Topics in Parallel and Distributed Computing:

Introducing Concurrency in Undergraduate Courses[1,2]

Chapter 9

Parallel Computing in a Python-Based Computer Science Course

Thomas H. Cormen

Dartmouth College

thc@cs.dartmouth.edu

[1]How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*, 1st Edition, Morgan Kaufmann, ISBN : 9780128038994, Pages: 360.

[2]Free preprint version of the CDER book: `http://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book`.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 9

PARALLEL COMPUTING IN A PYTHON-BASED COMPUTER SCIENCE COURSE

Thomas H. Cormen

Dartmouth College

thc@cs.dartmouth.edu

ABSTRACT

Drawing heavily from Blelloch's work on the vector model and scan operations, we work our way up to performing a parallel version of the quicksort algorithm on a shared-memory machine with p processors. We first see how to perform reduction and scan operations in parallel. We then examine parallel meld and permute operations, which lead to unsegmented partitioning in parallel. We then introduce segmented operations and modify our partitioning procedure to work with segmented operations, leading to a fully parallel version of quicksort.

Python code appears for most, but not all, of the operations in this chapter. We omit Python code for partitioning and for the full parallel quicksort because these programs make for excellent exercises.

The Python code we use does not itself run in parallel. In order to run code in parallel with Python, we would have to lock into a particular Python library. The concepts behind parallel programming are more important than the exact means to achieve parallelism. Therefore, we use Python for-loops and indicate which can be run in parallel and which cannot. An outer for-loop (i.e., a for-loop that is not nested in another for-loop) that runs for at most p iterations is parallelizable and therefore can run on the p processors concurrently. A

for-loop that is nested within a parallelizable for-loop is not parallelizable; all of its iterations run sequentially on a given processor. Instructors who wish to have their students actually run the code in parallel may translate our code to their favorite Python library.

For sufficiently advanced courses, this chapter includes a simple analysis of the tradeoffs between sequential and parallel versions of reduction and scan operations.

Relevant core courses: This material applies to a CS 1 or CS 2 course.

Relevant PDC topics: Shared memory; vector model; reduction; scan; segmented operations; analysis of algorithms; sorting; quicksort.

Learning outcomes:

1. Break down a complex computation (quicksort) into individual parallel operations.
2. See how to perform reductions and scans in parallel.
3. Understand costs of parallel operations in a shared-memory environment.
4. Gain practice in using parallel operations.
5. Understand how to analyze an asymptotic running time that is the sum of two different terms, each of which may dominate.
6. If students are sufficiently advanced, analyze the tradeoffs between sequential and parallel computing.
7. Learn how to write code in a parallel style using Python.
8. If necessary: Learn about Python's lambda construct and how to design Python functions that take a variable number of parameters. This material is not covered in this chapter, as there are many resources elsewhere for it.

Suggested student background: Students should know

1. Basic aspects of Python programming.

2. How to analyze simple code using O-notation.
3. How to understand, write, and analyze running times of recursive functions.
4. (Recommended:) Python's `lambda` construct and the `*` operator to gather and scatter function parameters.

Context for use: This material is designed for a CS 1 or CS 2 course. Some of the material might be too advanced for certain CS 1 courses, depending on the students. For example, at Dartmouth, our CS 1 course draws from the general student population, and not from just the technically oriented students. Several of the non-technical students have had trouble understanding this material. A course that draws primarily technical students should be able to incorporate this material.

The programming material here is based on Python, because that's what we teach in our CS 1 course at Dartmouth. The material should be adaptable to other programming languages, but the treatment here sticks to Python.

The model is a shared-memory machine with multiple processors, but the programming style derives from Blelloch's work on the vector model and scan operations [1212]. All operations are on Python lists, and scan operations (also known as prefix computations) figure heavily.

All code is provided, except the code for unsegmented parallel partitioning and the code for parallel quicksort; the code for these functions is provided in a supplement. Students should implement these functions in an assignment. Implementing unsegmented parallel partitioning is a good warmup assignment for implementing full parallel quicksort.

9.1 Parallel programming

Suppose that we want to write code that takes advantage of a computer with multiple processors and a shared memory, which is the simplest parallel computer to program. Computer scientists have developed several ways to program such machines. Rather than fixing on any particular one, we'll write code that "looks" parallel (we'll see what that means a little later) and would be easy to convert to true parallel code on a real parallel system, but that runs sequentially on our own laptops.

Let's imagine that we have at our disposal p processors, each of which can access a common, shared memory. We'll assume that each processor can access any variable, any object, any item of any list, etc. To avoid the havoc that would occur when multiple processors try concurrently to change the value of the same variable, we will consider only programs in which that does not occur. In particular, we will partition our data among the p processors, so that for an n-item list, each processor is responsible for n/p items. We'll always assume that if $n > p$, then n is a multiple of p, and we'll consistently use m to equal n/p.

9.1.1 Parallelizable loops

From a programming point of view, we will call a for-loop that runs for at most p iterations, in which the result of each iteration has no effect on other iterations, ***parallelizable***. We can imagine each of the at most p iterations running on its own processor *in parallel* (i.e., concurrently), so that if each iteration takes time t, all the iterations together take time only $O(t)$, rather than the $O(pt)$ time they would take if we ran them sequentially. For example, if each iteration of a parallelizable for-loop takes constant time, then the for-loop takes constant time when run in parallel. If that for-loop is operating on a list of n items, then in order for all iterations to run in parallel, we must have $n \leq p$. On the other hand, if $n > p$, then a parallelizable for-loop has to work on n/p items per processor, so that the best we could hope for from the parallelizable for-loop would be $O(n/p)$ time (or, equivalently,

$O(m)$ time).

Let's make this idea more concrete. Suppose we have three lists, `a`, `b`, and `c`, each of length p, and we want to add the corresponding items of `a` and `b` into the corresponding item of `c`. Here's a simple for-loop that does the job:

```
for i in range(p):
    c[i] = a[i] + b[i]
```

This for-loop is parallelizable, since each iteration has no effect on other iterations. Thus, because each iteration takes constant time, if we have p processors, then this for-loop runs in constant time, i.e., $O(1)$ time, in parallel.

Now suppose that each of the lists `a`, `b`, and `c` has n items, where $n > p$, so that $m > 1$. Then here are nested for-loops that add the corresponding items of `a` and `b` into the corresponding item of `c`:

```
# This outer for-loop is parallelizable.
for i in range(p):
    # This inner for-loop is not parallelizable.
    for j in range(m):
      c[i*m + j] = a[i*m + j] + b[i*m + j]
```

The idea here is that processor 0 is responsible for the m indices 0 to $m-1$, processor 1 is responsible for the m indices m to $2m-1$, processor 2 is responsible for the m indices $2m$ to $3m-1$, and so on. More generally, processor i, for $i = 0, 1, 2, \ldots, p-1$, is responsible for the m indices $im, im+1, im+2, \ldots, im+(m-1)$. We can parallelize the outer for-loop, so that each processor performs its $m = n/p$ iterations of the inner for-loop in parallel with the other processors, but these m iterations of the inner for-loop run sequentially on the processor. Since each iteration of the inner for-loop takes constant time, this computation runs in $O(m)$ time in parallel.

9.1.2 Barriers

Many parallel computations get to a point where no processor should proceed until all processors have arrived at the same point. By analogy, think of what happens when several people dine together at a restaurant: nobody starts the next course until everyone

has finished the current course. When all processors need to arrive at the same point in a parallel computation, we call that a ***barrier***. We'll assume that we have a function `barrier`, which takes no parameters, and returns only once all processors have called it.

There are several ways to implement a barrier, and we won't go into them here. We will count the number calls to `barrier` when we analyze parallel running times, however. We'll assume that the time for each barrier call is given by the parameter β.

9.1.3 Outline

In this chapter, we will work our way up to performing a parallel version of the quicksort algorithm [3, Chapter 7]. The approach we use draws heavily from Blelloch's work on the vector model and scan operations [1212].

We start by seeing how to perform reduction and scan operations in parallel. We then add parallel meld and permute operations, which give us the operations we need to partition a list in parallel, with a result similar to that produced by sequential quicksort. In order to perform all the recursive steps on quicksort, we introduce segmented operations, and then modify our partitioning procedure to work with segmented operations.

Python code appears for most, but not all, of the operations in this chapter. We omit Python code for partitioning and for the full parallel quicksort because these programs make for excellent exercises.

This chapter closes by analyzing the tradeoffs between sequential and parallel implementations of reduction and scan operations.

9.2 Parallel reduction

A ***reduction*** operation applies an associative operator to the items of a list, giving the single result of applying the operator. For example, a plus-reduction on a list `x` with n items gives the value `x[0]` + `x[1]` + `x[2]` + $\cdots$ + `x[n-2]` + `x[n-1]`. (Recall that an associative operator—let's call it $\oplus$ to keep it generic—satisfies the property that $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ for any operands a, b, and c.)

At first glance, you might think that computing the plus-reduction on a list is inherently sequential, because we have to first sum `x[0]` + `x[1]`, and then add in `x[2]`, then add in `x[3]`, and so on. But because the operation we're performing—addition in this case—is associative, we can add values in other orders. For example, we could first compute `x[0]` + `x[1]` + `x[2]` + $\cdots$ + `x[n/2-1]`, and we could compute `x[n/2]` + `x[n/2 + 1]` + `x[n/2 + 2]` + $\cdots$ + `x[n-1]`, and we could then sum these two results to give the plus-reduction of the entire list. In other words, instead of parenthesizing our additions as ($(\cdots(($`x[0]` + `x[1]`$)$ + `x[2]`$)$ + $\cdots$ + `x[n-2]`$)$ + `x[n-1]`$)$, we could parenthesize them as ($($`x[0]` + `x[1]` + `x[2]` + $\cdots$ + `x[n/2-1]`$)$ + ($($`x[n/2]`+`x[n/2 + 1]`+`x[n/2 + 2]`+$\cdots$+`x[n-1]`$)$, where we can further parenthesize the sums ($($`x[0]` + `x[1]` + `x[2]` + $\cdots$ + `x[n/2-1]`$)$ and ($($`x[n/2]` + `x[n/2 + 1]` + `x[n/2 + 2]` + $\cdots$ + `x[n-1]`$)$ however we choose.

