Programming Parallel Algorithms*

Guy E. Blelloch
Guy.Blelloch@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

In the past 20 years there have been a huge number of algorithms designed for parallel computers, most which have been designed for one of the variants of the Parallel Random Access Machine (PRAM) model. Unfortunately there has been limited progress in getting practical implementations of the algorithms on any real parallel machine. Although discrepancies between the PRAM model and actual implementations of parallel machines (particularly as regards communication costs) has played a part in this lack of progress, another significant problem is the lack of good programming languages. With the languages that come with existing parallel machines it can be a major project to implement a simple algorithm, and once implemented the code is unlikely to port to any other parallel machine.

This paper describes a data-parallel language, NESL, designed for programming parallel algorithms. NESL currently runs on the Connection Machine CM-2 and the Cray Y-MP, and generates reasonably efficient code for both machines. The paper gives several examples of simple algorithms implemented in the language. All examples include the full code and run on both the CM-2 and Cray Y-MP.

1 Introduction

Although there has been much work on the design of parallel algorithms there has been little on the design of languages for expressing these algorithms. At first sight it seem that designing a language for PRAM algorithms should be trivial. In fact, designing a *vanilla* PRAM language would be quite easy. By vanilla we mean a language that directly simulates a synchronous p-processor PRAM. This paper claims, however, that the bookkeeping required to program a "vanilla" PRAM model is similar in difficulty to the bookkeeping that would be required to explicitly control all your memory (manage one giant block of memory) in a serial language. When informally describing algorithms it is easy to ignore this bookkeeping, but when programming it every detail needs to be precisely stated. Section 2 outlines some of the problems with programming a vanilla PRAM model.

This paper outlines the data-parallel language NESL [5]. The language was designed to make it as easy as possible to describe parallel algorithms by removing any unnecessary bookkeeping. NESL currently runs on the Connection Machine CM-2 and the Cray Y-MP as well as on any serial workstation.[1] Various benchmark algorithms achieve very good running times on these machines [10, 6]. The language is based on vectors (sequences) as a primitive parallel data type, and parallelism is achieved through operations on these vectors [4]. In NESL the user thinks about parallel operations over collections of values, rather than about how data is mapped to processors and the control of these processors. The mapping is done at a lower level.

The complexity for algorithms in NESL is derived in terms of two numbers: the *work* and *step* complexities [4]. The work complexity represents the serial work executed by an algorithm—an upper bound on the running time if executed on a serial RAM. The step complexity represents the deepest path taken by the algorithm—the running time if executed with an unbounded number of processors. These two complexities can be used to place an upper bound on the asymptotic running times for the PRAM. In NESL, if the work

*This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

[1] It is available via anonymous FTP. You may send a request to the author if you are interested.

complexity is W and the step complexity is S, then the bound on a PRAM with scan primitives [3] is

$$t = O(W/P + S) \tag{1}$$

and the bound on a PRAM without scan primitives is

$$t = O(W/P + S \lg n) \tag{2}$$

where P is the number of processors. This formula can be derived from Brent's scheduling principle [9] as shown in [4], and is similar to the use of Brent's principle as used informally to aid algorithm analysis in [24, 16].

NESL is a strongly-typed side-effect free language. It runs within an interactive environment, has a Lisp-like syntax[2], and allows the user to run the environment, including the compiler, on a local workstation while executing interactive calls to NESL programs on the CRAY Y-MP or CM-2 (or any other workstation, if so desired). NESL currently generates a portable intermediate code called VCODE [7].

Control parallel languages that have some feature that are similar to NESL include ID [22, 1], Sisal [17], and Proteus [20]. ID and Sisal are both side-effect free and supply operations on collections of values.

2 The Vanilla PRAM model

This section considers a few of the problems with trying to program directly in a *vanilla* PRAM model. The first problem is that when programming a parallel machine we can't just pick the number of processors to match the size of our data; the bookkeeping needed to explicitly keep track of where the data is located and which processors need to do what can be cumbersome.

Consider the case where we have more processors than data elements. To avoid the possibility of processors with no data inadvertently clobbering memory, we must be certain that these processors are inactive. Languages for the Connection Machine use the notion of a *context-flag* to handle this, where a context flag is a bit on each processor that activates and deactivate the processor. Since the number of data elements can change during an algorithm, and we might have several data sets of different sizes (eg. the edges and verices of a graph), algorithms often require continuously setting and resetting the flags. Manipulating these flags can generate cumbersome code and can be a source for a huge number of bugs in programming parallel machines.

