Programming Parallel Algorithms

Guy E. Blelloch

Keywords: Data-parallel, parallel algorithms, supercomputers, nested parallelism, PRAM model, parallel programming languages, collection-oriented languages.

Note: *This paper is also available on the World-Wide Web at:*
`http://web.scandal.cs.cmu.edu/www/cacm.html`.
The Web version is interactive and allows the reader to try the parallel algorithms presented in this paper.

In the past 20 years there has been tremendous progress in developing and analyzing parallel algorithms. Researchers have developed efficient parallel algorithms to solve most problems for which efficient sequential solutions are known. Although some of these algorithms are efficient only in a theoretical framework, many are quite efficient in practice or have key ideas that have been used in efficient implementations. This research on parallel algorithms has not only improved our general understanding of parallelism, but in several cases has led to improvements in sequential algorithms. Unfortunately there has been less success in developing good languages for programming parallel algorithms, particularly languages that are well suited for teaching and prototyping algorithms. There has been a large gap between languages that are too low level, requiring specification of many details that obscure the meaning of the algorithm, and languages that are too high-level, making the performance implications of various constructs unclear. In sequential computing many standard languages such as C or Pascal do a reasonable job of bridging this gap, but in parallel languages building such a bridge has been significantly more difficult.

Our research involves developing a parallel language that is useful for teaching as well as for implementing parallel algorithms. To achieve this, an important goal has been to develop a language that allows high-level descriptions of parallel algorithms while also having a well understood mapping onto a performance model (*i.e.* bridges the gap). Based on our research, we believe that the following two features are important for achieving this goal:

- A language-based performance model that uses *work* and *depth* rather than a machine-based model that uses "running time".
- Support for *nested data-parallel* constructs. This is the ability to apply a function in parallel to each element of a collection of data, and the ability to nest such parallel calls.

In this paper we describe these features and explain why they are important for programming parallel algorithms. To make the ideas concrete, we describe the programming language NESL [5], which we designed based on the features, and go through several examples of how to program and analyze parallel algorithms using the language. We have been using NESL for three years in undergraduate and graduate courses on parallel algorithms [8]. The algorithms we cover in this paper are relatively straightforward. Many more algorithms can be found through the Web version of this paper.

The paper is organized as follows. Section 1 introduces and motivates the use of *work* and *depth* as complexity measures. Section 2 introduces the NESL language and the use of nested data parallelism. It also includes a definition of NESL's performance model. Section 3 gives examples of several parallel algorithms. We assume a basic understanding of the performance analysis of sequential algorithms.

1 Work and Depth

Analyzing performance is a key part of studying algorithms. Although such analysis is not used to predict the exact running time of an algorithm on a particular machine, it is important in determining how the running time grows as a function of the input size. To analyze performance a formal model is needed to account for the costs. In parallel computing the most common models are based on a set of processors connected either by a shared memory, as in the Parallel Random Access Machines (PRAM, see Figure 1), or through a network, as with the hypercube or grid models. In such *processor-based models*, performance is calculated in terms of the number of instruction cycles a computation takes (its running time) and is usually expressed as a function of input size and number of processors.

An important advance in parallel computing was the introduction of the notion of *virtual models*. A virtual model is a performance model that does not attempt to represent any machine that we would actually build, but rather is a higher-level model that can then be mapped onto various real machines. For example, the PRAM is often viewed as a virtual model [25]. From this viewpoint, it is agreed that a PRAM cannot be built directly since in practice it is unreasonable to assume that every processor can access a shared memory in unit time. Instead, the PRAM is treated as a virtual machine that can be mapped onto more realistic machines efficiently by simulating multiple processors of the PRAM on a single processor of a host machine. This simulation imposes some slowdown K, but requires K fewer processors, so the total cost (processor-time product) remains the same. The advantage of virtual models over physical machine models is that they can be easier to program.