In our code, we will use two global variables:

- `p`: the number of processors, and
- `m`: the number of items per processor, i.e., n/p.

9.2.1 Reducing in parallel when $n \leq p$

The function `simple_reduce` in Listing 9.1 shows how we can compute a reduction on a list `x` with $n \leq p$ items in parallel. The parameter `op` is a Python function that computes some associative function of two parameters, and the parameter `ident` is the identity for the operation `op`. For example, if `op` is a function that adds two numbers, then `ident` would be 0, and if `op` is a function that multiplies two numbers, then `ident` would be 1. We need the identity in case the list is empty.

Instead of breaking the problem down as we did above, `simple_reduce` works as follows. We create a new list, `subproblem`, of half the size. That is, `subproblem` has size $n/2$, represented by the variable `half`, except that we round up when n is odd by computing `half = (n+1)/ 2`, using integer division. We then combine consecutive pairs of values of `x` into `subproblem` using the operator `op`. For example, if `op` is addition, then `subproblem[0]` equals `x[0] + x[1]`, `subproblem[1]` equals `x[2] + x[3]`, `subproblem[2]` equals `x[4] + x[5]`,

Listing 9.1: The function simple_reduce.

```
# Return the reduction of all values in an n-item list x.  The
# parameter op is a two-parameter function for the reduction.  The
# parameter ident is the identity for op, in case the list is empty.
# Assumes that n <= p, where p is the number of processors.  The
# reduction takes O(beta log n) time in parallel.
def simple_reduce(x, op, ident):
    n = len(x)

    if n == 0:
        return ident    # empty list
    elif n == 1:
        return x[0]     # base case: only one value
    else:
        # Create a new list, subproblem, of half the size, rounding
        # up if n is odd.
        half = (n+1) / 2
        subproblem = [None] * half

        # Using a parallelizable for-loop, combine pairs of values
        # in x into subproblem.
        for i in range(half-1):
            subproblem[i] = op(x[2*i], x[2*i+1])

        # Need special code to handle the last one or two values in
        # x, in case n is odd.
        if n % 2 == 0:
            subproblem[half-1] = op(x[2*half-2], x[2*half-1])
        else:
            subproblem[half-1] = x[2*half-2]

        barrier()

        # Return the reduction of the subproblem.  This recursive
        # call is on a problem of half the size.
        return simple_reduce(subproblem, op, ident)
```

and so on. `simple_reduce` uses special code to handle the case when n is odd, in which case the last item of `subproblem` gets just the last value of `x`. Observe that the reduction of `x` must equal the reduction of `subproblem`. Hence, we simply recurse on `subproblem`, passing `subproblem` to the recursive call of `simple_reduce`, and we return the result of the recursive call. The base cases occur when the length of the list is 1, in which case we just return the value of `x[0]`, and when the list is empty, where we return the identity `ident` for the operation.

Pictorially, Figure 9.1 shows how a single step of the recursion works on a list with the values 2, 3, 4, 2, 3, 6, 5, 2 and the operation of addition, and Figure 9.2 shows how the entire recursion unfolds.

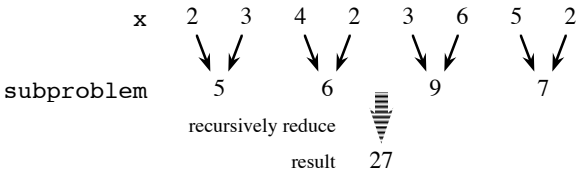

Figure (9.1) A single recursive call in `simple_reduce`.

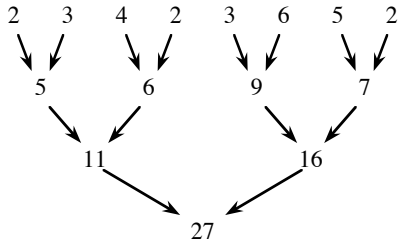

Figure (9.2) All recursive calls in `simple_reduce`.

To analyze this function, we observe that the for-loop is parallelizable. Why? The computation in each iteration—filling in `subproblem[i]`—is independent of the computation in all other iterations. Since $n \leq p$ and we have p processors, we can execute this for-loop in constant time in parallel. Thus, each recursive call takes constant time in parallel,

plus the time for the recursive call. The number of recursive calls is $O(\log n)$, since we (approximately) halve n in each call. (After $O(\log n)$ recursions, we get down to a problem size of 1. Rounding up $n/2$ when n is odd doesn't affect the number of recursive calls in terms of O-notation.)

Notice the call to the `barrier` function immediately preceding the recursive call. Why do we need it? Each processor must have computed its portion of the reduction before that result is given to a recursive call. Referring to Figure 9.2, in the first call of `simple_reduce`, processor 0 adds 2 and 3, producing 5; processor 1 adds 4 and 2, producing 6; processor 2 adds 3 and 6, producing 9; and processor 3 adds 5 and 2, producing 7. In the second call of `simple_reduce` (the first call made recursively), processor 0 adds 5 and 6, producing 11; and processor 1 adds 9 and 7, producing 16. Now, in order for processor 0 to add 5 and 6, the values 5 and 6 must have already been produced. Although processor 0 produces the 5, it's processor 1 that produces the 6, and so processor 0 should not execute the recursive call until processor 1 has completed its original call. Similarly, for processor 1 to add 9 and 7, these values must have been produced by processors 2 and 3 in the original call. The call of `barrier` ensures that all processors have produced the values needed for a recursive call before the recursive call occurs.

We have $O(\log n)$ recursive calls, and each recursive call takes constant time per processor, plus β for the call to `barrier`. Thus, we can perform a reduction operation in parallel in $O(\beta \log n)$ time.

9.2.2 Reducing in parallel when $n > p$

Now, what about when $n > p$? The `reduce` function, in Listing 9.2, handles this case. *Note:* `reduce` is a built-in Python function that performs a reduction operation sequentially on a Python list; by redefining it, we perform the reduction in parallel. The idea is to create a list `reductions` of length p and have each processor combine its own portion of m items into one position of `reductions`. Then we can call `simple_reduce` on `reductions` to produce the reduction of the entire list.

Listing 9.2: The functions `reduce` and `reduce_one`.

```
# Return the reduction of all values in an n-item list x.  The
# parameter op is a two-parameter function for the reduction.  The
# parameter ident is the identity for op, in case the list is empty.
# Assumes that there are p processors, n > p, m = n / p, and
# processor i works on x[i*m] through x[i*m + m-1]. Takes O(m + beta
# log p) time in parallel.
def reduce(x, op, ident):
    # Create a list for reducing the values in each processor.
    reductions = [ident] * p

    # Using a parallelizable outer for-loop, reduce the values in
    # each processor's portion of x.  With p processors, this loop
    # takes O(m) time.
    do_in_parallel(reduce_one, reductions, x, op)
    barrier()

    # Now we have a list of p reductions, so just return what the
    # recursive reduce function returns.  This call takes O(beta log
    # p) time in parallel with p processors.
    return simple_reduce(reductions, op, ident)

# Perform one step of a reduction within processor i, combining the
# jth item in processor i into reductions[i].
def reduce_one(i, j, reductions, x, op):
    reductions[i] = op(reductions[i], x[i*m + j])
```

Let's look at the `do_in_parallel` function in Listing 9.3. It takes as its first parameter a function `f`, and then it takes a tuple `params`, which comprises the gathered parameters to the function `f`. `do_in_parallel` runs two nested loops. The outer loop, with the header `for i in range(p)`, emulates running in parallel on p processors. The inner loop, with the header `for j in range(m)`, works on each item in processor i. By "working on each item," we mean calling the function `f` on each item j in processor i. `params` are just the parameters to `f`.

Now we can understand how `reduce` works. By passing to `do_in_parallel` the parameters `reduce_one`, `reductions`, `x`, `op`, it calls `reduce_one` once for each combination of `i`

Listing 9.3: The function `do_in_parallel`.

```
# Function for performing a computation in parallel.  Assumes that
# there are p processors that can operate independently and in
# parallel, and that each processor has m items from each list it
# operates on.  The parameter f is the name of a function that
# operates on item j in processor i, and params give the parameters
# to f.  params should not include i and j, because these variables
# are loop variables within do_in_parallel.
def in_parallel(f, *params):
    # Call f for each i and j.  The outer loop on i can run in
    # parallel, but the inner loop on j cannot.
    for i in range(p):
        for j in range(m):
            f(i, j, *params)
```

and `j`, where $i = 0, 1, 2, \ldots, p-1$ and $j = 0, 1, 2, \ldots, m-1$. And `reduce_one` just combines `x[i*m + j]` into `reductions[i]`, using the operator `op`. For example, if `op` is addition, then `reduce_one` adds `x[i*m + j]` into `reductions[i]`. Get used to the expression `something[i*m + j]`; it denotes the jth item in processor i's portion of the list `something`. (Recall that processor 0 gets the first m items, processor 1 gets the next m items, and so on.)

After the call to `do_in_parallel` has completed, `reductions[i]` has the reduction of all the values for which processor i is responsible: `x[i*m]` through `x[i*m + (m-1)]`. Remember that `reductions` has exactly p items, which we need to reduce. After calling `barrier` to make sure that all processors have computed their own reductions, a call to `simple_reduce` does the trick, and `reduce` returns what `simple_reduce` returns.

Now we can analyze the parallel running time of `reduce`. `do_in_parallel` takes $O(m)$ parallel time, since each processor can combine its m items in $O(m)$ time. Then there's a single call to `barrier`, taking $O(\beta)$ time. As we've seen, the call to `simple_reduce` takes $O(\beta \log p)$ time. The total parallel time for `reduce`, therefore, comes to $O(m + \beta + \beta \log p)$. The middle term, β, is dominated by the $\beta \log p$ term, and so we can drop it. Remembering

that $m = n/p$, we have a parallel running time of $O(n/p + \beta \log p)$. When $n/p > \beta \log_2 p$, the n/p term dominates the running time, and we get the best parallel speedup possible. When $\beta \log_2 p > n/p$, then n/p must be pretty small: divide both sides by β and then raise 2 to both sides, getting $p > 2^{n/(\beta p)}$, so we're not too disappointed that the running time is not $O(n/p)$; $O(\beta \log p)$ is still darned good.