[2] It should not, however, be confused with Lisp or Scheme. In fact, we are currently working on a new version of the language in which the syntax will be much closer to the syntax of the ML language [21].

In the other case where we have more data elements than processors, we have to place multiple elements per processor and write loops to simulate all the elements. Writing these loops makes the code messy (especially since they might be different lengths in different processors). One way to avoid this is to use the notion of *virtual processors*, as included by most of the CM language. The idea is that one can place multiple virtual-processors per physical processor, and the underlying software is responsible for doing the looping. The question now becomes, is the number of virtual processors fixed (as was the case in the original CM languages), or can it vary during our computation? If it is fixed, we are hardly any better off than originally since often the size of our data changes during the computation. On the other hand, if we allow the number of processors to change, we no longer have a vanilla PRAM model and this has to be taken into account in complexity analysis.

As another problem, consider an algorithm that splits its data into parts and allocates some subset of the processors to each subproblem. Although in an informal description it is easy to say that we can do this, and easy to make statements such as "without loss of generality, let's assume that the number of parts evenly divides the number of processors", for a programmer to do it can be a bookkeeping nightmare.

3 Parallel Operations

NESL abstracts away from the PRAM model and is not based on the notion of processors, although it is easy to map its complexity back onto a PRAM model. NESL supports parallelism through operations on vectors. This parallelism can be achieved in two ways. First, any function can be applied over the elements of a vector. For example, the expression

```
(over ((a #v(7 -2 5 4)))
  (negate a))
⇒ #v(-7 2 -5 -4) : v.int
```

negates each elements of the vector `#v(7 -2 5 4)`. This construct can be read as "for each element `a` in the vector `#v(7 -2 5 4)`, negate `a`" (in the example, the symbol ⇒ points to the result of the expression, and the `v.int` indicates that the result is of type *vector of integers*). The general for of an `over` expression is

(`over` ((*variable-name value-exp*)`+`)
  body-exp)

where the `+` signifies that the *variable-name value-exp* pairs can be repeated one or more times. All *value-exp* expressions must return vectors of equal length . The `over` form can also be expressed in shorthand as

```
    (v.negate #v(7 -2 5 4))
⇒   #v(-7 2 -5 -4) : v.int
```

This shorthand **v.func** form is considered syntactic sugar for the `over` form (see [5]).

In NESL, any function, whether primitive or user defined, can be applied to each element of a vector. So, for example, we could define a factorial function

```
    (defop (factorial i) (int <- int)
      (if (= i 1)
          1
          (* i (factorial (- i 1)))))
⇒   factorial : int <- int
```

and then apply it over the elements of a vector

```
    (v.factorial #v(3 1 7))
⇒   #v(6 1 5040) : v.int
```

In this example, the

(defop (*function-name argument**) (*type*)
body-exp)

form is used to define `factorial`. The *type* argument specifies the type of the function, which in this case is specified as `(int <- int)`, indicating a function that maps integers to integers.

An `over` form applies a body to each element of a vector. We will call each such application an *instance*. Since there are no side effects in NESL, there is no way to communicate among the instances of an `over` form. An implementation can therefore execute the instances in any order it chooses without changing the result. In particular, the instances can be implemented in parallel, therefore giving `over` its parallel semantics. To derive the *work* complexity of an `over` form, we add the complexities of all the instances. To derive the *step* complexity of an `over` form, we take the maximum of the complexities of all the instances.

In addition to the `over` form, a second way to take advantage of parallelism in NESL is through a set of vector functions. The vector functions operate on whole vectors and all have relatively simple parallel implementations. For example the function `+-reduce` sums the elements of a vector.

```
    (+-reduce #v(2 1 -3 11 5))
⇒   16 : int
```

Since addition is associative, this can be implemented on a parallel machine in logarithmic time using a tree. Another common vector function is the `permute` function, which permutes a vector based on a second vector of indices. For example:

```
(defop (select s k) (int <- v.int int)
  (with ((l   (length s))
         (p   (elt s (/ l 2)))
         (les (pack s (v.< s v.p))))
    (if (< k (length lesser))
      (select les k)
      (with ((grt (pack s (v.> s v.p))))
        (if (>= k (- l (length grt)))
          (select grt
                  (- k (- l (length grt))))
          p)))))
```

Figure 1: An implementation of selection. The function `select` returns the `k`th smallest element from the input vector `s`.

```
    (permute "nesl" #v(2 1 3 0))
⇒   "lens" : v.char
```

In this case, the 4 characters of the string `"nesl"` (the term *string* is used to refer to a vector of characters) are permuted to the indices `#v(2 1 3 0)` (`n` → 2, `e` → 1, `s` → 3, and `l` → 0). The implementation of the `permute` function on a PRAM can use an exclusive write into memory.