Virtual models can be taken a step further and used to define performance in more abstract measures than just running time on a particular machine. A pair of such measures are work and depth: *work* is defined as the total number of operations executed by a computation, and *depth* is defined as the longest chain of sequential dependencies in the computation. Consider, for example, summing 16 numbers using a balanced binary tree (see Figure 2). The work required by this computation is 15 operations (the 15 additions). The depth of the computation is 4 operations since the longest chain of dependencies is the depth of the summation tree—the sums need to be calculated starting at the leaves and going down one level at a time. In general summing n numbers on a balanced tree requires $n-1$ work and $\log_2 n$ depth. Work is usually viewed as a measure of the total cost of a computation (integral of needed resources over time), and also specifies the running time if the algorithm is executed on a sequential processor. The depth represents the best possible running time, assuming an ideal machine with an unlimited number of processors.

Work and depth have been used informally for many years to describe the performance of parallel algorithms [23], especially when teaching them [17, 16]. The claim is that it is easier to describe, think about, and analyze algorithms in terms of work and depth rather than in terms of running time on a processor-based model (a model based on P processors). Furthermore, work and depth together tell us a lot about expected performance on various machines. We will return to these points, but we first describe in more detail how work and depth can be incorporated into a computational model. There are basically three classes of such models—circuit models, vector machine models, and language-based models—and we briefly describe each.

Circuit Models: In circuit models an algorithm is specified by designing a circuit of logic gates to solve the problem. The circuits are restricted to have no cycles. For example we could view Figure 2 as a circuit in which the inputs are at the top, each $+$ is an adder circuit, and each of the lines between adders is a bundle of wires. The final sum is returned at the bottom. In circuit models the circuit size (number of gates) corresponds to *work*, and the longest path from an input to an output corresponds to *depth*. Although for a particular input size one could build a circuit to implement an algorithm, in general circuit models are viewed as virtual models from which the size and depth of the designs tell us something about the performance of algorithms on real machines. As such, the models have been used for many years to study various theoretical aspects of parallelism, for example to prove that certain problems are hard to solve in parallel (see [17] for an overview). Although the models are well suited for such theoretical analysis, they are not a convenient model for programming parallel algorithms.

Vector Machine Models: The first programmable machine model based on work and depth was the Vector Random Access Machine (VRAM) [4]. The VRAM model is a sequential random-access machine (RAM) extended with a set of instructions that operate on vectors (see Figure 3). Each location of the memory contains a whole vector, and the vectors can vary in size during the computation. The vector instructions include element-wise operations, such as adding the corresponding elements of two vectors, and aggregate operations, such as extracting elements from one vector based on another vector of indices. The depth of a computation in a VRAM is simply the number of instructions executed by the machine, and the work is calculated by summing the lengths of the vectors on which the computation operates. As an example, Figure 3 shows VRAM code for taking the sum of n values. This code executes the summation tree in Figure 2—each loop iteration moves down the tree one level. The VRAM is again a virtual model since it would be impractical to build the vector memory because of its dynamic nature. Although the VRAM is a good model for describing many algorithms that use vectors or arrays, it is not an ideal model for directly expressing algorithms on more complicated data structures, such as trees or graphs.

Language-Based Models: A third choice for defining a model in terms of work and depth is to define it directly in terms of language constructs. Such a *language-based performance model* specifies the costs of the primitive instructions, and a set of rules for composing costs across program expressions. The use of language-based models is certainly not new. Aho and Ullman, in their popular introductory text book "Foundations of Computer Science" [2], define such a model for deriving running times of sequential algorithms. The approach allows them to discuss the running time of the algorithms without introducing a machine model. A similar approach can be taken to define a model based on work and depth. In this approach work and depth costs are assigned to each primitive instruction of a language and rules are specified for combining both parallel and sequential expressions. Roughly speaking, when executing a set of tasks in parallel the total work is the sum of the work of the tasks and the total depth is the maximum of the depth of the tasks. When executing tasks sequentially, both the work and depth are summed. These rules are made more concrete when we describe NESL's performance model in Section 2.2, and the algorithms in the paper illustrate many examples of how the rules can be applied.

We note that language-based performance models seem to be significantly more important for parallel algorithms than for sequential algorithms. Unlike Aho and Ullman's sequential model which corresponds almost directly to a machine model (the Random Access Machine) and is defined purely for convenience, there seems to be no satisfactory machine model that captures the notion of work and depth in a general way.