Going back to the code for a moment, notice that although the function `reduce_one` has parameters `i` and `j`, we do not include these parameters in the call to `do_in_parallel`. That's because `do_in_parallel` provides them as the parameters to its nested loops. We will see this pattern frequently in the parallel code that we develop.

9.3 Parallel scanning

A ***scan*** operation is somewhat like a reduction in that we wish to apply an associative operator—again, let's call it $\oplus$—to all items in a list. Unlike a reduction, which produces one answer, a scan operation on an n-item list `x[0..n-1]` produces n answers. In particular, if the result of the scan operation is in the n-item list `result`, then `result[i]` equals `x[0]` $\oplus$ `x[1]` $\oplus$ `x[2]` $\oplus \cdots \oplus$ `x[i-2]` $\oplus$ `x[i-1]`. That is, the ith position of the result should hold the combination of the first $i-1$ items of `x`. What about `result[0]`? It should hold the identity for the associative operator. For example, in a plus-scan, `result[0]` should equal 0.

We call this type of scan an ***exclusive scan***, since the value of `result[i]` does not include `x[i]`. In an ***inclusive scan***, `result[i]` equals the sum of the first i items (not $i-1$) of `x`. We focus on the exclusive scan for two reasons. First, if we have the result of an exclusive scan, it's easy enough to combine `x[i]` into each position i to get the result of an inclusive scan. We cannot necessarily go backwards, because not all associative operators have an inverse for every operand. (For example, matrix multiplication is associative, but if A, B, and C are matrices, and C equals the matrix product $A \times B$, we cannot necessarily determine B given A and C, since A might not be invertible or even square.) Second, exclusive scans have more practical applications than inclusive scans. We'll see good applications of exclusive scans later.

Now, *this* operation looks inherently sequential. It would certainly be easy enough to compute it sequentially, given the identity `ident` for the function `op`:

```
result[0] = ident

for i in range(1, n):
    result[i] = op(result[i-1], x[i-1])
```

Seeing this computation makes us think that we need to compute `result[i-1]` before computing `result[i]`. Therefore, we would think that we have to compute `result[0]`, then `result[1]`, then `result[2]`, and so on. That's a sequential computation, and it would take $O(n)$ time.

9.3.1 Scanning in parallel when $n \leq p$

Believe it or not, we can perform a scan operation in $O(\beta \log n)$ time when $n \leq p$. In other words, scanning is no harder than reduction, even though scanning seems inherently sequential.

Let's take an example of a plus-scan, where the list `x` has the values 2, 3, 4, 2, 3, 6, 5, 2. We expect the result to be 0, 2, 5, 9, 11, 14, 20, 25. Figure 9.3 shows a schematic drawing of the process, and Listing 9.4 gives the Python code for the function `simple-scan`.

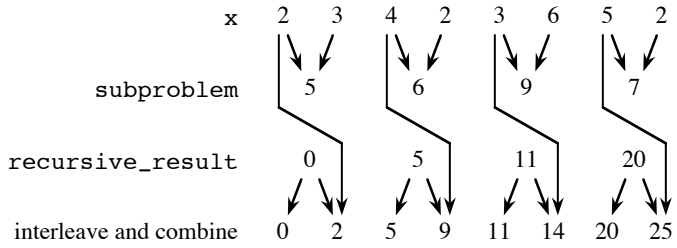

Figure (9.3) How to perform a scan operation in parallel.

We first combine consecutive pairs of values of `x` into a list `subproblem`, exactly as we did in the recursive `simple_reduce` function. As in `simple_reduce`, the list `subproblem` has $n/2$ items, rounding up if n is odd. In our example, `subproblem` has the values 5, 6, 9, 7. As

Listing 9.4: The function simple_scan

```
.
# Return a list of n values, whose ith value is the reduction of all
# values in the first i-1 items of x under an operation op, and
# whose 0th value is the identity ident for op. Assumes that n <= p,
# where p is the number of processors.  Takes O(beta log n) time in
# parallel.
def simple_scan(x, op, ident):
    # Create a list to hold the result of the scan.
    n = len(x)
    result = [None] * n

    if n == 1:
        result[0] = ident   # base case: only one value
    else:
        # Create a new list, subproblem, of half the size, rounding
        # up if n is odd.
        half = (n+1) / 2
        subproblem = [None] * half

        # Using a parallelizable for-loop, combine pairs of values
        # in x into subproblem.
        for i in range(half-1):
            subproblem[i] = op(x[2*i], x[2*i+1])

        # Need special code to handle the last one or two values in
        # x, in case n is odd.
        if n % 2 == 0:
            subproblem[half-1] = op(x[2*half-2], x[2*half-1])
        else:
            subproblem[half-1] = x[2*half-2]

        barrier()

        # Having created a subproblem of half the size, recurse on
        # it.
        recursive_result = simple_scan(subproblem, op, ident)

        # Now, for even values of i, recursive_result[i/2] holds the
        # reduction of all values from x[0] through x[i-1], so
        # that's what we want in result[i].  For odd values of i,
        # recursive_result[i/2] holds the reduction of all values
        # from x[0] through x[i-2], so that result[i] should get the
        # combination of recursive_result[i/2] and x[i-1]. This
        # for-loop is parallelizable.
        for i in range(n):
            if i % 2 == 0:
                result[i] = recursive_result[i/2]
            else:
                result[i] = op(recursive_result[i/2], x[i-1])

        barrier()

    return result
```

in `simple_reduce`, we recurse on `subproblem`, but now we call `simple_scan` recursively, not `simple_reduce`. The recursive call is on a problem half the size. We assign the result to a list named `recursive_result`. In our example, `recursive_result` has the values 0, 5, 11, 20.

Let's think about the situation after we return from the recursive call. For even indices i, `recursive_result[i/2]` holds the combination (in our example, the sum) of all values in `x[0]` through `x[i-1]`, and that's exactly what we want in the ith position of the result. Having created a list `result` of length n, we simply set `result[i] = recursive_result[i/2]` for even values of i. For odd values of i, `recursive_result[i/2]` holds the combination of all values in `x[0]` through `x[i-2]`, and so combining `x[i-1]` to `recursive_result[i/2]` gives us what we want in `result[i]`. We can think of filling in the `result` list as interleaving and combining, where we take the values in `recursive_result` and copy them into the even-indexed positions of `result`, and we combine the values of `recursive_result` to odd-indexed values of `x`, putting the sums into the odd-indexed positions of `result`.

When you look at the code in `parallel_scan`, you see that both the for-loops are parallelizable. Therefore, the analysis is the same as for the `parallel_reduce` function, and we get that the parallel running time is just $O(\beta \log n)$. We've succeeded in parallelizing the scan operation!

9.3.2 Scanning in parallel when $n > p$

The `scan` function in Listing 9.5 handles the case when $n > p$. We start by performing a scan on the values in each processor's portion of the list `x`, placing the results into the list `result`. Processor i is responsible for the sublist from `x[i*m]` through `x[i*m + m-1]`, and so we want `result[i*m + j]` to hold `x[i*m]` $\oplus$ `x[i*m + 1]` $\oplus$ `x[i*m + 2]` $\oplus \cdots \oplus$ `x[i*m + j-1]`, and `result[i*m]` should hold `ident`, the identity for the operation `op`. We perform this scan by calling `do_in_parallel`, passing the function `scan_body`. When `j` is 0, `scan_body` just puts `ident` into the jth position of `result` in processor i; otherwise, it combines the result in the $(j-1)$st position of processor i with the value of `x` in the $(j-1)$st position of processor i. As we saw with reducing, the call to `do_in_parallel` takes $O(m)$ time in parallel.

Listing 9.5: The functions scan, scan_body, and accumulate.

```
# Return a list of n values, whose ith value is the reduction of all
# values in the first i-1 items of x under an operator op, and whose
# 0th value is the identity ident.  Assumes that there are p
# processors, n > p, m = n / p, and processor i works on x[i*m]
# through x[i*m + m-1]. Takes O(m + beta log p) time in parallel.
def scan(x, op, ident):
    # Create a list to hold the result of the scan.
    n = m * p
    result = [ident] * (m * p)

    # In parallel, perform a scan on the values in each processor's
    # portion of x, so that result[i*m + j] has the reduction of the
    # values in x[i*m] through x[i*m + j-1] and result[i*m] has the
    # identity for op.  With p processors, this loop takes O(m)
    # time.
    do_in_parallel(scan_body, result, x, op, ident)

    # Create a list, prior_reductions, where prior_reductions[i]
    # will hold the reduction of all the values in processor i's
    # portion of x.
    prior_reductions = [None] * p

    # Using a parallelizable for-loop, fill in prior_reductions.
    for i in range(p):
        prior_reductions[i] = op(result[i*m + m-1], x[i*m + m-1])
    barrier()

    # Perform a scan on prior_reductions, so that processor_scan[i]
    # will hold the reduction of all values in portions of x
    # belonging to processors 0 through i-1.  This call takes O(beta
    # log p) time in parallel with p processors.
    processor_scan = simple_scan(prior_reductions, op, ident)

    # In parallel, combine processor_scan with the scan values
    # already computed.
    do_in_parallel(accumulate, result, processor_scan, op)
    barrier()

    return result

# Perform one step of a scan within processor i, combining the
# (j-1)st item in processor i into the result for the jth item in
# processor i.
def scan_body(i, j, result, x, op, ident):
    if j == 0:
        result[i*m + j] = ident
    else:
        result[i*m + j] = op(result[i*m + j-1], x[i*m + j-1])

# Given the result of the scan of all processors, "add" it into the
# jth result in processor i.
def accumulate(i, j, result, prior, op):
    result[i*m + j] = op(prior[i], result[i*m + j])
```

We're not done, however. We need to combine into each item of `result` the combination of values in all previous processors. In other words, we want to combine into `result[i*m + j]`, for $j = 0, 1, 2, \ldots, m-1$, the value `x[0]` $\oplus$ `x[1]` $\oplus$ `x[2]` $\oplus \cdots \oplus$ `x[i*m - 1]`. We create a list `prior_reductions` of length p, where `prior_reductions[i]` will eventually have the combination of all the values belonging to processors 0 through $i-1$. Initially, `prior_reductions[i]` gets the sum of all of processor i's values by assigning to it `result[i*m + m-1]`$\oplus$`x[i*m + m-1]` (recall that `result[i*m + m-1]` has `x[i*m]`$\oplus$`x[i*m + 1]`$\oplus$ `x[i*m + 2]` $\oplus \cdots \oplus$ `x[i*m + m-2]`, so that combining `x[i*m + m-1]` gives the combination of all of processor i's portion of `x`). We then call `simple_scan` on `prior_reductions`, assigning the result into `processor_scan`, so that `processor_scan[i]` holds the combination of all the values belonging to processors 0 through $i-1$. The call to `simple_scan` takes $O(\beta \log p)$ time.