Some other vector functions include `(length v)`, which returns the length of the vector `v`, `(elt v i)`, which removes the $\mathtt{i}^{th}$ element from the vector `v`, and `(pack v f)` which takes a vector of elements `v` and a vector of flags `f` of equal length and packs the elements where the flag is true into a smaller vector. For example,

```
s            =  #v(4 8 2 3 1 7 2)
f            =  #v(f f t f t f t)

(length s)   =  7
(elt s 1)    =  8
(pack s f)   =  #v(2 1 2)
```

Each vector function in NESL has a *work* and *step* complexity associated with it. The work complexity of each of the functions is typically proportional to the size of one or more of the vector arguments. The step complexity for all the primitive functions is $O(1)$.

4 Selecting: An Example

We consider an example of the use of vectors in NESL. The algorithm we consider solves the problem of finding the $\mathtt{k}^{th}$ smallest element in a set `s`, using a parallel version of the quickselect algorithm [14]. Quickselect is similar to quicksort, but only calls itself recursively on either the elements lesser or greater than the pivot. The

NESL code for the algorithm is shown in Figure 1. The `with` form is used to bind local variables and is like a `let*` in Common Lisp. The code first binds `l` to the length of the input vector `s`, and then extracts the middle element of `s` as a pivot. The algorithm then selects all the elements less than the pivot, and places them in a vector that is bound to `les`. This is done by comparing each element of `s` to the pivot, and using the `pack` function to remove the elements that fail the test. The `v.pivot` means that the pivot is extended to the same length as `s`.

After the pack, if the number of elements in the set `les` is greater than `k`, then the $\mathtt{k}^{th}$ smallest element must belong to that set. In this case, the algorithm calls `select` recursively on `les` using the same `k`. Otherwise, the algorithm selects the elements that are greater than the pivot, again using `pack`, and can similarly find if the $\mathtt{k}^{th}$ element belongs in the set `grt`. If it does belong in `grt`, the algorithm calls itself recursively, but must now readjust `k` by subtracting off the number of elements lesser and equal to the pivot. If the $\mathtt{k}^{th}$ element belongs in neither `les` nor `grt`, then it must be the pivot, and the algorithm returns this value.

This code has an expected work complexity that is the same as the time for the serial algorithm. This can be shown to be $O(n)$ [11]. Since we expect only to make $O(\lg n)$ recursive calls to `order` (this is not hard to show), and each call has a constant number of steps, the step complexity is $O(\lg n)$. The `select` algorithm only uses primitives that require exclusive memory access, so the algorithm maps onto a EREW PRAM with an expected asymptotic running time

$$T(n) = O(n/p + \lg n) \ .$$

5 Nested Parallelism

In NESL the elements of a vector can be any valid data item, including vectors. This rule permits the nesting of vectors to an arbitrary depth. A nested vector can be written as

```
#v(#v(2 1) #v(7 3 0) #v(4))
```

This vector has type vector of vector of integers (`v.v.int`). Given nested vectors and the rule that any function can be applied in parallel over the elements of a vector, NESL necessarily supplies the ability to apply a parallel function multiple times in parallel; we call this ability *nested parallelism*. For example, we could apply the parallel vector function `+-reduce` over a nested vector:

```
   (over ((v #v(#v(2 1) #v(7 3 0) #v(4))))
     (+-reduce v))
⇒  #v(3 10 4) : v.int
```

In this expression there is parallelism both within each `+-reduce`, since the vector function has a parallel implementation, and across the three instances of `+-reduce`, since `over` is defined such that all instances can run in parallel.

