CHAPTER

Merge Sort 13

In a **divide-and-conquer** algorithm, either the divide or merge step can sometimes become a bottleneck if done serially. Parallel merge sort demonstrates a case where the merge step can be parallelized to avoid a bottleneck by using divide-and-conquer. The net effect is a divide-and-conquer algorithm running inside a divide-and-conquer algorithm. This illustrates the practical importance of having a parallel framework that supports efficient **nested** parallelism.

Serial merge sort has two desirable properties:

- It is stable (that is, the order of equal elements is not changed).
- It has guaranteed asymptotic running time $O(N \lg N)$.

Weighing against these desirable properties is the disadvantage that merge sort is an out-of-place sort, and thus has a bigger memory and cache footprint than in-place sort algorithms. Conversely, Quicksort is an in-place sort, but lacks the other two properties. In particular, though Quicksort's expected time is $O(N \lg N)$, its worst-case time is $O(N^2)$.

Serial merge sort is a divide-and-conquer algorithm where the basic recursive steps are:

1. Divide the sequence into two subsequences.
2. Sort each subsequence.
3. Merge the sorted subsequences.

The merge step has work $\Theta(N)$ to merge N items. If done serially, it can become a bottleneck just as serial partitioning can for Quicksort (Section 8.9.3). Indeed, there is a symmetry to their respective bottlenecks. In Quicksort, a serial divide took time $O(N)$. In merge sort, a serial merge takes time $O(N)$. The effect on the complexity analysis recurrence relations is similar, with similar consequences: The asymptotic speedup for sorting N items is $O(\lg N)$. Though efficient for small core counts and big sequences, it is not **scalable**.

13.1 PARALLEL MERGE

Improving the speedup requires parallelizing the merge step. How to do so is a good lesson in why it is sometimes better to seek a new algorithm than parallelize the obvious serial algorithm. To see this, consider two sorted sequences X and Y that must be merged. The usual serial algorithm inspects the head items of the two sequences and moves the smaller head item to the output sequence. It repeats this incremental inspection until either X or Y becomes empty. Any remaining items are appended to the output. Listing 13.1 shows the code, where X and Y are represented by the half-open intervals `[xs,xe)` and `[ys,ye)`, respectively.

```
void serial_merge( T* xs, T * xe, T * ys, T * ye, T* zs ) {
   while( xs!=xe && ys!=ye ) {
      bool which = *ys < *xs;
      *zs++ = std::move(which ? *ys++ : *xs++);
   }
   std::move( xs, xe, zs );
   std::move( ys, ye, zs );
}
```

LISTING 13.1

Serial merge. Appendix D.3 explains the meaning of `std::move`.

A subtle point worth attention is the test in the inner loop. If a key in X and a key in Y are equal, the key in X comes first in the output. This rule for breaking ties ensures that if `serial_merge` is used for a merge sort then the sort is stable.

Trying to parallelize `serial_merge` directly is hopeless. The algorithm is inherently serial, because each iteration of the `while` loop depends upon the previous iteration. However, there is a remarkably simple **divide-and-conquer** alternative to the usual serial algorithm. Without loss of generality, assume that sequence X is at least as long as sequence Y. The sequences can be merged recursively as follows:

1. Split X into two contiguous subsequences X_1 and X_2 of approximately equal length. Let K be the first key of X_2.
2. Use binary search on Y to find the point where K could be inserted into Y. Split Y at that point into two subsequences Y_1 and Y_2.
3. Recursively merge X_1 and Y_1 to form the first part of the output sequence. Recursively merge X_2 and Y_2 to form the second part of the output sequence.

The two sub-merges are independent and thus can be done in parallel. Listing 13.2 shows a Cilk Plus implementation, using the same argument convention as for `serial_merge`. For clarity, the code has been written using full recursion. It is a straightforward exercise to turn it into semi-iterative form as in Quicksort (Listing 8.11 on page 234).

A point sometimes overlooked in implementing parallel merge is retaining the stability condition so that if keys in X and Y are equal then X comes first in the output sequence. Doing so requires a slight asymmetry in the binary search:

- If splitting Y at element `*ym`, then elements of X equal to `*ym` go in X_1.
- If splitting X at element `*xm`, then elements of Y equal to `*xm` go in Y_2.

The distinction between `std::upper_bound` and `std::lower_bound` achieves this asymmetry.

