the small elements and the pivot. An offset of *pdest* achieves that. This is computed on line 16.

The total complexity of partitioning is dominated by prefix-sum. Time complexity is $O(\log n)$, and work complexity is $O(n)$. Retaining the sequential quick-sort tree, there are $O(\log n)$ expected levels. (Please refer to a textbook on data structures or algorithms for an analysis of the number of expected levels.) The expected time complexity is $O(\log^2 n)$, and the expected work complexity is $(n \log n)$.

Note that the structure of the algorithm changes quite significantly in the BSP model. Processors cannot directly transfer an element to its proper side 'in-place.' Instead, each processor must send each element to the appropriate destination processor. Receiving processors may receive the elements in a location of their choice. (See Exercise 7.14.)

Back to the PRAM model, let us see if the time complexity can be reduced. Pipelining of prefix-sum could be an option, but no fast pipelining prefix-sum is known. Could we completely do without prefix-sum? We use it to separate the lists into small and large subsets. It may be possible to get by without such separation. Knowing the pivot v, each processor p knows the subset its element $list1[p]$ is in. Suppose, we simply let it use that information to determine how to proceed. Depending on whether $list1[p]$ is in $list1s$ or $list1l$, If p can determine its subset's next pivot, it can test again. For example, if $list1[p]$ is in $list1s$, and the pivot for $list1s$ is v_s, p needs to compare $list1[p]$ with pivot v_s next to determine which subset of $list1s$ $list1[p]$ lies in. It can go down the tree, comparing with a sequence of pivots. Thus, the initially *small* subset would be recursively divided into *small-small* and *small-large* subsets, and so on. If we can continue this process, labeling small as bit 0 and large as bit 1 at each test, the smallest element will see a sequence $0, 0, 0 \ldots$ and the largest would see $1, 1, 1 \ldots$. The bit for the first partition appearing first. This pivot itself is in neither set and may be assigned an *end* symbol indicating the termination of its bit sequence. Processor p terminates when $list1[p]$ is chosen as the pivot. Questions remain:

- How does each process receive an appropriate pivot in each round?
- How does the final rank of each element determined?

In principle, one may use any pivot. The ideal pivot for any set is its median. We generally rely on 'good' pivots by randomly choosing one of the elements in a set as its pivot. Pivots outside the range of values in a set are 'bad,' as they lead to up to n levels in the quicksort-tree. If we can find the minimum value m and the maximum values M within a set, $\frac{M+m}{2}$ may be a good choice. Section 7.3 explains how to find the minima in (1) time with $O(n^2)$ work with the common write facility in CRCW PRAM. However, that algorithm requires the knowledge of the full set, as each element is compared with every other element of the set. Another possibility is to evolve a consensus. If processor p knows the subset (or subsubset, etc.) its element $list1[p]$ is in, and a per-arranged location associated with that subset, it may write its own value as a proposal for the pivot of its subset. Concurrent write is required. In the arbitrary CRCW-PRAM model, one of the proposers succeeds. In the next step, all members of the set read the winning pivot value and use it to classify themselves at the next level of the tree.

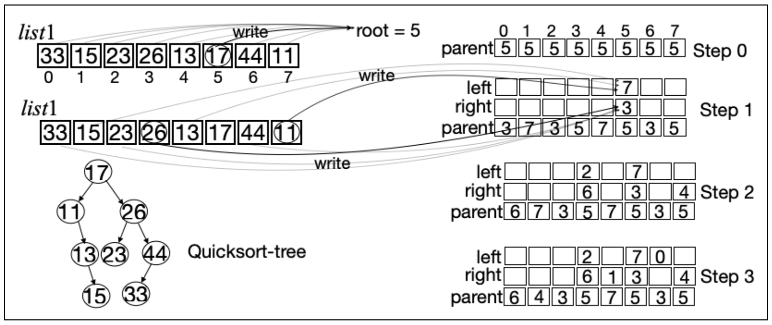


Figure 7.17: CRCW Quick-sort. For levels 0 and 1, all writes are shown as arcs. The winner is shown in a dark color. In this example, processor 5 wins writing to root first. Hence *list*[5] becomes pivot. Processors with elements less than *list*[5] compete to write to *left*[5]. Processor 7 wins, and its element, 11, becomes the left child. Similarly, processor 3 wins on the large side. The losers on each side update their parents to 7 and 3, respectively. Processor 5, the parent processor, does not lie on either side and, hence, does not compete to write. The same process continues at other levels.

How to pre-arrange the location per subset? One way to designate unique location for each subset is its position in the quick-sort tree itself. See Figure 7.17. We count the levels downward. The first partition is at level 0, which creates two sublists, one contains all elements in the left subtree of the parent (say, the *small* sublist) and the other in the right subtree. Let $parent[p], left[p]$, and $right[p]$ contain the indexes, respectively, of the parent, the left child, and the right child for element $list1[p]$. $parent[p]$ doubles as the pivot for element $list1[p]$. Initially, all elements use the same pivot, and $parent[p] = root$, one selected pivot. Once the sublists separate, $parent[p]$ changes accordingly.