Once we have `processor_scan` the way we want it, we just combine `processor_scan[i]` into each value in processor i's portion of `result`. Again, we need to perform a computation for each of the m items in each processor, and so we call `do_in_parallel`, passing the function `accumulate`, to do the job in $O(m)$ parallel time.

As in the `reduce` function, the total parallel running time of `scan` works out to $O(m + \beta \log p)$, or $O(n/p + \beta \log p)$.

9.3.3 Inclusive scans in parallel

As claimed above, once we have the result of an exclusive scan, we can perform an inclusive scan by simply combining the result of the exclusive scan for position i with the value in `x[i]`. The `inclusive_scan` function, in Listing 9.6, calls `scan` to perform an exclusive scan, and then, in parallel, it combines the result of the exclusive scan with the corresponding value in `x`.

Because the parallel time for a call of `tack_on` is $O(m)$, the `inclusive_scan` function takes $O(m + \beta \log p)$ time, just like `scan`.

Listing 9.6: The functions `inclusive_scan` and `tack_on`.

```
# Inclusive scan version of scan.  Returns a list of n values, whose
# ith value is the reduction of all values in the first i items of a
# under an operator op, and whose 0th value is the identity ident.
# Assumes that there are p processors, n > p, m = n / p, and
# processor i works on x[i*m] through x[i*m + m-1]. Takes O(m + beta
# log p) time in parallel.
def inclusive_scan(x, op, ident):
    result = scan(x, op, ident)
    do_in_parallel(tack_on, result, x, op)
    barrier()
    return result

# Perform one step of turning an exclusive scan into an inclusive
# scan.  Given the result of an exclusive scan in the jth item of
# result, processor i "adds" in the jth item of the original list x
# being scanned.
def tack_on(i, j, result, x, op):
    result[i*m + j] = op(result[i*m + j], x[i*m + j])
```

9.4 Copy-scans

In a ***copy-scan***, we copy the value in `x[0]` to every position in the result. We'll see that copy-scan is a surprisingly useful operation. For example, when partitioning for quicksort, we can use a copy-scan operation to copy the value of the pivot to all positions so that we can compare each value with the pivot in parallel. A copy-scan makes sense only as an inclusive scan, since there is no identity operator (and no reason that we'd ever want to copy the value in the 0th position to everywhere *except* the 0th position).

9.4.1 Copy-scanning in parallel when $n \leq p$

Unlike the other scan operations that we've seen, when $n \leq p$ we can perform a copy scan in constant parallel time, plus the time for a barrier. Why? Recall that we agreed we'd design our algorithms so that two processors won't try to *write* into the same memory

location simultaneously, but there's no reason that multiple processors cannot *read* from the same memory location simultaneously—as long as they all want to read the same value. And that's exactly what we want in a copy-scan. The `simple_copy_scan` function in Listing 9.7 runs in $O(\beta)$ parallel time.

Listing 9.7: The function `simple_copy_scan`.

```
# Return a list of n values, where each value is the value in x[0].
# Uses a parallelizable for-loop.  Assumes that n <= p, where p is
# the number of processors.  Takes O(beta) time in parallel.
def simple_copy_scan(x):
    n = len(x)
    result = [None] * n

    # This loop is parallelizable and runs in O(1) time if n <= p,
    # because we can read x[0] in parallel.
    for i in range(n):
        result[i] = x[0]
    barrier()

    return result
```

9.4.2 Copy-scanning in parallel when $n > p$

When $n > p$, we first perform a simple copy scan to copy the first value in processor 0 into all other processors, and then the usual loop structure copies this value throughout the list in parallel. The `copy_scan` function appears in Listing 9.8.

The `copy_scan` function takes $O(\beta)$ time in parallel for the call to `simple_copy_scan` and then $O(m + \beta)$ time in parallel for the call of `do_in_parallel`, for a total parallel time of $O(m + \beta)$.

Listing 9.8: The functions copy_scan and copy_body.

```
# Return a list of n values, where each value is the value in
# x[0]. Assumes that processor i works on a[i*m] through a[i*m +
# m-1].  Assumes that there are p processors, n > p, m = n / p, and
# processor i works on x[i*m] through x[i*m + m-1].  Takes O(m +
# beta) time in parallel.
def copy_scan(x):
    # Create a list to hold the result of the scan.
    n = m * p
    result = [None] * n

    # Scan x[0] into each processor.
    copy = [None] * p
    copy[0] = x[0]
    copy = simple_copy_scan(copy)

    # Copy the result of scanning x[0] into each processor
    # throughout the processor's portion of the result.
    do_in_parallel(copy_body, result, copy)

    barrier()

    return result

# Perform one step of copying the result of a copy-scan into item j
# of the result in processor i.
def copy_body(i, j, result, copy):
    result[i*m + j] = copy[i]
```

9.5 Partitioning in parallel

Recall how quicksort relies on partitioning a list. In fact, that's really all that quicksort is: each recursive call partitions a sublist of the full list. We can parallelize quicksort by parallelizing partitioning. Scan operations are key, along with two other parallel operations, meld and permute, which we'll see.

We'll be using boolean values quite a bit, but instead of `True` and `False`, it will be more convenient to use 1 and 0 instead, where 1 means `True` and 0 means `False`.

9.5.1 Meld operations

A ***meld*** operation takes values from k lists, where $k \geq 2$, and combines them into a single list, according to k boolean lists. This operation is best illustrated with an example. Here, we meld $k = 3$ lists—`x`, `y`, and `z`— holding values, according to k boolean lists—`a`, `b`, and `c`:

`index`	0	1	2	3	4	5	6	7	8	9	10	11	12	13
`a`	0	0	**1**	**1**	**1**	0	0	**1**	0	0	**1**	**1**	**1**	**1**
`x`	0	0	**0**	**1**	**2**	3	3	**3**	4	4	**4**	**5**	**6**	**7**
`b`	**1**	0	0	0	0	0	**1**	0	0	0	0	0	0	0
`y`	**8**	9	9	9	9	9	**9**	10	10	10	10	10	10	10
`c`	0	**1**	0	0	0	**1**	0	0	**1**	**1**	0	0	0	0
`z`	10	**10**	11	11	11	**11**	12	12	**12**	**13**	14	14	14	14
`result`	8	10	0	1	2	11	9	3	12	13	4	5	6	7

If `x[i]` has a 1, then the ith position of the result gets `x[i]`; if `b[i]` has a 1, then the ith position of the result gets `y[i]`; and if `c[i]` has a 1, then the ith position of the result gets `z[i]`. Values of `x`, `y`, and `z` that go into the result are in boldface in the above example, as are the boolean values that cause them to go there. We assume that for each position, exactly one of the boolean lists holds a 1 in that position, as in the above example.

We can perform a meld operation in $O(km + \beta)$ time in parallel. The code for the `meld` function is in Listing 9.9. The input to the `meld` function is a little complicated. It is a list named `lists`, composed of k 2-tuples. The first item of each of the k 2-tuples is a list of n data items. The second item of each 2-tuple is a list of n 0/1 values, indicating whether the data in the corresponding position makes it into the result. So the parameter `lists` is a list of 2-tuples of lists! We can access the hth 2-tuple by `lists[h]`. We can access the data list in the hth 2-tuple by `lists[h][0]`, and we can access the corresponding 0/1 list by `lists[h][1]`. And to access the data in item j of processor i in the hth list: `lists[h][0][i*m + j]`. Likewise, to access the corresponding 0/1 value: `lists[h][1][i*m + j]`. For our example

above, the call would be `meld([(x, a), (y, b), (z, c)])`; the order of the tuples does not matter, so that the call `meld([(y, b), (z, c), (x, a)])` would produce the same result.

Listing 9.9: The `meld` function.

```
# Meld k 2-tuples together on p processors with m items per
# processor. lists is a list of k lists.  Each of these 2-tuples
# consists of two lists.  The first list contains data to be melded,
# and the second list has a 1 if the data in the corresponding
# position of the first list is to be melded, and 0 otherwise.
# Assumes that there are p processors, n > p, m = n / p, and
# processor i works on x[i*m] through x[i*m + m-1]. Takes O(mk +
# beta) time in parallel with p processors.
def meld(lists):
    n = m * p
    result = [None] * n
    k = len(lists)
    do_in_parallel(meld_operation, result, lists, k, m)
    barrier()
    return result

# Operation to store into the jth item of the result in processor i.
# Goes through all k 0/1 lists, and when it finds a 1 in the jth
# item of processor i, puts the data in the jth item of processor i
# into the result.
def meld_operation(i, j, result, lists, k, m):
    for h in range(k):
        if lists[h][1][i*m + j] == 1:
            result[i*m + j] = lists[h][0][i*m + j]
```

Notice that the function `meld_operation`, which is called for each $i = 0, 1, 2, \ldots, p-1$ and $j = 0, 1, 2, \ldots, m-1$, has a for-loop that runs for $h = 0, 1, 2, \ldots, k-1$. Therefore, to fill in all m items of `result` in a given processor, it takes $O(km + \beta)$ parallel time. If k is a constant (3 in the above example), then the meld operation takes $O(m + \beta)$ time in parallel.

9.5.2 Permute operations

A ***permute*** operation takes two lists, say `perm` and `x`, and it produces a list `result` such that `result[perm[i]]` gets the value of `x[i]`. We assume that the `perm` list is a permutation

(i.e., a rearrangement) of the indices $0, 1, 2, \ldots, n-1$, so that what we are really doing is permuting the values in the list `x` according to the indices in `perm`: `perm[i]` gives the index in `result` that `x[i]` goes to. For example, here's a permute operation that uses the result of melding in the previous example as the `perm` list:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
perm	8	10	0	1	2	11	9	3	12	13	4	5	6	7
x	4	6	2	1	3	7	4	2	5	6	1	2	1	3
result	2	1	3	2	1	2	1	3	4	4	6	7	5	6

For example, because `perm[0]` is 8, the value 4 in `x[0]` appears in `result[8]`. Similarly, because `perm[1]` is 10, the value of `x[1]`, which is 6, appears in `result[10]`. And because `perm[4]` is 2, the 3 in `x[4]` appears in `result[2]`.