Let N be the total length of the two sequences being merged. The recursion depth never exceeds about $2\lg(N/\texttt{MERGE\_CUT\_OFF})$. The qualification "about" is there because when the sequences do not divide evenly at every level of recursion the depth may be one deeper. The depth limit follows from the observation that each two consecutive levels of recursion cut the problem size about in half or better. Here is a sketch of the proof.

```
void parallel_merge( T* xs, T* xe, T* ys, T* ye, T* zs ) {
   const size_t MERGE_CUT_OFF = 2000;
   if( xe-xs + ye-ys <= MERGE_CUT_OFF ) {
      serial_merge(xs,xe,ys,ye,zs);
   } else {
      T *xm, *ym;
      if( xe-xs < ye-ys ) {
         ym = ys+(ye-ys)/2;
         xm = std::upper_bound(xs,xe,*ym);
      } else {
         xm = xs+(xe-xs)/2;
         ym = std::lower_bound(ys,ye,*xm);
      }
      T* zm = zs + (xm-xs) + (ym-ys);
      cilk_spawn parallel_merge( xs, xm, ys, ym, zs );
      /* nospawn */ parallel_merge( xm, xe, ym, ye, zm );
      // Implicit cilk_sync
   }
}
```

LISTING 13.2

Parallel merge in Cilk Plus.

Let N be the size of the initial problem and N' be the size of the biggest subproblem after two levels of recursion. For simplicity of exposition, only cases where sequences split evenly are considered. The point is to prove $N' \leq N/2$.

Let $|X|$ and $|Y|$ denote the length of X and Y, respectively. Without loss of generality, assume $|X| \geq |Y|$. There are two cases:

- The first level of recursion splits X and the second level splits Y. Both sequences have been halved, so $N' = N/2$.
- Both levels split X. This can only happen when $|X| \geq 2|Y|$. Since $|X| + |Y| = N$, we know $|Y| \leq N/3$. Therefore, $N' = |X|/4 + |Y| = (N - |Y|)/4 + |Y| = N/4 + (3/4)|Y| \leq N/4 + (3/4)(N/3) = N/2$.

13.1.1 TBB Parallel Merge

The parallel merge code can be translated to Intel TBB by implementing the parallel **fork–join** with `tbb::parallel_invoke` and **lambda expressions** instead of `cilk_spawn`. Listing 13.3 shows the rewritten lines.

13.1.2 Work and Span of Parallel Merge

The work and span for `parallel_merge` cannot be found using the recurrence solutions given in Section 8.6, because the recurrences have the wrong form. Instead, other methods have to be used. The

```
tbb::parallel_invoke(
    [=]{parallel_merge( xs, xm, ys, ym, zs );},
    [=]{parallel_merge( xm, xe, ym, ye, zm );} );
```

LISTING 13.3

Converting parallel merge from Cilk Plus to TBB. The lines shown are the equivalent of the `cilk_spawn`/`cilk_sync` portion of Listing 13.2.

span is analyzed first because it is simpler. In the worst case each two levels of recursion halve the problem size. Each level performs a binary search, so there are two binary searches per halving of the problem size. Hence, the span for `parallel_merge` has the recurrence:

$$T_\infty(N) = 2\Theta(\lg N) + T_\infty(N/2).$$

The 2 before the Θ can be immediately eliminated since it is mathematically swallowed by the Θ. Expanding the remaining recurrence and factoring the Θ yields:

$$T_\infty(N) = \Theta[\lg(N) + \lg(N/2) + \lg(N/4) + \lg(N/3) + \cdots + 1],$$

which is equivalent to:

$$T_\infty(N) = \Theta[(\lg N) + (\lg N) - 1 + (\lg N) - 2 + (\lg N) - 3) + \cdots + 1].$$

The right side is a decreasing arithmetic series starting at $\lg N$ and hence has the quadratic sum $\Theta((\lg N)^2)$.

The recurrence for work is:

$$T_1(N) = \Theta(\lg N) + 2T_1(N/2),$$

which after expanding the recurrence and factoring the Θ becomes:

$$T_1(N) = \Theta[\lg N + 2\lg(N/2) + 4\lg(N/4) + 8\lg(N/8) + \cdots].$$

Substitute $K = \lg N$ to get:

$$T_1(N) = \Theta[K + 2(K-1) + 4(K-2) + 8(K-3) + \cdots + 2^{K-1}(1)].$$

A visual trick to solving this series is to note that the terms are the column sums of the following $K \times K$ triangular matrix:

$$\begin{array}{cccccc}
1 \\
1 & 2 \\
1 & 2 & 4 \\
1 & 2 & 4 & 8 \\
\vdots & \vdots & \vdots & \vdots & \ddots \\
1 & 2 & 4 & 8 & \ldots & 2^{K-1}
\end{array}$$

Each row is a geometric series. The sum of the ith row is $2^i - 1$. Summing all K rows is $2^{K+1} - 2 - k$. Thus $T_1(N) = \Theta(2^K) = \Theta(N)$.