NESL supplies a handful of functions for moving between levels of nesting. These include `flatten`, which takes a nested vector and flattens it by one level. For example,

```
   (flatten #v(#v(2 1) #v(7 3 0) #v(4)))
⇒  #v(2 1 7 3 0 4) : v.int
```

Another useful function is `bottop` (for bottom and top), which takes a vector of values and creates a nested sequence of length 2 with all the elements from the bottom half of the input vector in the first element and elements from the top half in the second element (if the length of the vector is odd, the bottom part gets the extra element). For example,

```
   (bottop "nested")
⇒  #v("nes" "ted") : v.v.char
```

As an example, consider how the function `+-reduce` might be implemented,

```
(defop (+-reduce a) (int <- v.int)
  (if (= (length a) 1)
    (elt a 0)
    (with (((v1 v2)
            (vsep (v.+-reduce (bottop a)))))
       (+ v1 v2))))
```

This code tests if the length of the input is one, and returns the single element if it is. If the length is not one, it uses `bottop` to split the sequence in two parts, and then applies itself recursively to each part in parallel. When the parallel calls return, the function `vsep` (for vector separate) converts the resulting vector of two elements into a *tuple* (a tuple in NESL is just a pair of values). Pattern matching in the `with` assigns these two results to `v1` and `v2`, which are then summed and returned. The code effectively creates a tree of parallel calls which has depth $\lg n$, where n is the length of `a`, and executes a total of $n-1$ calls to `+`. To simulate the built-in `+-reduce`, it would be necessary to add code to return the identity (0) for empty vectors. Using our composition rules for `v.`, the complexities of this algorithm are

```
(defop (primes n) (v.int <- int)
  (if (= n 1)
    #v.int()
    (with ((proot  (primes (isqrt n)))
           (sieves (over ((p proot))
                     (iseq (* 2 p) p n)))
           (sieve  (flatten sieves))
           (l      (length sieve))
           (flags  (put (dist f l)
                        sieve
                        (dist t n))))
      (drop 2 (pack-index flags)))))
```

Figure 2: Finding all the primes less than n.

$$W(n) = \begin{cases} O(1) & n = 1 \\ 2W(n/2) + O(1) & n > 1 \end{cases} = O(n)$$

$$S(n) = \begin{cases} O(1) & n = 1 \\ S(n/2) + O(1) & n > 1 \end{cases} = O(\lg n) \ .$$

6 Primes: An Example

We consider another example: finding all the primes less than n. The algorithm is based on the sieve of Eratosthenes. The basic idea of the sieve is to find all the primes less than $\sqrt{n}$, and then use multiples of these primes to "sieve out" all the composite numbers less than n. Since all composite numbers less than n must have a divisor less than $\sqrt{n}$, the only elements left unsieved will be the primes. There are many parallel versions of the prime sieve, and several naive versions require a total of $O(n^{3/2})$ work and either $O(n^{1/2})$ or $O(n)$ parallel time. A well designed version should require no more work than the serial sieve ($O(n \lg\lg n)$), and polylogarithmic parallel time.

The version we use (see Figure 2) requires $O(n \lg\lg n)$ work and $O(\lg\lg n)$ steps. It works by first recursively finding all the primes up to $\sqrt{n}$, (`proot`). Then, for each prime `p` in `proot`, the algorithm generates all the multiples of `p` up to `n` (`sieves`). This is done with the `(iseq s d e)` function, which returns the integers starting at `s`, increasing by `d`, and up to, but less than, `e`. For example,

```
    (iseq 4 2 20)
⇒  #v(4 6 8 10 12 14 16 18) : v.int
```

The vector `sieves` is therefore a nested vector with each subvector being the sieve for one of the primes in `proot`. The function `flatten`, is now used to flatten this nested vector by one level, therefore returning a vector containing all the sieves. This vector of sieves is used by the `put` function to place a false flag in all positions that belong to a sieve. The `(dist a l)` function distributes a value `a` across a vector of length `l`. The `(put v i d)` function places the values from `v` (in this case, all false flags) into a vector `d` (in this case a constant vector of true flags), at positions `i` (in this case, the sieve positions). Finally the `pack-index` function returns a vector of indices where the flags remain true (elements which were not sieved), and `drop`, removes the first two elements (0 and 1), which are not considered primes.

The functions `iseq`, `flatten`, `put`, `pack-index` and `drop` all require a constant number of steps. Since `primes` is called recursively on a problem of size $\sqrt{n}$ the total number of steps require by the algorithm can be written as the recurrence:

$$S(n) = \begin{cases} O(1) & n = 1 \\ S(\sqrt{n}) + O(1) & n > 1 \end{cases} = O(\lg\lg n)$$

Almost all the work done by `primes` is done at the top level. At this top level, the work is proportional to the length of the vector `sieve`. Using the standard formula

$$\sum_{p \le x} 1/p = \log\log x + C + O(1/\log x)$$

where p are the primes [13], the length of this vector is:

$$\begin{aligned} \sum_{p \le \sqrt{n}} n/p &= O(n \log\log\sqrt{n}) \\ &= O(n \log\log n) \end{aligned}$$