Listing 9.10 shows the Python code for the `permute` function. The permute operation runs in $O(m + \beta)$ time in parallel.

Listing 9.10: The functions for `permute` and `permute_body`.

```
# Permute the list x according to the indices in the list perm. Copy
# x[i] into the perm[i]th position of the result.  Assumes that
# there are p processors, n > p, m = n / p, and processor i works on
# x[i*m] through x[i*m + m-1].  Takes O(m + beta) time in parallel.
def permute(x, perm):
    n = m * p
    result = [None] * n
    do_in_parallel(permute_body, result, perm, x, m)
    barrier()
    return result

# Perform one step of permuting.  Copies the value of x in the jth
# position of processor i into the index given by the jth position
# in processor i of perm.
def permute_body(i, j, result, perm, x, m):
    result[perm[i*m + j]] = x[i*m + j]
```

9.5.3 Partitioning

Now we can see how to partition in parallel. One difference from how to partition in the quicksort code that you've probably seen before is that we use the first item, not the last, as the pivot. Another difference is that we'll make three partitions; in order from left to right: items less than the pivot, items equal to the pivot, and items greater than the pivot. Sequential quicksort would recurse on only the left and right partitions, not the middle, but when we work in parallel, it's easiest to recurse on all three partitions. We'll show how to partition by example, using the list `x` from the previous example.

We'll refer to performing operations "in parallel." That will mean "in as parallel a fashion as possible," typically taking parallel time $O(m + \beta \log p)$ to reduce or scan, and parallel time $O(m + \beta)$ to copy-scan, permute, or meld a constant number of lists.

Here is how to partition in parallel. Refer to Figure 9.4 for a running example.

`index`	0	1	2	3	4	5	6	7	8	9	10	11	12	13
`x`	4	6	2	1	3	7	4	2	5	6	1	2	1	3
`pivot`	4	4	4	4	4	4	4	4	4	4	4	4	4	4
`less`	0	0	1	1	1	0	0	1	0	0	1	1	1	1
`eq`	1	0	0	0	0	0	1	0	0	0	0	0	0	0
`greater`	0	1	0	0	0	1	0	0	1	1	0	0	0	0
`less_scan`	0	0	0	1	2	3	3	3	4	4	4	5	6	7
`eq_scan`	0	1	1	1	1	1	1	2	2	2	2	2	2	2
`greater_scan`	0	0	1	1	1	1	2	2	2	3	4	4	4	4
`less_red`	8	8	8	8	8	8	8	8	8	8	8	8	8	8
`eq_red`	2	2	2	2	2	2	2	2	2	2	2	2	2	2
`eq_perm`	8	9	9	9	9	9	9	10	10	10	10	10	10	10
`greater_perm`	10	10	11	11	11	11	12	12	12	13	14	14	14	14
`perm`	8	10	0	1	2	11	9	3	12	13	4	5	6	7
partitioned `x`	2	1	3	2	1	2	1	3	4	4	6	7	5	6

Figure (9.4) The steps in parallel partitioning.

1. Copy-scan `x[0]` into a new list `pivot`, so that `pivot[i]` equals `x[0]` for all i. That way, we can compare all positions of `x` to the pivot in parallel.

2. In parallel, compute three boolean lists: `less[i]` indicates whether `x[i]` < `pivot[i]`; `eq[i]` indicates whether `x[i]` equals `pivot[i]`; and `greater[i]` indicates whether `x[i]` > `pivot[i]`.

3. Perform a plus-scan on each of `less`, `eq`, and `greater`, and perform a plus-reduction on `less` and `eq`. Call the scan results `less_scan`, `eq_scan`, and `greater_scan`. Store the reduction results into `less_red[0]` and `eq_red[0]` (because we are going to copy-scan them).

   At this point, if `x[i]` is less than `x[0]` (the pivot), then `less_scan[i]` has the index of where we want to permute `x[i]`. If `x[i]` equals the pivot, then `eq_scan[i]` has the index of where we want to permute `x[i]`, *but within the set of items equal to the pivot.* And if `x[i]` is greater than the pivot, then `greater_scan[i]` has the index of where we want to permute `x[i]`, *but within the set of items greater than the pivot.*

4. Copy-scan `less_red` and `eq_red`. Add the result of copy-scanning `less_red` into `eq_scan`, item by item, giving `eq_perm`. Add the sum of the results of copy-scanning `less_red` and `eq_red` into `greater_scan`, item by item, giving `greater_perm`.

   Now if `x[i]` equals the pivot, then `eq_perm[i]` has the index of where we want to permute `x[i]` within the entire list, and if `x[i]` is greater than the pivot, then `greater_scan[i]` has the index of where we want to permute `x[i]` within the entire list.

5. Meld the lists `less_scan`, `eq_perm`, and `greater_perm`, using the boolean lists `less`, `eq`, and `greater`. Call the resulting list `perm`, and it gives the index where each value of `x[i]` should go.

6. Permute the list `x` according to the indices in `perm`. The resulting list has the list `x` partitioned around the pivot `x[0]`.

   In the example, indices 0 through 7 of the partitioned list hold values of `x` that are less than the pivot 4; indices 8 and 9 hold values of `x` that equal the pivot 4; and indices

10 through 13 hold values of `x` that are greater than the pivot 4.

Thus, we have successfully partitioned the list `x`.

9.5.4 Analysis

Let's add up the costs of the steps:

Step		Parallel time
1.	Copy-scan the pivot	$O(m + \beta)$
2.	Compute `less`, `eq`, and `greater`	$O(m + \beta)$
3.	Plus-scan on `less`, `eq`, and `greater`	$O(m + \beta \log p)$
	Plus-reduce `less` and `eq`	$O(m + \beta \log p)$
4.	Copy-scan `less_red` and `eq_red`	$O(m + \beta)$
	Compute `eq_perm` and `greater_perm`	$O(m + \beta)$
5.	Meld	$O(m + \beta)$
6.	Permute	$O(m + \beta)$

The total cost to partition is dominated by the most "expensive" steps, which are steps 3 and 4. Thus, we can partition in parallel in only $O(m + \beta \log p)$ time.

9.6 Parallel quicksort

Although partitioning is the key step in quicksort, partitioning just once is not enough. In quicksort, we have to partition at each step of the recursion, except for the base cases. One way to think of quicksort running in parallel on a list is to repeatedly partition the partitions, until each partition contains only items with the same value.

For example, suppose that the initial list contains the values

$$\left| \begin{array}{cccccccccccccc} 4 & 6 & 2 & 1 & 3 & 7 & 4 & 2 & 5 & 6 & 1 & 2 & 1 & 3 \end{array} \right|$$

Here, the vertical bars demarcate the start and end of the list, which is initially one big partition. If we partition around the pivot value 4, we get the following partitions:

$$\left| \begin{array}{cccccccc} 2 & 1 & 3 & 2 & 1 & 2 & 1 & 3 \end{array} \right| \begin{array}{cc} 4 & 4 \end{array} \left| \begin{array}{cccc} 6 & 7 & 5 & 6 \end{array} \right|$$

Next, partition around the pivot 2 in the first partition, around the pivot value 4 in the second partition, and around the pivot value 6 in the third partition. We get these partitions:

1	1	1	2	2	2	3	3	4	4	5	6	6	7

At this point, every partition consists of only one value, and we're done: the entire list is sorted.

How to keep track of the partitions? We use ***segment bits*** stored in a ***segment list***. Let's call it `seg`. Then `seg[i]` is 1 if index i begins a segment, and it's 0 otherwise. Here's what `seg` and the data above, let's call it `x`, should look like before each step above:

`index`	0	1	2	3	4	5	6	7	8	9	10	11	12	13
`seg`	1	0	0	0	0	0	0	0	0	0	0	0	0	0
`x`	4	6	2	1	3	7	4	2	5	6	1	2	1	3

`seg`	1	0	0	0	0	0	0	0	1	0	1	0	0	0
`x`	2	1	3	2	1	2	1	3	4	4	6	7	5	6

`seg`	1	0	0	1	0	0	1	0	1	0	1	1	0	1
`x`	1	1	1	2	2	2	3	3	4	4	5	6	6	7

We can create ***segmented scan*** operations, which are just like the regular scan operations, except that they treat each segment separately. For example, in the second `seg` and `x` lists above, here's what a segmented plus-scan would produce:

`seg`	1	0	0	0	0	0	0	0	1	0	1	0	0	0
`x`	2	1	3	2	1	2	1	3	4	4	6	7	5	6
`result`	0	2	3	6	8	9	11	12	0	4	0	6	13	18

How do we implement segmented scan operations? We'll see a little later that we can implement them with unsegmented scans, if we think outside the box. We'll also need segmented reductions. Once we have segmented scans and reductions, we can perform quicksort.

Here is how we perform one partitioning step in parallel. Refer to Figure 9.5 for a running example.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
seg	1	0	0	0	0	0	0	0	1	0	1	0	0	0
x	2	1	3	2	1	2	1	3	4	4	6	7	5	6
pivot	2	2	2	2	2	2	2	2	4	4	6	6	6	6
less	0	1	0	0	1	0	1	0	0	0	0	0	1	0
eq	1	0	0	1	0	1	0	0	1	1	1	0	0	1
greater	0	0	1	0	0	0	0	1	0	0	0	1	0	0
less_scan	0	0	1	1	1	2	2	3	0	0	0	0	0	1
eq_scan	0	1	1	1	2	2	3	3	0	1	0	1	1	1
greater_scan	0	0	0	1	1	1	1	1	0	0	0	0	1	1
less_red	3	3	3	3	3	3	3	3	0	0	1	1	1	1
eq_red	3	3	3	3	3	3	3	3	2	2	2	2	2	2
index_scan	0	0	0	0	0	0	0	0	8	8	10	10	10	10
less_perm	0	0	1	1	1	2	2	3	8	8	10	10	10	11
eq_perm	3	4	4	4	5	5	6	6	8	9	11	12	12	12
greater_perm	6	6	6	7	7	7	7	7	10	10	13	13	14	14
perm	3	0	6	4	1	5	2	7	8	9	11	13	10	12
partitioned x	1	1	1	2	2	2	3	3	4	4	5	6	6	7

Figure (9.5) How the steps of parallel partitioning in quicksort unfold.