As described above, $parent[p]$ (*i.e.*, pivot) is set by the arbitrary write of CRCW-PRAM by processors of each sublist. Processors p with elements smaller than $list1[parent[p]]$ compete and write to the $left[parent[p]]$. One succeeds. Processors with elements larger than the pivot compete to become the right child. Again, one wins. All the writers read back to check if they won writing. The winner is the pivot. Losers need to continue the process. For simplicity, Listing 7.31 lets the chosen pivot also continue to process until the final termination. However, they are the only elements in their list – so they continue to repeat the same computation.

Listing 7.31: Arbitrary CRCW Quick-sort

```
//Quick-sort list
forall processor p in 0..(n-1)
  root = p
  parent[p] = root
done = 1
while(! done)
  forall processor p in 0..(n-1)
    done = 1
    if(list[p] < list[parent[p]])
      leftchild[parent[p]] = p // Write winner becomes the left child
      if(leftchild[parent[p]] != p) // Lost write. Retry with new parent.
        parent[p] = leftchild[parent[p]]
        done = 0
    if(list[p] > list[parent[p]])
      righttchild[parent[p]] = p // Write winner becomes the right child
      if(leftchild[parent[p]] != p)
        parent[p] = rightchild[parent[p]] // Lost write. Retry with new parent.
        done = 0
```

The expected time complexity of Listing 7.31 is $O(\log n)$, as it takes $O(1)$ time per iteration, with n processors active. The number of iterations is one more than the number of levels in the quicksort-tree. One last iteration is required to ensure that no processor sets *done* to 0. Thus, the total work complexity is $O(n \log n)$. Recall that concurrent writes incur a cost even in modern shared-memory platforms. Nonetheless, the algorithm described above is illustrative of a general application of leader election (which is a form of the consensus problem).

7.10 Fixing Processor Count: Sample-Sort

As mentioned earlier, concurrent PRAM operations do not translate well to practical programming platforms. In a more practical setting, P processors, each, hold $\frac{n}{P}$ of the initial list that is to be sorted. As described in Chapter 5, a good parallel design attempts to maximize P relatively independent tasks, each performed sequentially at one of the processors. In the context of merge-sort, one would locally sort the $\frac{n}{P}$ elements at each processor, and follow that up with a P-way merge of the P sorted lists. Similarly, in the quick-sort variant, it may be useful to partition the initial list into P sublists, with each sublist getting elements larger than the previous sublist. Then, each sublist can be sorted by one of the P processors independently of others. Both of these methods are facilitated by sample-sort.

Sample-sort is based on the creation of a smaller list of size $O(P)$, which is a cover for the initial list, not unlike the pipelined merge-sort. The main idea is to be able to find P disjoint ranges of values $\{R_i, i \in 0..(P-1)\}$ so that the upper limit of range i is less than the lower limit of range $i+1$, and the total number of elements in each range is roughly equal. Once $\{R_i\}$ is known, it is trivial to determine the range in which each element $list[i]$ lies. Thus, in $O(\log P)$ time and $O(n \log P)$ work, we can compute P lists $r[j], j \in 0..(P-1)$. $r[j][k]$ contains the range numbers that element k at processor [j] lies in. Ranges are also called *bucket*s.

Next, we collect all elements in bucket i at processor i. This is the partition step of Quick-sort. Each bucket can then be independently sorted. Alternatively, we can first sort the $\frac{n}{P}$ elements at each processor, and then distribute one bucket per processor. The so collected

P sorted sublists at each processor are merged to complete the sorting. Both schemes have similar communication requirements, and both rely on creating ranges in a way that each bucket is equitably distributed among all processors. This range creating follows the method of sampling as follows:

Listing 7.32: Parallel Sampling

```
1 // Find well distributed samples of list
2 forall processor p in 0..(P-1)
3   locally sort list[p] containing n/P elements available at processor p
4   sublist[p] = {list[p][i*n/(P*P)]} i in 1..(P-1) // Take P-1 separators
5 slist = sort(subllist[p], p in 0..P-1) using P processors // All separators
6 range = {slist[i*(P-1)]} i in 1..(P-1) // P-1 evenly sampled splitters
```

Listing 7.32 assumes that the input *list* is equally distributed among P processors; $list[p]$ comprises the set of elements at processor p. Each processor initially sorts its set to find well-separated samples. Each chooses $P-1$ *separators*. These $P*(P-1)$ total separators are again sorted, possibly using all P processors, in order to next find $P-1$ globally well-separated splitters. These splitters are put in the list $range[0..(P-2)]$. Two consecutive elements define a range. We may assume $range[-1] = -\infty$, and $range[P-1] = +\infty$ as default splitters. Thus, there are P ranges. We also allow each range to be open at its upper end to ensure that there is no overlap among consecutive ranges.