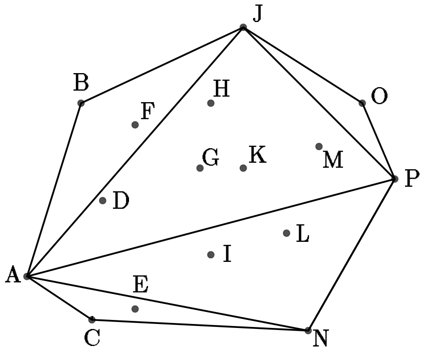
therefore giving a work complexity of $O(n \log\log n)$.

The asymptotic running time on a ERCW PRAM with scans (the `put` requires a concurrent write) is therefore

$$T(n) = O((n/p + 1) \lg\lg n) \ .$$

7 Convex-Hull: An Example

Our next example solves the planar convex hull problem: given n points in the plane, find which of these points lie on the perimeter of the smallest convex region that contains all points. The planar convex hull problem has many applications ranging from computer



[A B C D E F G H I J K L M N O P]

A [B D F G H J K M O] P [C E I L N]

A [B F] J [O] P N [C E]

A B J O P N C

Figure 3: An example of the *quickhull* algorithm. Each vector shows one step of the algorithm. Since A and P are the two x extrema, the line AP is the original split line. J and N are the farthest points in each subspace from AP and are, therefore, used for the next level of splits. The values outside the brackets are hull points that have already been found.

graphics [12] to statistics [15]. The algorithm we use to solve the problem is a parallel version [8] of the *quickhull* algorithm [23]. The quickhull algorithm was given its name because of its similarity to the quicksort algorithm. As with quicksort, the algorithm picks a "pivot" element, splits the data based on the pivot, and is recursively applied to each of the split sets. Also, as with quicksort, the pivot element is not guaranteed to split the data into equally sized sets, and in the worst case the algorithm will require $O(n^2)$ work.

Figure 3 shows an example of the quickhull algorithm, and Figure 4 shows the code. The algorithm is based on the recursive routine `hsplit`. This function takes a set of points in the plane ($\langle x, y\rangle$ coordinates) and two points `p1` and `p2` that are known to lie on the convex hull, and returns all the points lie on the hull clockwise from `p1` to `p2`. In the figure, given all the points `#v(A B C ... P)`, `p1 = A` and `p2 = P`, `hsplit` would return the vector `#v(B J O)`. In `hsplit`, the order of `p1` and `p2` matters, since if we switch `A` and `P`, `hsplit` would return the hull along the other direction `#v(N C)`.

The `hsplit` function works by first removing all the elements that cannot be on the hull since they lie below the line between `p1` and `p2`. This is done by removing elements whose cross product with `p1` and `p2` are negative. In the case `p1 = A` and `p2 = P`, the points `#v(B D F G H J K M O)` would remain and be placed in the vector `packed`. The algorithm now finds the point furthest from the line `p1-p2`. This point `pm` must be on the hull since as a line at infinity parallel to `p1-p2` moves toward `p1-p2`, it must first hit `pm`. The point `pm` (`J` in the running example) is found by taking the point with the maximum cross-product. Once `pm` is found, `hsplit` calls itself twice recursively using the points `(p1, pm)` and `(pm, p2)` (`(A, J)` and `(J, P)` in the example). When the recursive calls return, `hsplit` appends `r1` (the hull between `p1` and `pm`), the point `pm`, and `r2` (the hull between `pm` and `p2`). This gives the desired result.

The overall `convex-hull` algorithm works by finding the points with minimum and maximum `x` coordinates (these points must be on the hull) and then using `hsplit` to find the upper and lower hull. Each recursive call has a step complexity of $O(1)$ and a work complexity of $O(n)$. However, since many points might be deleted on each step, the work complexity could be significantly less. For m hull points, the algorithm runs in $O(\lg m)$ steps for well-distributed hull points, and has a worst case running time of $O(m)$ steps.

8 Conclusion

This paper has shown several examples of how a programming language can be used for describing parallel algorithms and for deriving their asymptotic complexities. The examples shown were relatively small so that we could give the full code. The ability to nest parallel calls is critical to the expression of many of these algorithms. This ability in not supplied by any of the other implemented data-parallel languages. The notion of deriving complexity in terms of *work* and *steps* is also critical in being able to compose the complexities.