1. In each segment, take the first item as the pivot in that segment, and copy-scan it throughout the segment. For example, let's start with the segments shown by `seg` and `x` in Figure 9.5. The segmented copy-scan gives the list `pivot` in the figure.

2. As in the unsegmented partitioning algorithm, compute three boolean lists: `less[i]` indicates whether `x[i]` $<$ `pivot[i]`; `eq[i]` indicates whether `x[i]` equals `pivot[i]`; and `greater[i]` indicates whether `x[i]` $>$ `pivot[i]`.

3. Perform a segmented plus-scan on each of `less`, `eq`, and `greater`, and perform seg-

mented plus-reductions on `less` and `eq`. Call the scan results `less_scan`, `eq_scan`, and `greater_scan`. Call the reduction results `less_red` and `eq_red`. Perform a segmented copy scan on `less_red` and `eq_red`.

Now, if `x[i]` is less than `pivot[i]`, then `less_scan[i]` has the index of where we want to permute `x[i]` *but within its current segment.* If `x[i]` equals `pivot[i]`, then `eq_scan[i]` + `less_red[i]` has the index of where we want to permute `x[i]` *within its current segment.* And if `x[i]` is greater than `pivot[i]`, then `eq_scan[i]`+`less_red[i]`+ `eq_red[i]` has the index of where we want to permute `x[i]` *within its current segment.*

4. Perform a segmented copy-scan of the index, giving `index_scan`.

5. Add `index_scan` and `less_scan`, giving `less_perm`. Add `index_scan`, `less_red`, and `eq_scan`, giving `eq_perm`. Add `index_scan`, `less_red`, `eq_red`, and `greater_scan`, giving `greater_perm`.

   Now `less_perm` gives the index *within the entire list* of where each item that is less than the corresponding pivot should go, `eq_perm` does the same for items that equal their pivots, and `greater_perm` does the same for items greater than their pivots.

6. Meld the lists `less_perm`, `eq_perm`, and `greater_perm`, using the boolean lists `less`, `eq`, and `greater`. Call the resulting list `perm`, and it gives the index *within the entire list* where each value of `x[i]` should go.

7. Permute the list `x` according to the indices in `perm`. The resulting list has each segment in `x` partitioned around the pivot in that segment.

8. As Figure 9.6 shows, to compute the new segment bits, add `index_scan` and `less_red`, giving `eq_start`. Add `index_scan`, `less_red`, and `eq_red`, giving `greater_start`. These values are the indices in which each new segment of values equal to the pivot and values greater than the pivot should start. Segments of values less than the pivot start where the original segments started. Make the new `seg` bits be 1 where `index` equals `index_scan`, `eq_start`, or `greater_start`; and 0 everywhere else.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
seg	1	0	0	0	0	0	0	0	1	0	1	0	0	0
less_red	3	3	3	3	3	3	3	3	0	0	1	1	1	1
eq_red	3	3	3	3	3	3	3	3	2	2	2	2	2	2
index_scan	0	0	0	0	0	0	0	0	8	8	10	10	10	10
partitioned x	1	1	1	2	2	2	3	3	4	4	5	6	6	7
eq_start	3	3	3	3	3	3	3	3	8	8	11	11	11	11
greater_start	6	6	6	6	6	6	6	6	10	10	13	13	13	13
new seg	1	0	0	1	0	1	0	0	1	0	1	1	0	1

Figure (9.6) How to compute the new segments bits in parallel after one parallel partitioning step of quicksort.

9. This process repeats until the entire list is sorted. To determine whether the entire list is sorted, compute a boolean list `sorted` such that `sorted[i]` equals `(index[i] == 0) or (x[i-1] <= x[i])`. (Notice that because the `or` operator is short-circuiting, no error occurs for $i = 0$.) Setting `sorted[i]` to 1 if `index[i]` is 0 makes it so that we don't need to compare the first value in the list to the value before it. Then just perform an `and`-reduction on `sorted`. If the `and`-reduction produces the value 1, then we're done. Otherwise, keep going.

9.6.1 Analysis

Let's see how long this quicksort algorithm takes. In the worst case, all the partition sizes are highly unbalanced, and sequential quicksort can take $O(n^2)$ time. On average, however, the partition sizes are close enough to equal that sequential quicksort takes $O(n \log n)$ time to sort an n-item list. (See [3, Section 7.4] for a rigorous analysis.)

Suppose we were to draw out a recursion tree for sequential quicksort, and suppose that the tree has d levels. Then the running time of sequential quicksort would be $O(nd)$. In the worst case, $d = n$, and on average d is $O(\log n)$.

For our parallel version, assuming that we can perform each segmented operation in $O(n/p+\beta \log p)$ parallel time (we'll see how in the next section), each iteration takes $O(n/p+\beta \log p)$ time in parallel. Therefore, the total parallel running time is $O(d(n/p + \beta \log p))$.

Again, let's see how to interpret running times that have the sum of two terms (such as n/p and $\beta \log p$). One of the terms dominates the other, and in terms of O-notation, that's the one that matters. For the moment, let's assume that $n/p \geq \beta \log p$, so that the n/p term dominates. In the worst case, $d = n$, and the parallel running time is $O(n(n/p))$, or $O(n^2/p)$, and we've managed to reduce the sequential time by a factor of p. In the average case, d is $O(\log n)$, and parallel quicksort takes parallel time $O((n/p) \log n)$, or $O((n \log n)/p)$. Again we've reduced the sequential time by a factor of p. Given that we have p processors, having the parallel running time be a factor of p lower than the sequential running time is the best we can hope for.

What if $n/p < \beta \log p$? Then the parallel time is $O(d\beta \log p)$, or $O(n\beta \log p)$ in the worst case and $O((\log n)(\beta \log p))$ in the average case. In both cases, we trade a factor of n in the sequential time for a factor of $\beta \log p$ in the parallel time. That's a good trade except when $\beta \log p$ is in the narrow range $n/p < \beta \log p < n$.

9.7 How to perform segmented scans and reductions

To close the loop, we need to see how to perform segmented scan and reduction operations. The basic idea uses enhanced operators whose operands are tuples. We also have to be able to perform backward scans within segments.

9.7.1 Segmented scans

We start with a basic segmented scan (not a segmented copy-scan, which we'll see later), in the procedure `segmented_scan` in Listing 9.11. We'll show how this function works by doing a max-scan on the list `x` shown in Figure 9.7, with the answer being the list `result`. Remember that scans are, by default, exclusive.

We're going to perform a segmented scan by performing an *un*segmented scan, but the

Listing 9.11: The functions segmented_scan, form_tuple, and unform_tuple.

```
# Perform a segmented exclusive scan operation on a list x of data,
# with a 0-1 list seg of segment bits.  Like a regular scan, but
# restarts the scan for each position i such that seg[i] is 1.
# Assumes that seg[0] is 1.  The parameter op gives the operation to
# perform, and the parameter ident is the identity for the
# operation.  Assumes that there are p processors, n > p, m = n / p,
# and processor i works on x[i*m] through x[i*m + m-1].  Takes O(m +
# beta log p) time in parallel.
def segmented_scan(x, seg, op, ident):
    n = m * p

    # Do an inclusive plus-scan on the segment bits, calling the
    # result seg_number.
    seg_number = inclusive_scan(seg, add, 0)

    # Make a list of tuples whose ith item is the ith item in
    # seg_number and the ith item in x.
    tuples = [None] * n
    do_in_parallel(form_tuple, tuples, seg_number, x, m)
    barrier()

    # Do an exclusive scan on tuples.
    tuple_scan = scan(tuples, segmented_op(op), (0, ident))

    # Where the ith item in seg is 1, the result of the segmented
    # scan is the identity.  Elsewhere, the result is the second
    # part of each tuple in tuple_scan.
    result = [None] * n
    do_in_parallel(unform_tuple, result, seg, tuple_scan, ident, m)
    barrier()

    return result

# Form a tuple of the segment number and value in x.
def form_tuple(i, j, tuples, seg_number, x, m):
    tuples[i*m + j] = (seg_number[i*m + j], x[i*m + j])

# After having scanned the tuples, use the identity where a segment
# starts.  Use the second value in the tuple everywhere else.
def unform_tuple(i, j, result, seg, tuple_scan, ident, m):
    if seg[i*m + j] == 1:
        result[i*m + j] = ident
    else:
        result[i*m + j] = tuple_scan[i*m + j][1]
```

index	0	1	2	3	4	5	6	7	8
seg	1	0	0	1	0	0	0	1	0
x	5	7	6	3	9	4	5	2	6
result	$-\infty$	5	7	$-\infty$	3	9	9	$-\infty$	2

Figure (9.7) The result of a segmented max-scan on a list `x`.

operator we use in the unsegmented scan will be a strange one that combines `op` (the operator for the segmented scan) and the segment bits. In particular, this operator—let's call it `@`—takes two 2-tuples as operands and returns a 2-tuple. Each 2-tuple consists of a segment number and a value. So we can think of the operator `@` as computing `(c[0], c[1]) = (a[0], a[1])@(b[0], b[1])`. Here, `c[0]`, `a[0]`, and `b[0]` are segment numbers. We will also require that `a[0]` $\leq$ `b[0]`.

The functions `segmented_operation` and `segmented_op` in Listing 9.12 give the rule for computing `@`. There's a ton going on in these few lines of code, so let's take it slowly. The idea is that `segmented_op` implements the `@` operator. Given the operator `op` that we're doing the segmented scan on, `segmented_op` returns another function, which is unnamed and takes just the parameters `a` and `b`, assumed to be 2-tuples of the form described above. By using the `lambda` form of Python, we build `op` into this function. So `segmented_op` returns a modified form of `segmented_operation` in which `op` is built-in.

Think of it this way. Let's name the function returned by `segmented_op` by assigning it to `f`, where `op` is `max`: `f = segmented_op(max)`. Then we can call `f(a, b)`, where `a` and `b` are our 2-tuples, and `f` returns a 2-tuple.