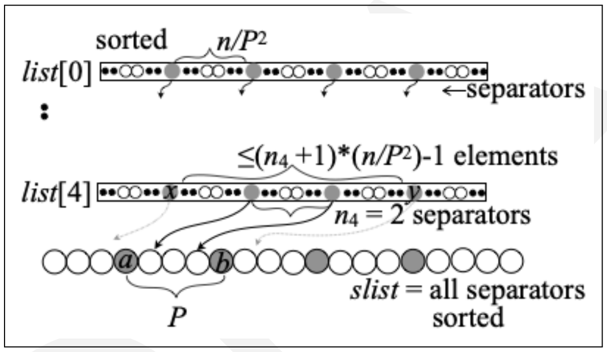


Figure 7.18: Sampling a list. Assume $P = 5$. $list[0]$ and $list[4]$ are shown after local sorting. Filled circles stand for chosen separators, $P-1$ per processor. *slist* shows all separators of all processors. Again, filled circles in *slist* show the chosen $P-1$ splitters.

range is a $\frac{2n}{P}$ cover of *list*, meaning there are no more than $\frac{2n}{P}$ elements in *list* between any consecutive elements of *range*. Figure 7.18 explains this. Note that between any two consecutive elements of *range*, say a and b, there are P elements of *slist*, each of which is a separator selected on line 4 of Listing 7.32. Suppose n_i of these separators come from processor i. These n_i separators are necessarily consecutive separators of processor i, and any separator just before or just after these n_i separators in $list[p]$ (see x and y in the figure) are not in the range $a..b$. Up to $\frac{n}{P^2}-1$ non-separators just after x and up to $\frac{n}{P^2} - 1$ non-separators just before y, could be in the range $a..b$. Counting them in, no more than $(n_i + 1)\frac{n}{P^2} - 1$ elements in $list[i]$ may lie in the range $a..b$. That means there are never more than

$$\sum_{i=0}^{P-1} \left[(n_i + 1)\frac{n}{P^2} - 1\right] = \frac{n}{P^2}\sum_{i=0}^{P-1}(n_i + 1) - P \leq \frac{2n}{P}$$

total elements in $list$ in the range $a..b$, since $\sum n_i = P$.

Local sequential sorting at line 3 requires $O(\frac{n}{P}\log\frac{n}{P})$ time and $O(n\log\frac{n}{P})$ work, with P processors performing each sort. Forming *sublist* takes $O(P)$ steps at each processor. The parallel sort of P sorted lists, one per processor, and each of size $P - 1$, appears to be similar to the original problem and may be performed recursively. However, this step is not the bottleneck for the entire algorithm if P is small compared to n. Pair-wise merge, as in the merge-tree of Figure 7.10, suffices. The height of the tree is $\log P$. At level i, $\frac{P}{2^{i+1}}$ processors send $2^i(P-1)$ elements each to their siblings, and those $\frac{P}{2^{i+1}}$ siblings merge two lists of size $2^i(P-1)$ each.

The computation time at level i is $O(2^i P)$, and work is $O(P^2)$. For the BSP model, $O(\frac{P}{2^{i+1}})$ messages are sent at level i. Hence, the total time complexity of merging at line 5 is $O(P^2)$, and work complexity is $O(P^2\log P)$. $O(P)$ messages are sent. Sampling *slist* into *range* at one processor takes $O(P)$ time.

Finally, to complete sample-sort, the list *range* is broadcast to all processors requiring $O(P)$ messages. After all the processors receive *range*, they form P buckets each, taking $O(\frac{n}{P}\log P)$ time each. Then, each processor sends its elements of bucket i (if any) to processor i. In the BSP model, each processor sends a single message to possibly all $P - 1$ other processors, leading to the communication cost of $P * (P-1) = O(P^2)$. After receiving its bucket from $P - 1$ other processors, processor i, merges them in time $O(\frac{n}{P})$, since no bucket has more than $\frac{2n}{P}$ elements.

Thus, the total time complexity is bounded by the initial local sorting: $O(\frac{n}{P}\log\frac{n}{P})$.

7.11 *Exploiting Parallelism: Minimum Spanning Tree*

We close this chapter with an example of discovering opportunities for parallelization in sequential graph algorithms. An oft-required task for weighted graphs is to find its skeleton – the minimum spanning tree. In this context, weighted graphs have weights associated with their edges. The spanning tree of a graph is a tree that includes all its vertices and a subset of its edges. The tree must be connected and without any cycles by definition. (A cycle is a path that does not repeat any vertex.) The weight of a spanning tree is the sum of the weights of all its edges. Many spanning trees may be formed in a graph. The minimum spanning tree (or MST) of a graph is one whose weight is no greater than the others. Figure 7.19 shows the edge-weights of an example graph. The MST is shown in solid edges. Non-MST edges are in dashed lines.

