Odd-even transposition sort

- Odd-even transposition sort is a sorting algorithm that's similar to bubble sort, but that has considerably more opportunities for parallelism.
- serial odd-even transposition sort can be implemented as follows:

```
for (phase = 0; phase < n; phase++)
        if (phase % 2 == 0)
for (i = 1; i < n; i += 2)
        if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
        else
                for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

- The list a stores n ints, and the algorithm sorts them into increasing order.
- During an "even phase" (phase % 2 == 0), each odd-subscripted element, a[i], is compared to the element to its "left," a[i-1], and if they're out of order, they're swapped.
- During an "odd" phase, each odd-subscripted element is compared to the element to its right, and if they're out of order, they're swapped. A theorem guarantees that after ***n*** phases, the list will be sorted.

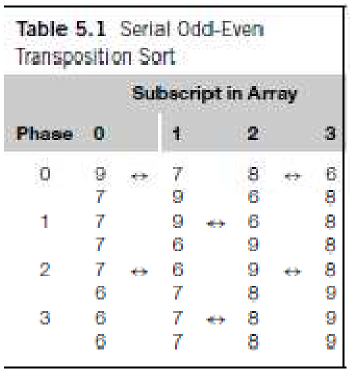


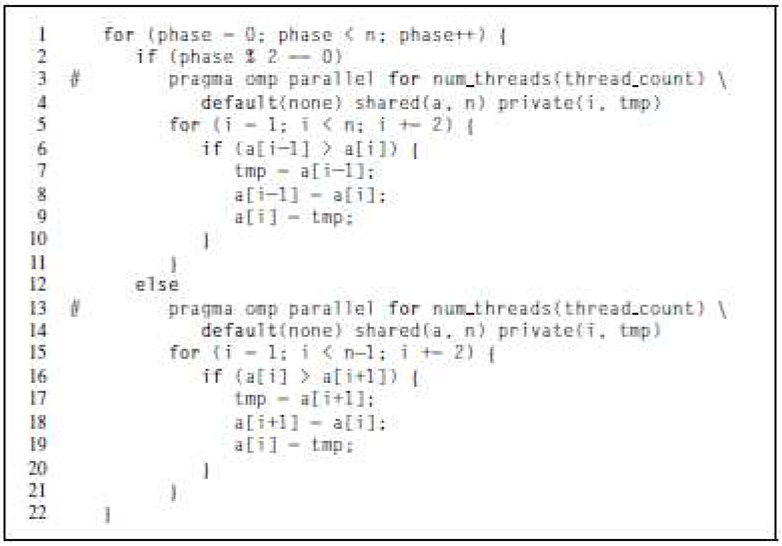
Table 5.1 Serial Odd-Even Transposition Sort

	Subscript in Array						
Phase	0		1		2		3
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

- As a brief example, suppose a = [9, 7, 8, 6]. Then the phases are shown in Table 5.1. In this case, the final phase wasn't necessary, but the algorithm doesn'tbother checking whether the list is already sorted before carrying out each phase.It's not hard to see that the outer loop has a loop-carried dependence.
- As an example, suppose as before that a = [9, 7, 8, 6]. Then in phase 0 the inner loop will compare elements in the pairs .(9,7) and (8,6), and both pairs are swapped. So for phase 1 the list should be [7, 9, 6, 8], and during phase 1 the elements in the pair .(9,6) should be compared and swapped. However, if phase 0 and phase 1 are executed simultaneously, the pair that's checked in phase 1 might be .(7,8), which is in order. Furthermore, it's not clear how one might eliminate this loop-carried dependence, so it would appear that parallelizing the outer for loop isn't an option.

- The inner ***for*** loops, however, don't appear to have any loop-carried dependences. For example, in an even phase loop, variable i will be odd, so for two distinct values of i, say i = j and i = k, the pairs (j-1, j) and (k-1, k) will be be disjoint. The comparison and possible swaps of the pairs (a[j-1], a[j]) and (a[k-1], a[k]) can therefore proceed simultaneously.

- OpenMP implementation of odd-even sort