Remarkably, both serial and parallel merging have the same asymptotic work! An intuition for this is that the binary search time $\Theta(\lg N)$ is dominated by the $\Omega(N)$ time required to touch each element in the sub-merges, and hence makes no additional contribution to the asymptotic work. However, the constant factors hiding behind Θ deserve some attention. If base case is as small as possible (that is, `MERGE_CUT_OFF`$= 1$), then `parallel_merge` requires about 50% more comparisons than `serial_merge`. The 50% tax, however, rapidly diminishes if `MERGE_CUT_OFF` is raised. Raising the latter to 16 reduces the tax to about 6%. In practice, `MERGE_CUT_OFF` needs to be higher anyway to amortize parallel scheduling overheads, and then the tax practically disappears. Indeed, a more significant concern becomes the less cache-friendly behavior of the search-based merging compared to serial merging, but this too is amortized away by a sufficiently large `MERGE_CUT_OFF`.

To summarize, the work and span for parallel merge are:

$$T_1(N) = \Theta(N), \tag{13.1}$$

$$T_\infty(N) = \Theta(\lg^2 N), \tag{13.2}$$

which indicates a theoretical speedup of $\Theta\left(\frac{N}{\lg^2 N}\right)$, assuming that memory bandwidth is not a bottleneck.

13.2 PARALLEL MERGE SORT

Turning the sketch (at the chapter opening) of parallel merge sort into code is straightforward. However, merge sort is not an in-place sort because the merge step cannot easily be done in place. Temporary storage is needed. Furthermore, to minimize copying, the merge steps should alternate between merging from the original buffer into the temporary buffer and vice versa. One way to do this is to define parallel merge sort with a flag argument `inplace` that controls the destination of the sorted sequence:

- If `inplace` is true, the destination is the original input buffer.
- Otherwise, the destination is the temporary buffer.

In both cases, the non-destination buffer is used as scratch space.

Listing 13.4 shows a Cilk implementation. It sorts a sequence defined by the half-open interval [`xs`,`xe`). Argument `zs` should point to a buffer of length `xe`–`xs`.

The parallel base case uses a stable sort in order to preserve stability of the overall sort. The flag `inplace` flipflops at each recursion level. When sorting in place, the subsorts copy into the temporary buffer, and the `parallel_merge` operation copies the items back into the original buffer. When not sorting in place, the subsorts are done in place, and the `parallel_merge` copies the items into the destination buffer.

A TBB implementation is similar, except that the fork–join is implemented with `parallel_invoke` instead of the Cilk keywords, similar as discussed for the TBB implementation of `parallel_merge` (Section 13.1).

```
void parallel_merge_sort( T* xs, T* xe, T* zs, bool inplace ) {
   const size_t SORT_CUT_OFF = 500;
   if( xe-xs<=SORT_CUT_OFF ) {
      std::stable_sort( xs, xe );
      if( !inplace )
         std::move( xs, xe, zs );
  } else {
      T* xm = xs + (xe-xs)/2;
      T* zm = zs + (xm-xs);
      T* ze = zs + (xe-xs);
      cilk_spawn parallel_merge_sort( xs, xm, zs, !inplace );
      /* nospawn */ parallel_merge_sort( xm, xe, zm, !inplace );
      cilk_sync;
      if( inplace )
         parallel_merge( zs, zm, zm, ze, xs );
      else
         parallel_merge( xs, xm, xm, xe, zs );
  }
}
```

LISTING 13.4

Parallel merge sort in Cilk Plus.

13.2.1 Work and Span of Merge Sort

Let N be the length of the input sequence. The recurrences for work and span are:

$$T_1(N) = \Theta(N) + 2T_1(N/2),$$
$$T_\infty(N) = \Theta(\lg^2 N) + T_\infty(N/2).$$

The recurrence for T_1 is case 2 of the Master method (see Section 8.6) and has the closed form solution $T_1(N) = \Theta(N \lg N)$. This is the same as for a serial merge sort, which is not surprising since the constituent components of `parallel_merge_sort` have the same asymptotic work bounds as their counterparts in the serial algorithm. The solution T_∞ can be found by observing that if $K = \lg N$, then the recurrence expands to the series:

$$K^2 + (K-1)^2 + (K-2)^2 + \cdots + 1,$$

which has a cubic sum. Hence, $T_\infty(N) = \Theta(\lg^3 N)$.

Thus, the asymptotic speedup for `parallel_merge_sort` is $\Theta\left(\frac{N \lg N}{\lg^3 N}\right) = \Theta\left(\frac{N}{\lg^2 N}\right)$. This suggests that, given about a million keys, on the order of a thousand processors might be used profitably if memory bandwidth does not become a constraint.

13.3 SUMMARY

Merge sort is a classic divide-and-conquer algorithm that lends itself to the parallel fork–join pattern and is easily written in Cilk or TBB. However, for good speedup, the merge step must also be parallelized, because otherwise it takes linear time. The merge step can be parallelized by divide-and-conquer, where the divide step generates two submerges that can be run in parallel.