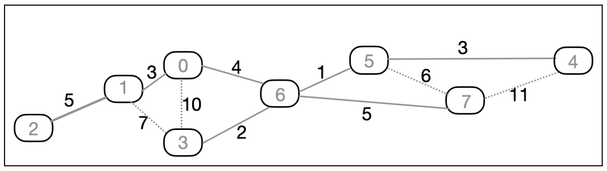


Figure 7.19: Minimum Spanning Tree of a Weighted Graph

Prim's algorithm[6] is a greedy sequential algorithm to compute MST of a given weighted undirected graph G with n vertices and m edges. It incrementally builds MST by adding one vertex and its connecting edge at a time, starting with an arbitrary vertex. At each step, it selects the edge with the least weight among those connecting any vertex $v \in$ MST constructed so far with any vertex $w \in G-$ MST.

[6] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6): 1389–1401, 1957

In Listing 7.33, when vertex *minv* is added to MST, its neighbor in the MST is specified by Parent[*minv*]. In other words, (Parent[*minv*], *minv*) is the connecting edge. The first vertex added to the MST has a null Parent. An auxiliary array Cost[w] maintains the weight of the least-weight edge connecting vertex w to the evolving MST. Cost is not known in the beginning, and an overestimate is maintained. These estimates are lazily improved when a neighbor of w is included in MST. When (Parent[*minv*], *minv*) is added to MST, the weight of that edge, which is Cost[*minv*], is smaller than the weights of all other edges connecting any vertex in MST with any vertex in $G-$ MST.

Inductively, if the current edges in MST are in the final **MST**, edge (Parent[*minv*], *minv*) must also be in the final **MST**. Otherwise, the path in the final **MST** from *minv* to Parent[*minv*] would have to go through another edge (w, v) where $w \in G-$ MST and $v \in$ MST. However, then the cost of that final **MST** could be reduced by replacing edge (w, v) with (Parent[*minv*], *minv*).

Listing 7.33: Prim's MST algorithm

```
for v in 1..n // n = Number of vertices in $G$. We are counting from 1 here.
  Cost[v] = ∞
  Parent[v] = null
MST = null
GV = {Vertices in G}
for i in 1..n
  minv = vertex with min Cost[v] ∀ v ∈ GV // Break tie arbitrarily
  GV = GV - minv
  MST = MST ∪ (Parent[minv], minv)
```

```
10   for each vertex adjacent to minv
11     if Cost[v] > EdgeWeight(minv, v)
12       Cost[v] = EdgeWeight(minv, v)
13       Parent[v] = minv
```

The time-consuming operations in this loop is the minimum cost discovery on line 7 and the Cost updates on line 10. These are usually implemented with the help of a priority queue holding up to n keys. This priority queue stands for GV. The cost reduction on line 12 requires up to $deg[minv]$ decrease-key operations in the priority queue, $deg[minv]$ being the degree of vertex $minv$. Of course, these $deg[minv]$ edges connected to $minv$ are updated only once when $minv$ is added to MST. Fibonacci Heaps[7] require $O(1)$ amortized time per decrease. Relaxed Heaps[8] can complete each decrease in $O(1)$ time in the worst case. Both require $O(\log n)$ time for each extraction of the minimum Cost vertex on line 7. This adds up to $O(m + n \log n)$ sequential time for the entire algorithm. Note that for dense graphs where, say, $m > n \log n$, this time is bounded by $O(m)$. For sparser graphs, $n \log n$ dominates.

[7] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, jul 1987. ISSN 0004-5411. URL https://doi.org/10.1145/28869.28874

[8] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31 (11):1343–1354, nov 1988

Let's see how Prim's algorithm admits parallelism. The initialization on line 1 can be completed on EREW PRAM in $O(1)$ time and $O(n)$ work. The loop on line 6 is inherently sequential and requires n iterations. Maybe, the priority queue operations can be parallelized: a single extract-min operation and multiple decrease-key operations.

One option is to subdivide the queue among many processors. Up to n processors could be used, with processor v associated with maintaining Cost$[v]$. Thus, the time to build this 'trivial' queue is $O(1)$ with $O(n)$ work. We can then resort to the minima-finding algorithms discussed in Section 7.3. That would require $O(\log n)$ time with $O(n)$ work on EREW PRAM to extract the minimum cost vertex. Decrease operations can be completed in $O(\log n)$ time with $O(n)$ work, with each processor v updating Cost$[v]$ in parallel. Note that $O(\log n)$ time would be required for each processor to read the value of $minv$. If we allow CREW PRAM, the decrease operations would complete in $O(1)$ time. EdgeWeight$(minv, v)$ can be located in $O(1)$ time by processor v with an adjacency matrix-based representation. This is demonstrated in Listing 7.34.

Listing 7.34: Parallelized Prim's MST algorithm

```
1 forall v in 1..n // n = Number of vertices in G
2   Cost[v] = ∞
3   Parent[v] = null
4   inMST[v] = false
5 MST = null
6 GV = Build Priority Queue on Cost // Sequential time complexity is $O(n)$
```

```
7  for i in 1..n
8    minv = ExtractMin (GV) // Break tie arbitrarily
9    inMST[minv] = true
10   Append (Parent[minv], minv) to MST
11   forall v in 1..n
12     if inMST[v] == false && Cost[v] > EdgeWeight[minv][v] // EdgeWeight stored in adjacency matrix
13       DecreaseKey(GV) for v from Cost[v] to EdgeWeight[minv][v]
14       Cost[v] = EdgeWeight[minv][v]
15       Parent[v] = minv
```

Listing 7.34 takes $O(n \log n)$ time and $O(n^2)$ work, given that all n processors are busy for all n iterations. That is not efficient, particularly for sparse graphs. Maybe, a more specialized data structure can help: a parallel priority queue based on Binomial Heaps[9] allows $O(1)$-time $O(\log n)$-work extraction and decrease-key operations on CREW PRAM (see Section 7.11).