```
for (phase = 0; phase < n; phase++) {
   if (phase % 2 == 0)
#     pragma omp parallel for num_threads(thread_count) \
         default(none) shared(a, n) private(i, tmp)
      for (i = 1; i < n; i += 2) {
         if (a[i-1] > a[i]) {
            tmp = a[i-1];
            a[i-1] = a[i];
            a[i] = tmp;
         }
      }
   else
#     pragma omp parallel for num_threads(thread_count) \
         default(none) shared(a, n) private(i, tmp)
      for (i = 1; i < n-1; i += 2) {
         if (a[i] > a[i+1]) {
            tmp = a[i+1];
            a[i+1] = a[i];
            a[i] = tmp;
         }
      }
}
```

Fig: First OpenMP implementation of odd-even sort

- First, although any iteration of, say, one even phase doesn't depend on any other iteration of that phase, we've already noted that this is not the case for iterations in ***phase p*** and ***phase p+1***. We need to be sure that all the threads have finished phase p before any thread starts ***phase p+1***. However, like the ***parallel*** directive, the ***parallel for*** directive has an implicit barrier at the end of the loop, so none of the threads will proceed to the next phase, ***phase p+1***, until all of the threads have completed the current phase***, phase p***.

- A second potential problem is the overhead associated with forking and joining the threads. The OpenMP implementation may fork and join thread count threads on each pass through the body of the outer loop.

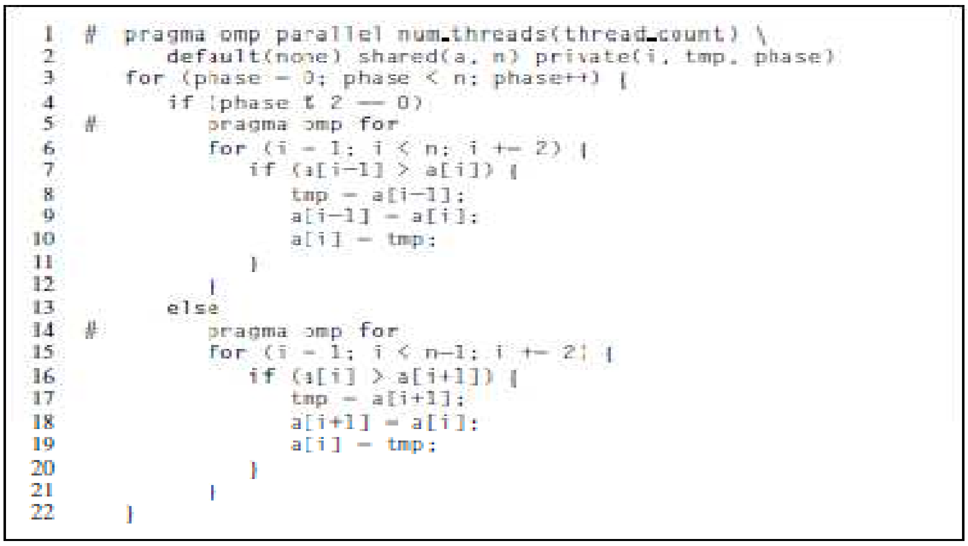
- Each time we execute one of the inner loops, we use the same number of threads, so it would seem to be superior to fork the threads once and reuse the same team of threads for each execution of the inner loops. Not surprisingly, OpenMP provides directives that allow us to do just this.

- The first row of Table 5.2 shows run-times for 1, 2, 3, and 4 threads on one of our systems when the input list contained 20,000 elements.

- We can fork our team of thread_count threads before the outer loop with a ***parallel*** directive. Then, rather than forking a new team of threads with each execution of one of the inner loops, we use a for directive, which tells OpenMP to parallelize the for loop with the existing team of threads.

This modification to the original OpenMP implementation is shown in Program below.

- The ***for*** directive, unlike the ***parallel for*** directive, doesn't fork any threads. It uses whatever threads have already been forked in the enclosing ***parallel*** block.



```
#   pragma omp parallel num_threads(thread_count) \
       default(none) shared(a, n) private(i, tmp, phase)
    for (phase = 0; phase < n; phase++) {
       if (phase % 2 == 0)
#         pragma omp for
          for (i = 1; i < n; i += 2) {
             if (a[i-1] > a[i]) {
                tmp = a[i-1];
                a[i-1] = a[i];
                a[i] = tmp;
             }
          }
       else
#         pragma omp for
          for (i = 1; i < n-1; i += 2) {
             if (a[i] > a[i+1]) {
                tmp = a[i+1];
                a[i+1] = a[i];
                a[i] = tmp;
             }
          }
    }
```

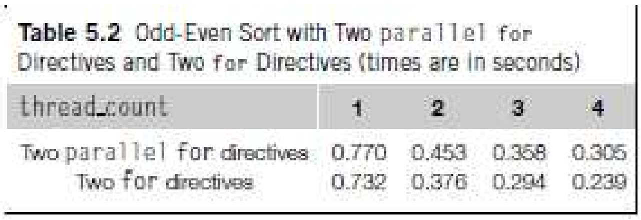


Table 5.2 Odd-Even Sort with Two `parallel for` Directives and Two `for` Directives (times are in seconds)

`thread_count`	1	2	3	4
Two `parallel for` directives	0.770	0.453	0.358	0.305
Two `for` directives	0.732	0.376	0.294	0.239

- Run-times for this second version of odd-even sort are in the second row of Table 5.2. When we're using two or more threads, the version that uses two for directivesis at least 17% faster than the version that uses two parallel for directives, sofor this system the slight effort involved in making the change is well worth it.