In addition to the examples in this paper, many other algorithms have been coded in NESL, including quicksort, radixsort, mergesort, the Shiloach-Vishkin and Awerbuch-Shiloach connected-components algorithms [25, 2], a randomized work efficient list-ranking algorithm [19], various tree operations based on Euler tours [26], an algorithm for matching parentheses in a string, and a graph separator algorithm [18].

Acknowledgments

I would like to thank Marco Zagha, Tim Freeman, Jay Sipelstein, Margaret Reid-Miller, John Greiner, Jonathan Hardwick, Siddhartha Chatterjee,

```
(defrec point (x int) (y int))

(defop (cross-product o p1 p2) (int <- point point point)
  (- (* (- (x p1) (x o)) (- (y p2) (y o)))
     (* (- (y p1) (y o)) (- (x p2) (x o)))))

(defop (hsplit points p1 p2) (v.point <- v.point point point)
  (if (< (length points) 2)
      points
    (with ((cross   (v.cross-product points v.p1 v.p2))
           (packed  (pack points (v.plusp cross)))
           (pm      (elt points (max-index cross)))
           ((r1 r2) (vsep (v.hsplit (vtup packed packed)
                                    (vtup p1     pm)
                                    (vtup pm     p2)))))
      (append r1 (cons pm r2)))))

(defop (convex-hull points) (v.point <- v.point)
  (with ((min-x (elt points (min-index (v.x points))))
         (max-x (elt points (max-index (v.x points)))))
    (append (cons min-x (hsplit points min-x max-x))
            (cons max-x (hsplit points max-x min-x)))))
```

Figure 4: Code for Quickhull. The `defrec` form defines a record with two slots, an `x` slot and a `y` slot, which are both integers. The commands `(x p)` and `(y p)` are used to access the two slots from the record `p`.

Bob Harper and Gary Miller for many helpful comments on the language.

References

[1] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[2] Baruch Awerbuch and Yossi Shiloach. New connectivity and msf algorithms for ultracomputer and pram. In *Proceedings International Conference on Parallel Processing*, pages 175–179, 1983.

[3] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989.

[4] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[5] Guy E. Blelloch. Nesl: A nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, January 1992.

[6] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Portable data-parallel algorithms. In Preparation, 1992.

[7] Guy E. Blelloch, Siddhartha Chatterjee, Fritz Knabe, Jay Sipelstein, and Marco Zagha. VCODE reference manual (version 1.1). Technical Report CMU-CS-90-146, School of Computer Science, Carnegie Mellon University, July 1990.

[8] Guy E. Blelloch and James J. Little. Parallel solutions to geometric problems on the scan model of computation. In *Proceedings International Conference on Parallel Processing*, pages Vol 3: 218–222, August 1988.

[9] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the Association for Computing Machinery*, 21(2):201–206, 1974.

[10] Siddhartha Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1991.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press and McGraw-Hill, 1990.

[12] H. Freeman. Computer processing of line-drawing images. *Computer Surveys*, 6:57–97, 1974.

[13] G. H. Hardey and E. M. Wright. *An Introduction to the Theory of Numbers, 5th ed.* Oxford University Press, Oxford, New York, 1983.

[14] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.

[15] J. G. Hocking and G. S. Young. *Topology.* Addison-Wesley, Reading, MA, 1961.

[16] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science—Volume A: Algorithms and Complexity.* MIT Press, Cambridge, Mass., 1990.

[17] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2.* Lawrence Livermore National Laboratory, Mar 1985.

[18] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. A unified geometric approach to graph separators. In *Proceedings Symposium on Foundations of Computer Science*, 1991.

[19] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proceedings Symposium on Foundations of Computer Science*, pages 478–489, October 1985.

[20] Peter H. Mills, Lars S Nyland, Jan F. Prins, John H. Reif, and Robert A. Wagner. Prototyping parallel and distributed programs in Proteus. Technical Report Computer Science UNC-CH TR90-041, University of North Carolina, 1990.

[21] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Mass., 1990.

[22] Rishiyur S. Nikhil. Id reference manual. Technical Report Computation Structures Group Memo 284-1, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1990.

[23] Franco P. Preparata and Michael I. Shamos. *Computational Geometry—An Introduction.* Springer-Verlag, New York, 1985.

[24] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel Max-Flow algorithm. *J. Algorithms*, 3:128–146, 1982.

[25] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[26] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.