[9] Gerth Stølting Brodal. Priority queues on parallel machines. *Parallel Computing*, 25(8):987–1011, 1999

The total time on line 8 is then $O(n)$ with $O(n \log n)$ work. However, the inner loop on line 11 does not meet those bounds. We need to restructure this loop to focus on the actual number of edges incident on vertex *minv*, as in Listing 7.35. We assume that array $adj[v]$ (of size $deg[v]$) stores the identifier of every vertex w adjacent to vertex v. Similarly, EdgeWeight$[v][j]$ stores the weight of edge $(v, adj[v][j])$.

Listing 7.35: Restructuring the Cost update loop at line 11 of Listing 7.34 Prim's MST algorithm with Parallel Priority Queue

```
  for j in 1..deg[minv]
    v = adj[minv][j]
    if inMST[v] == false && Cost[v] > EdgeWeight[minv][j]
      DecreaseKey(GV) for v from Cost[v] to EdgeWeight[minv][j]
      Cost[v] = EdgeWeight[minv][j]
      Parent[v] = minv
```

The total number of decrease-key operations now is $O(m)$, leading to the total time $O(m + n)$ and work $O((m + n) \log n)$. This is a minor improvement over sequential time for sparse graphs. It is worth noting that other graph algorithms, *e.g.*, Dijkstra's shortest path algorithm[10], also benefit similarly from parallel priority queue operations similarly. That is left as an exercise.

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959

Parallel Priority Queue

Binomial trees of rank r are defined recursively as follows:

1. Binomial tree of rank 0 is a single node. Call it B_0.

2. Binomial tree of rank r, B_r, is formed by two Binomial trees of rank $r - 1$, $B1_{r-1}$ and $B2_{r-1}$ by making the root of one, say $B1$, a

child of the other, $B2$. For example, in Figure 7.20(c), B_2 is constructed by making the root of $B2_1$ the second child of the root of $B1_1$.

In general, the children of the root of B_r are respectively themselves roots of B_0, B_1, .. B_{r-1}, as seen in Figure 7.20(d). A simple induction on rank shows that a Binomial tree of rank r has 2^r nodes and depth r. A Binomial heap stores keys at all nodes with the constraint that the key at every node is smaller (or larger for max-Heap) than the keys at its children. The parallel priority queue Q is represented as a

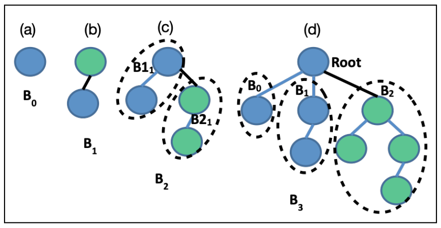


Figure 7.20: Examples of Binomial Trees of rank 0 to rank 4 from left to right

forest of Binomial heaps with the constraint that there are between 1 and 3 binomial heaps of every rank r, $r \leq Q_r$, the rank of the priority queue. In particular, let $n_i(Q)$ be the number of trees with rank i in Q. $n_i(Q) = \{1, 2, 3\}$ for $i = 0..Q_r$. This ensures the $Q_r \leq (1 + \log n)$ if n items are stored in the queue. We associate the rank of a Binomial heap also with its root, meaning the root of a rank r heap has rank r. Also, among a given set of nodes, the one with the minimum key is called the minimum node in short. The forest Q has the following ordering property on the key values at its nodes.

Minimum Root: The minimum root of rank r in Q is smaller than all roots of rank $r' \geq r$.

By this property, the minimum key of the queue is always at the minimum root among roots of rank 0. For simplicity, we assume that the key values are unique, meaning all ties are broken. Let $Q.root[i]$ be the list of roots with rank i. Further, let node.child$[i]$ denote the list of children of any node. We refer to the key at a node by node.key. Two operations *Link* and *Unlink* are defined, which help maintain this property using multiple processors.

$Link(h1, h2)$ links the root $h2$ of heap $H2_r$ as a child of $h1$, the root of another heap $H1_r$, each of rank r. For the heap order property, $h1$.key< $h2$.key for this operation. This forms a new heap of rank $r + 1$ with root h. This linking is an $O(1)$-time operation by a single processor assuming $h2$ can be appended to $h1.child$ in $O(1)$. In addition, roots $h1$ and $h2$ must be removed from $Q.root[r]$ and h appended to $Q.root[r + 1]$. This can be accomplished by, say, using a linked list structure for *root* and *child*. In principle, this requires $O(1)$-time memory allocation. This can be handled by allocating P nodes per invocation of *Link* when using P processors.