How does `segmented_operation` work? It first checks to see whether `b`'s segment number is greater than `a`'s segment number. If it is, then it just returns the 2-tuple `b`. Otherwise, the two segment numbers must be equal (recall that we require `a`'s segment number to be less than or equal to `b`'s, so if `a`'s is not less than `b`'s, then it must be equal), and the 2-tuple returned has this segment number as the first component and `op(a[1], b[1])` as the second

Listing 9.12: The functions segmented_operation and segmented_op.

```
# Perform a segmented operation on two tuples, a and b. The first
# item in each tuple is a segment number, and the second item in
# each tuple is a value.  op is the operation to
# perform. Assumption: a's segment number is less than or equal to
# b's segment number. If a's segment number is strictly less than
# b's segment number, then the result is b.  Otherwise, the segment
# numbers must be equal, and the result is a tuple comprising this
# segment number and the operator op applied to a's value and b's
# value.
def segmented_operation(a, b, op):
    if a[0] < b[0]:
        return b
    else:
        return (a[0], op(a[1], b[1]))

# Return a function on two tuples, a and b, performing a segmented
# operation op.  This operation, op, is built into the returned
# function.  The function returned may be called later on.
def segmented_op(op):
    return lambda a, b: segmented_operation(a, b, op)
```

component.

The idea behind segmented_operation is that if a and b are from the same segment, then we just perform op on the values in a and b. If they're in different segments, however, then b must be in a segment numbered higher than a's segment. That means b is starting a new segment, and we want to restart the scan just as if b was at index 0.

Now let's see, in Figure 9.8, how segmented_scan works on our above example to perform a max-scan. The first step is to compute segment numbers. The first segment is number 1, the second segment is number 2, and so on. An *inclusive* plus-scan on the segment bits does the trick, creating the list seg_number.

Next, we form the 2-tuples of the segment number and value in x by calling do_in_parallel on the form_tuple function:

index	0	1	2	3	4	5	6	7	8
seg	1	0	0	1	0	0	0	1	0
x	5	7	6	3	9	4	5	2	6
seg_number	1	1	1	2	2	2	2	3	3
tuples	(1, 5)	(1, 7)	(1, 6)	(2, 3)	(2, 9)	(2, 4)	(2, 5)	(3, 2)	(3, 6)
tuple_scan	(0, –∞)	(1, 5)	(1, 7)	(1, 7)	(2, 3)	(2, 9)	(2, 9)	(2, 9)	(3, 2)
result	–∞	5	7	–∞	3	9	9	–∞	2

Figure (9.8) The steps in implementing a segmented max-scan from unsegmented operations.

```
# Form a tuple of the segment number and value in x.
def form_tuple(i, j, tuples, seg_number, x, m):
    tuples[i*m + j] = (seg_number[i*m + j], x[i*m + j])
```

We call the resulting list `tuples`.

Now we do an exclusive scan on `tuples`, using the function returned by the call of `segmented_op(op)` as the function for the exclusive scan. Let's call the result `tuple_scan`. Where `seg[i]` equals 1, the result of the segmented scan is the identity for `op`, passed to `segmented_scan` as the parameter `ident`. Where `seg[i]` equals 0, the result is the second part of each tuple in `tuple_scan`; the `segmented_op` function effectively restarted the scan at the beginning of the segment. We call `do_in_parallel` on the function `unform_tuple`, which does exactly this:

```
# After having scanned the tuples, use the identity where a segment
# starts.  Use the second value in the tuple everywhere else.
def unform_tuple(i, j, result, seg, tuple_scan, ident, m):
    if seg[i*m + j] == 1:
        result[i*m + j] = ident
    else:
        result[i*m + j] = tuple_scan[i*m + j][1]
```

And we get the result shown in `result`.

The time to perform `segmented_scan` is dominated by the calls to `inclusive_scan` and

`scan`, one of each, for a parallel running time of $O(n/p+\beta \log p)$. (The calls to `do_in_parallel` and `barrier` just add another $O(n/p + \beta)$.)

9.7.2 Segmented inclusive scans

Once we have exclusive segmented scans, segmented inclusive scans are easy. We just use the `tack_on` function from Listing 9.6:

```
# Like segmented_scan, but performs an inclusive scan.
def segmented_inclusive_scan(x, seg, op, ident):
    result = segmented_scan(x, seg, op, ident)
    do_in_parallel(tack_on, result, x, op)
```

Like `segmented_scan`, the `segmented_inclusive_scan` function takes $O(n/p + \beta \log p)$ parallel time.

9.7.3 Segmented copy-scans

To perform a segmented copy-scan, we can think of performing a segmented inclusive scan using a copy operator:

```
# Operation for a segmented copy-scan.  Always returns the first of
# its two parameters.
def copy(a, b):
    return a
```

The only problem is that we don't really have an identity for the copy operation. So we can perform a segmented copy-scan by first doing a segmented scan with a copy operator and then just putting the original value back where segment bits are 1. The code appears in Listing 9.13. Here's how it works for our example. After calling `segmented_scan`, we have

index	0	1	2	3	4	5	6	7	8
seg	1	0	0	1	0	0	0	1	0
x	5	7	6	3	9	4	5	2	6
result	None	5	5	None	3	3	3	None	2

And then after calling `do_in_parallel` to perform `segmented_copy_scan_body`:

Listing 9.13: The functions `segmented_copy_scan` and `segmented_copy_scan_body`.

```
# Perform a segmented copy-scan operation on a list x of data, with
# a 0-1 list seg of segment bits.  Like a regular copy-scan, but
# restarts the scan for each position i such that seg[i] is 1.
# Assumes that seg[0] is 1.  Assumes that there are p processors, n
# > p, m = n / p, and processor i works on x[i*m] through x[i*m +
# m-1].  Takes O(m + beta log p) time in parallel.
def segmented_copy_scan(x, seg):
    result = segmented_scan(x, seg, copy, None)
    do_in_parallel(segmented_copy_scan_body, result, seg, x, m)
    barrier()
    return result

# Where seg[i] is 1, just copy x[i] into result[i].
def segmented_copy_scan_body(i, j, result, seg, x, m):
    if seg[i*m + j] == 1:
        result[i*m + j] = x[i*m + j]
```

index	0	1	2	3	4	5	6	7	8
seg	1	0	0	1	0	0	0	1	0
x	5	7	6	3	9	4	5	2	6
result	5	5	5	3	3	3	3	2	2

Once again, the segmented version of this operation has the same parallel running time, $O(n/p + \beta \log p)$, as its unsegmented counterpart.

9.7.4 Segmented reductions

To perform a segmented reduction, perform a segmented *inclusive* scan, and then perform a *backward*, segmented copy-scan on the result. This operation places the result of each segmented reduction into all the items in its segment.

How to perform a backward, segmented copy-scan? It's *almost* enough to just reverse the lists `x` and `seg`, perform a normal (forward) copy-scan, and then reverse the result. Again, using vertical bars to delineate segments (you'll see in a moment why we're not showing `seg`),

we'd get something like this:

`x`	5	7	6	3	9	4	5	2	6
reversed `x`	6	2	5	4	9	3	6	7	5
copy-scan	6	6	5	5	5	5	6	6	6
reverse again	6	6	6	5	5	5	5	6	6

That example gives the idea, but we cannot just reverse `seg` to indicate the start of each segment. Reversing `seg` places a 1 in the *last* position of each segment, not the first position:

`seg`	1	0	0	1	0	0	0	1	0
`x`	5	7	6	3	9	4	5	2	6
reversed `seg`	0	1	0	0	0	1	0	0	1
reversed `x`	6	2	5	4	9	3	6	7	5

Observe, however, that if position i ends a segment, then position $i + 1$ must start the next segment. In order to create the correct segment bits for the reversed version of `x`, therefore, we just need to shift the reversed segment bits one position to the right, filling the vacated bit at index 0 with a 1.

Listing 9.14 shows code to shift in parallel by a given number of positions, indicated by the parameter `amount`, and filling vacated positions with the value given by the parameter `fill`. If `amount` is positive, the shift is to the right, and if `amount` is negative, the shift is to the left. Listing 9.14 also gives the code to reverse a list in parallel. So now a backward segmented copy-scan is easy:

```
# Like segmented_copy_scan, but copies the last item in each segment
# throughout the segment.
def backward_segmented_copy_scan(x, seg):
    return reverse(segmented_copy_scan(reverse(x),
        shift(reverse(seg), 1, 1)))
```

And so is a segmented reduction:

```
# Perform a segmented reduction operation, copying the result of the
# reduction within each segment throughout the segment. The
# reduction operator is op, and the identity value for op is ident.
# Assumes that there are p processors, n > p, m = n / p, and
# processor i works on x[i*m] through x[i*m + m-1].  Takes O(m +
# beta log p) time in parallel.
def segmented_reduce(x, seg, op, ident):
    return backward_segmented_copy_scan(segmented_inclusive_scan(x,
        seg, op, ident), seg)
```

The `shift` and `reverse` functions take $O(n/p + \beta)$ parallel time, and so the parallel running time for `segmented_reduce` is the same as for `segmented_inclusive_scan` and `segmented_copy_scan`: $O(n/p + \beta \log p)$.

9.8 Comparing sequential vs. parallel running times

What are the tradeoffs between parallel and sequential computation? Given the additional complexity of parallel computing, when does it pay to compute in parallel instead of sequentially? In this section, we'll perform a simple analysis for reduction and scan operations.