Unlink is the inverse of *Link*. $Unlink(r)$ separates H_{r+1}, a heap of rank $r+1$ into two heaps of rank r each. This is accomplished by nullifying the last child $h1$ of root h of H_{r+1}. After this, h is removed from $Q.roots[r+1]$, and h and $h1$ are both added to $Q.roots[r]$. Again $Unlink(h)$ takes one processor $O(1)$ time.

The following conditions allow *Link* and *Unlink* to maintain the Priority queue properties.

1. One linking is done for the heaps of each rank, except the minimum root of any rank r never participates in *Link*. This means that linking is only carried out if at least 3 roots of rank r exist. This ensures that the Minimum Root Property is not perturbed by *Link*.

2. Condition 1 also means that at most one pair of nodes of rank r may be linked together. If this holds for all ranks, up to two trees of each rank r may be removed by linking, and at most one tree of rank r may be added (when two rank $r-1$ trees are linked). This means that if there are fewer than 3 trees of rank r before the linkings, there can be at most 3 trees of rank r after the linking. On the other hand, if there are more than 2 trees of rank r ($n_r(Q)$ can temporarily reach 4) before the linkings, $n_r(Q)$ reduces by at least 1. Thus, $n_r(Q) \in \{1, 2, 3\}$ for all r after the linkings.

3. Only the minimum root of any rank $r, r > 0$, is unlinked. After the unliking, that minimum root becomes a root of rank $r-1$. It may become the minimum root of rank $r-1$, or a smaller root of rank $r-1$ may already exist. Hence, the Minimum Root property continues to hold.

4. Condition 3 also means that a root at each rank is unlinked. This removes one root of rank r, but adds two roots of rank r, which were unlinked from the erstwhile minimum root of rank $r+1$, unless r was the maximum rank. In that case, the maximum rabkn reduces by 1. Thus, the net increase in $n_r(Q)$ is 1 for $Q_r > r > 0$. Since the roots of rank 0 cannot be unlinked, $n_0(Q)$ increases by 2 after the unlinkings.

The parallel linking and unlinking are specified in Listing 7.36 and 7.37.

Listing 7.36: Parallel Link

```
forall r in $0..\lfloor 1 + \log n \rfloor - 2$
  if(Q.root[r].size >= 3)
       (max, nextmax) = Two largest roots of rank r
    L = Link(Q.root[r][max], Q.root[r][nextmax])
    Remove(Q.root[r], max)
    Remove(Q.root[r], nextmax)
    Add(Q.root[r+1], L)
```

Listing 7.37: Parallel Unlink

```
forall r in 1..$\lfloor 1 + \log n \rfloor - 1$
  if(Q.root[r].size >= 1)
        min = Smallest roots of rank r
    (L1, L2) = Unink(Q.root[r][min])
    Remove(Q.root[r], min)
    Add(Q.root[r-1], L1)
    Add(Q.root[r-1], L2)
```

To insert an element to Q, a new singleton node e is created with the element, and e is inserted to $Q.root[0]$ and Parallel Link of Listing 7.36 is invoked on Q. The minimum element is always listed in $Q.root[0]$ (which can be kept sorted by the key). To remove the node with the minimum element, call it $Q.root[0][min]$, we simply remove the node from $Q.root[0]$. However, this may violate the Minimum Root property. The erstwhile second smallest root need not be in a rank 0 tree. In that case, the minimum root at rank 1 must be the new smallest, as it is guaranteed to be smaller than the smallest roots at higher ranks. One invocation of Parallel Unlink of Listing 7.37 can bring the new minima to rank 0. This can lead to 4 roots of rank 0, however. One invocation of Parallel Link (Listing 7.36) brings it down to 2.

MST with Parallel Priority Queue

Although it is possible to also decrease keys on $O(1)$ time using a pipelined version of *Link* and *Unlink*[11], for MST computation, it is sufficient to simply insert the new Cost$[v]$. This allows multiple entries for v to co-exist, but the new cost is always lower, and it will be extracted first. This would imply that those stale larger key values could be extracted later. We can use the inMST$[v]$ marker to discard such extraction. Since at most m edges exist in the graph, up to m entries may now exist in the priority queue, and up to m extractions could occur on line 1 of Listing 7.34 (see Listing 7.38). That amounts to $O(m)$ total time to compute MST with $(m \log m)$ work. Note that $\log m$ is $O(\log n)$. As a result, the total work remains $O(m \log n)$. We assume $m \geq n - 1$. Otherwise, a spanning tree of G does not exist.

[11] Gerth Stølting Brodal. Priority queues on parallel machines. *Parallel Computing*, 25(8):987–1011, 1999

Listing 7.38: Parallelized Prim's MST algorithm with Parallel Priority Queue

```
forall v in 1..n // n = Number of vertices in G
```

```
2    Cost[v] = ∞
3    Parent[v] = null
4    inMST[v] = false
5  MST = null
6  GV = Build Priority Queue on Cost
7  for i in 1..n
8    repeat
9      minv = ExtractMin (GV) // Break tie arbitrarily
10   until inMST[minv] is false
11   inMST[minv] = true
12   Append (Parent[minv], minv) to MST
13   forall v in 1..degree[minv]
14     if inMST[v] == false && Cost[v] > EdgeWeight[minv][v]
15       Cost[v] = EdgeWeight[minv][v]
16       Parent[v] = minv
17       Insert(GV, (v , Cost[v]))
```

A shortcoming of all variants of the Prim's algorithm demonstrated above is the sequential loop (line 6 of Listing 7.33). This can only be addressed by an entirely different structure. For example, Sollin's algorithm[12] begins with each vertex of G as an isolated tree. It merges these trees (not unlike the connected components algorithm of Section 7.6) by including the minimum-weight edge going out of each tree, thus building larger trees. This affords an opportunity for multiple parallel mergers. Indeed, the number of trees remaining after each iteration is at most half of the number before the iteration. This is explored in Exercise 7.21.

[12] Joseph Jájá. *Introduction to Parallel Algorithms*. Pearson, 1992

7.12 *Summary*

This chapter presents several techniques useful to develop parallel algorithms. A model like PRAM often helps simplify thinking about the algorithm. Nonetheless, we need to evaluate the algorithms also in the context of practical architecture. For most shared-memory machines, CREW PRAM works well. Arbitrary-CRCW PRAM has some overhead but still works. On the other hand, only EREW PRAM is likely to translate well to message passing machines. For this environment, the BSP model provides avenues for a more careful accounting of overheads.

In either case, while designing parallel algorithms, we can either think in terms of the total number of operations required or the number of processors required. They are equivalent formulations of work complexity as a function of the problem size n. Sometimes, it is convenient to instead think of the number of processors p also as a parameter. The time taken by p processors on the problem with size n then becomes the way to measure complexity.

Many times the task of designing an efficient algorithm amounts to finding which sequential dependencies to break. This could be accomplished *e.g.*, by repeating a computation or deferring a computation by allowing it to proceed concurrently. Deferring works well in case the partial results of the originally dependent computation can be later retrofitted with the late-arriving results of that concurrent computation.

Divide and conquer is among the most pervasive paradigms for parallel algorithm design, just as it is for sequential algorithm design. Division into two sub-problems at a time is quite standard in the sequential domain. That is often valuable in parallel algorithms as well and manifests as a binary computation tree. However, digging a bit deeper into the work-scheduling principle, subdivision into more than two problems at a time is useful in the parallel domain. Furthermore, in parallel algorithms, it may not always pay to carry out the recursion all the way to the top of the recursion tree, where increasingly fewer processors are employed. Nonetheless, if the total work remaining at the top of the tree is small, it matters little. Indeed, we can exploit that fact by using a less work-efficient and more time-efficient algorithm at the top levels.

Accelerated cascading is a powerful design pattern for parallel algorithms. It allows us to combine a divide and conquer solution with low time complexity but high work complexity with one that has a higher time complexity and a lower work complexity. Usually, the lower time complexity is obtained by subdividing the problem much more aggressively than into two each time. For example, if we subdivide a problem of size n into $O(\sqrt{n})$ each time, the height of the tree shrinks to doubly-log in n: $\log \log n$. If we subdivide into two (or a fixed number) at a time, the height is $O(\log n)$. However, the shrinkage in height can come at the cost of work complexity. By using the slower algorithm at the lower levels of the tree, we can quickly reduce the problem size. Employing the higher-work complexity algorithm on the smaller problems then adds up to lower total work.

Pointer jumping is another handy tool for graph and list traversal, where multiple processors can proceed with multiple traversals in parallel, exploiting each other's traversals. Pipelining is a useful tool when an algorithm is divisible into parts that need to be performed in sequence by a series of data items. Inserting multiple items into a tree is a good example. Many graph algorithms are based on processing edges or vertices in a certain order. This order can vary dynamically, and can be updated each time the next vertex or edge is selected. Minimum spanning tree computation is an example. Parallel operation of priority queues can be a useful tool in such situations.

Exercise

7.1. Give an $O(1)$ EREW PRAM algorithm to find the index of the single 1 in $BITS$, a list of n bits. There is at most one 1 in $BITS$. If no 1 exists in $BITS$, the output must be n.

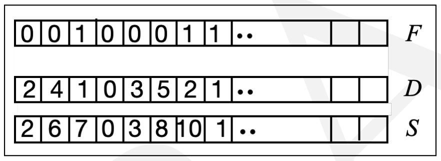
7.2. Analyze the time and work complexity of the recursive dependency-breaking parallel algorithm to compute the prefix-sum introduced as Method 1 in Section 7.1.

7.3. Prove that the algorithm in Listing 7.5 computes the prefix sum. Analyze its time and work complexity.

7.4. Modify the algorithm in Listing 7.7 computes the exclusive prefix sum. (Try not to first compute the prefix-sum before computing the exclusive sum from it).