We'll start with reductions where $n \leq p$. Let t be the time to perform one operation, that is, one call of the function `op` in `simple_reduce`, along with whatever overhead per loop iteration is incurred to call `op` repeatedly in either parallel or sequential code. Let h be the overhead per recursive call, which includes testing for the base cases, creating the list of half the size, handling the case in which n is odd, the barrier, and making the recursive call. Then the total time T_1^* to perform the reduction operation sequentially is

$$T_1^* = t(n-1) \ ,$$

since the operation must be applied $n-1$ times. The total time T_p to perform the reduction operation in parallel is

$$T_p = (t+h) \log n \ ,$$

Listing 9.14: Functions for shifting and reversing a list in parallel.

```
# Return a list that is the same as list x, but with all items
# shifted to the right by amount positions.  If amount is negative,
# then shift to the left.  Fill in evacuated positions with the
# parameter fill.  Assumes that there are p processors, n > p, m = n
# / p, and processor i works on x[i*m] through x[i*m + m-1].  Takes
# O(m + beta) time in parallel.
def shift(x, amount, fill):
    n = m * p
    result = [None] * n
    do_in_parallel(shift_body, result, x, amount, fill, n, m)
    barrier()
    return result

def shift_body(i, j, result, x, amount, fill, n, m):
    index = i*m + j
    if index < amount or index >= n + amount:
        result[index] = fill
    else:
        result[index] = x[index - amount]

# Return the reverse of a list x.  Assumes that there are p
# processors, n > p, m = n / p, and processor i works on x[i*m]
# through x[i*m + m-1].  Takes O(m + beta) time in parallel.
def reverse(x):
    n = m * p
    result = [None] * n
    do_in_parallel(reverse_body, result, x, n, m)
    barrier()
    return result

def reverse_body(i, j, result, x, n, m):
    index = i*m + j
    partner_index = n - index - 1
    result[partner_index] = x[index]
```

since each recursive call can perform all the calls to `op` in parallel. The parallel version is worthwhile when $T_p < T_1^*$, or

$$(t + h) \log n < t(n - 1) \ .$$

Dividing both sides by t gives

$$(1 + h/t) \log n < n - 1 \ .$$

Taking 2 to both sides gives

$$2^{(1+h/t) \log n} < 2^{n-1} \ ,$$

or

$$n^{1+h/t} < 2^{n-1} \ .$$

Let's look at some possible ratios of h/t. If $h/t = 5$, then $T_p < T_1^*$ for $n \geq 31$. If $h/t = 10$, then $T_p < T_1^*$ for $n \geq 68$, and if $h/t = 20$, then $T_p < T_1^*$ for $n \geq 154$.

The analysis is similar for scan operations when $n \leq p$. Now each recursive call of `simple_scan` performs the operation `op` twice in parallel and calls `barrier` twice, so that

$$T_p = 2(t + h) \log n \ .$$

The parallel version is worthwhile when $T_p < T_1^*$, or

$$n^{2(1+h/t)} < 2^{n-1} \ .$$

Using the same h/t ratios as before—5, 10, and 20—we get that $T_p < T_1^*$ for $n \geq 76$, 163, and 358, respectively.

Now let's look at reduction operations when $n > p$. The sequential time $T_1^* = t(n - 1)$

remains unchanged, but the parallel time for p processors becomes

$$T_p = tn/p + \beta + (t+h)\log p \ .$$

To keep things simple, we know that $\beta < h$, so let's bound T_p by

$$T_p < tn/p + h + (t+h)\log p \ ,$$

so that when we divide both sides by t, we find that $T_p < T_1^*$ when

$$n/p + h/t + (1+h/t)\log p < n - 1 \ .$$

If we hold the ratio h/t constant and increase the number p of processors, we find that the crossover point where T_p goes below T_1^* increases with p. For example, Figure 9.9 shows crossover points for $h/t = 5$, 10, and 20. Taking into account that the horizontal axes show $\log p$, these crossover points increase quite slowly with the number of processors.

For scan operations when $n > p$, there are two calls to `do_in_parallel` and two calls to `barrier`, and so the parallel time becomes

$$T_p = 2(tn/p + \beta) + (t+h)\log p \ ,$$

which we again bound by

$$T_p < 2(tn/p + h) + (t+h)\log p \ .$$

Again dividing both sides by t, we get that $T_p < T_1^*$ when

$$2(n/p + h/t) + (1+h/t)\log p < n - 1 \ .$$

As with the reduction operation, if we hold the ratio h/t constant and increase the number p

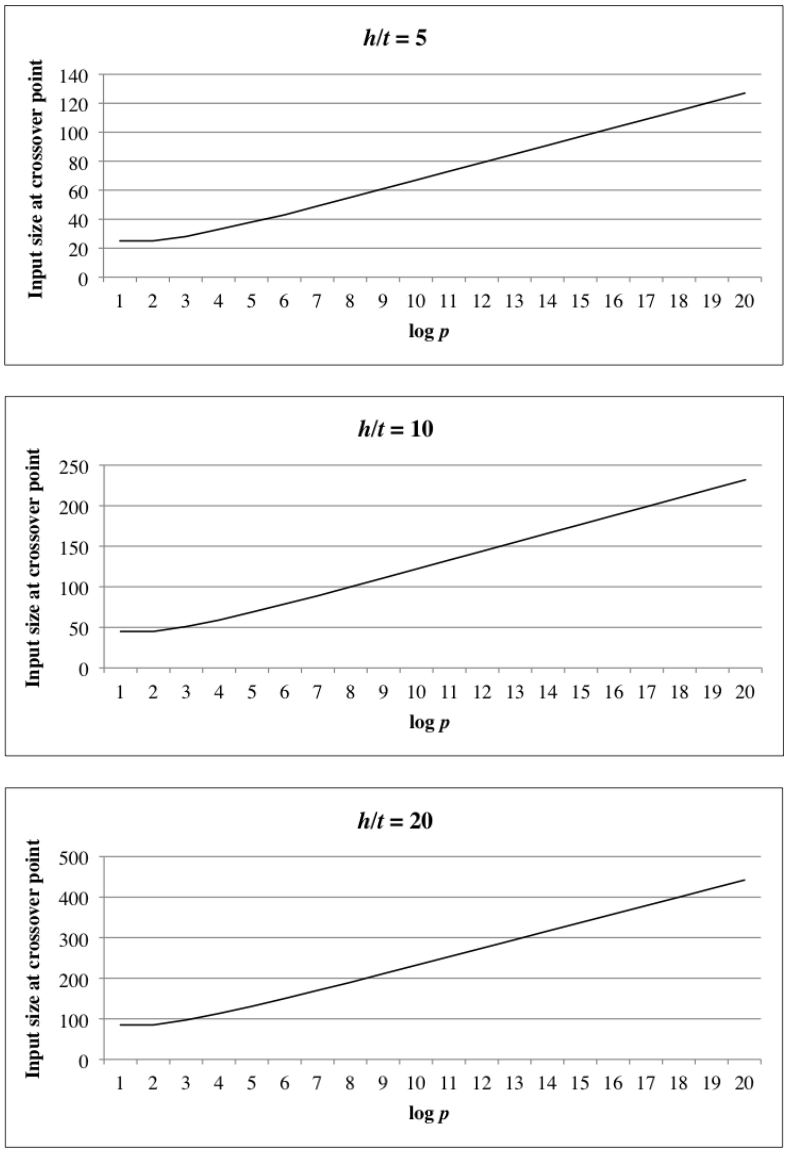

Figure (9.9) Input size n at crossover points for reduction operations at which $T_p < T_1^*$ for $h/t = 5$, 10, and 20 when $n > p$.

of processors, we once again find that the crossover point where T_p goes below T_1^* increases with p, now for $p \geq 8$. Figure 9.10 shows crossover points for $h/t = 5$, 10, and 20. Once again, the crossover points increase rather slowly with the value of p.

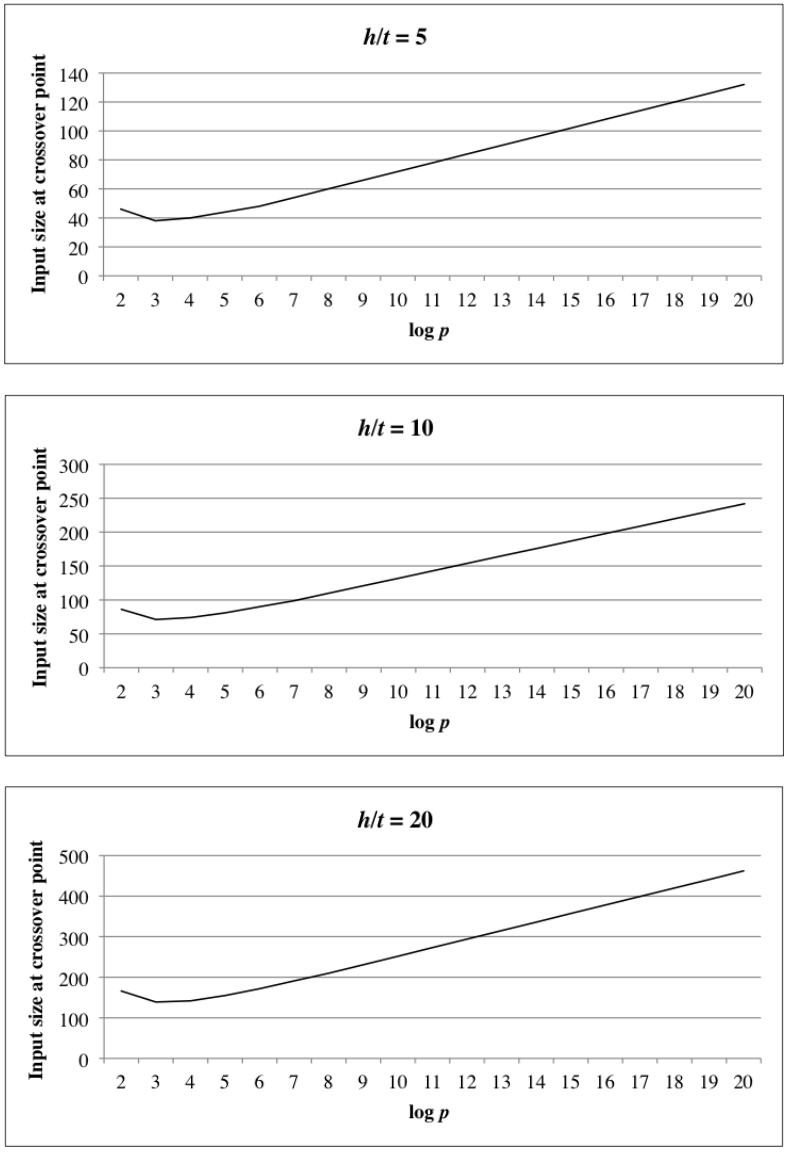

Figure (9.10) Input size n at crossover points for scan operations at which $T_p < T_1^*$ for $h/t = 5$, 10, and 20 when $n > p$.

REFERENCES

[1] G. E. Blelloch, "Scan primitives and parallel vector models," Ph.D. dissertation, Massachusetts Institute of Technology, 1988.

[2] ——, *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.