7.5. Re-pose all the three methods for parallel prefix-sum computation discussed in Section 7.1 under the BSP model. Analyze their time and work complexity, and compare their performance.

7.6. Devise and algorithm to compute segmented prefix-sum S of array D, where the segment markers are given in an array F, as shown below:



Segment begins at index where F does not have a 0 to its immediate left, and continues until a non-0 is discovered. We say a non-existent value like $F[-1]$ is non-0. Prefix-sum is computed for each segment.

7.7. Modify the segmented prefix-sum algorithm of Exerciseex:segscan to compute segmented prefix-minima M, given D and S. $M[i]$ is the minimum element $D[j]$ among all $j < i$ within the segment of i.

7.8. Efficiently Compute the polynomial

$$\sum_{i=0}^{n} D[i]x^i,$$

given an array D, the integer n, and a real number x.

7.9. Given a list BIT of n 1-bit values, find the lowest such index i that $B[i] = 1$. If no bit is 1, the answer is n. Find an $O(\log n)$ time algorithm with $O(n)$ work for EREW PRAM. Find an $O(1)$ time and $O(n)$ work algorithm for common-CRCW PRAM. (Hint: Subdivide BIT into blocks of $\sqrt{n}$.)

7.10. Given a list INT of n integers, compute list $ANSV$ such that $ANSV[i]$ stores the value $INT[j]$ found at the largest index j where $INT[j] < INT[i]$ and $j < i$. If no such j exists, the answer is n. Find an $O(1)$ time algorithm with $O(n^2)$ work to compute ANSV on Common-CRCW PRAM. (Hint: Use Exercise 7.9.) Find an $O(\log n)$ time algorithm with $O(n)$ work to compute ANSV for EREW PRAM.

7.11. Compute Prefix-minima M given input integer list D with n elements ($M[i]$ as the minimum of all $D[j]$ among $j < i$) in $O(\log \log n)$ time using $O(n \log \log n)$ work on Common-CRCW PRAM. (Hint: Use Exercise 7.10 to devise accelerated cascading.)

7.12. Recall Merge Method 1 in Section 7.2. There we subdivided the problem of merging two sorted lists into two unequal subproblems by finding the rank of the middle element of the larger list in the other list. This may not subdivide the smaller list equally. What if we also locate the middle element of the second list in the larger list. This will lead to three merger sub-problems, none of them with more than half the elements from either list. Does that lead to an improved performance in time and work? Analyze.

7.13. Given a connected undirected graph G with n vertices and a list of m edges E, where i^{th} edge $E[i]$ is a pair of integers $(u, v), u < n, v < n$ indicating that vertex number u and vertex number v have an edge between them. Given P PRAM processors, compute the list $RANK$ such that $RANK[j]$ is the level of vertex number i in breadth-first search of G starting at vertex 0. You may use any PRAM model.

7.14. Consider mapping the Quicksort partitioning algorithm on P message passing processors. Revise the algorithm discussed in Section 7.9 to account for the given P and provide the time complexity under the BSP model.

7.15. Reformulate the Sample sort algorithm in Section 7.10 for a CREW PRAM and analyze its time and work complexity.

7.16. A parallel strategy for sorting is to focus on finding the rank of each element. This is also called enumeration sort. With n^2 comparisons, ranks of each of n element can be determined. Of course, we still need to find the rank of an element after comparing it to $n-1$ other elements. How quickly can you enumeration-sort n elements on

(a) CREW PRAM

(b) EREW PRAM

Provide the time and work complexities.

7.17. The selection problem is to find the k^{th} smallest element in a list *List* of n unsorted elements. Devise a parallel selection algorithm taking $O(\log^2 n)$ time and $O(n)$ work on CREW PRAM. (Hint: Consider recursively reducing the problem of selecting from n_1 unordered items to a problem of selecting from no more than $\frac{3n_1}{4}$ items in $\log n_1$ time with $O(n_1)$ work.)

7.18. Assume a sorting algorithm that runs in O(log n) time with O(n log n) work on CREW PRAM. Combine this sorting algorithm with the selection algorithm in Exercise 7.17 to accomplish parallel selection in $O(\log\log n)$ time with $O(n)$ work.

7.19. Consider the first algorithm in Section 7.6. Modify the algorithm to produce labels such that those labels double as the component number. All components must be sequentially numbered, but in any arbitrary order. Is there any change to the work complexity of the connected components algorithm due to this additional requirement?

7.20. Consider the CREW-PRAM minimum spanning tree algorithm described in Section 3.3. What changes could make it work on EREW-PRAM? What is the resulting time and work complexity?

7.21. Provide an CREW-PRAM algorithm with $O(\log^2 n)$ time and $O(n^2)$ work complexities to compute the minimum spanning tree of a graph with n vertices. (Hint: Start with many small trees and merge them until a single tree remains.)

7.22. Provide an algorithm to compute the shortest path between two vertices v_1 and v_2 in an undirected weighted graph on EREW and CREW PRAM. respectively.