Why Work and Depth? We now return to the question of why models based on work and depth are better than processor-based models for programming and analyzing parallel algorithms. To motivate this claim we consider a particular algorithm, *Quicksort*, and compare the code and performance analysis of a parallel version of the algorithm using the two types of models. We argue that in the work-depth model the code is very simple, the performance analysis is closely related to the code, and the code captures the notion of parallelism in Quicksort at a very high level. This is not true with the processor-based model.

We start by reviewing sequential Quicksort, for which pseudocode is shown in Figure 4. A standard performance analysis proves that for n keys the algorithm runs in $O(n \log n)$ time on average (expected case). A similar analysis proves that the maximum depth of recursive calls is $O(\log n)$ expected case—we will use this fact later. Quicksort is not hard to parallelize. In particular, we can execute the two recursive calls in parallel, and furthermore, within a single Quicksort we can compare all the elements of S to the pivot a in parallel when subselecting the elements for S_1, and similarly for S_2 and S_3. The questions remain of how we program this parallel version and what is its performance?

We first consider programming and analyzing parallel Quicksort with a model based on work and depth. Figure 5 illustrates the NESL code for the algorithm. This code should be compared with the sequential pseudocode—the only significant difference is that the NESL code specifies that the subselection for `S1`, `S2` and `S3`, and the two recursive calls to `Quicksort` should be executed in parallel (in NESL curly brackets {} signify parallel execution). Since the parallel algorithm does basically the same operations as the sequential version, the work cost of the parallel version is within a small constant factor of the time of the sequential version ($O(n \log n)$ expected case). The depth cost of the algorithm can be analyzed by examining the recursion tree in Figure 5. The depth of each of the blocks represents the sum of the depths of all the operations in a single call to Quicksort (not including the two recursive calls). These operations are the test for termination, finding the pivot `a`, generating `S1`, `S2` and `S3`, and the two appends at the end. As discussed in more detail in Section 2.2 each of these operations has constant depth (*i.e.* is fully parallel). The depth of each block is therefore constant, and the total depth is this constant times the maximum number of levels of recursion, which we mentioned earlier is $O(\log n)$ expected case. This completes our analysis of Quicksort and says that the work of Quicksort is $O(n \lg n)$ and the depth is $O(\lg n)$, both expected case.[1] Note that we have derived performance measures for the algorithm based on very high-level code and without talking about processors.

We now consider code and analysis for parallel Quicksort based on a parallel machine model with P processors. We claim that in such a model the code will be very long, will obscure the high-level intuition of the algorithm and will make it hard to analyze the performance of the algorithm. In particular the code will have to specify how the sequence is partitioned across processors (in general the input length does not equal P and needs to be broken up into parts), how the subselection is implemented in parallel (for generating S_1, S_2 and S_3 in parallel), how the recursive calls get partitioned among the processors and then load-balanced, how the sub-calls synchronize, and many other details. This is complicated by the fact that in Quicksort the recursive calls are typically

[1] We note that the parallel version of Quicksort requires more memory than a good implementation of the sequential version. In particular the sequential version can be implemented in place while the parallel version requires about n scratch space.

not of equal sizes, the recursion tree is not balanced, and the S_2 sets have to be reinserted on the way back up the recursion. Although coding these details might help optimize the algorithm for a particular machine, they have little to do with core ideas. Even if we assume the simplest model with unit-time access to shared memory and built-in synchronization primitives, the fully parallel code for Quicksort in just about any language would require hundreds of lines of code. This is not just a question of verbosity but is a question of how we think about the algorithm.

Relationship of work and depth to running time. Work and depth can be viewed as the running time of an algorithm at two limits: one processor (work) and an unlimited number of processors (depth). In fact, the complexities are often referred to as T_1 and T_∞. In practice, however, we want to know the running time for some fixed number of processors. A simple but important result of Brent [9] showed that knowing the two limits is good enough to place reasonable bounds on running time for any fixed number of processors. In particular he showed that if we know that a computation has work W and depth D then it will run with P processors in time T such that

$$\frac{W}{P} \leq T < \frac{W}{P} + D \ .$$

This result makes some assumptions about communication and scheduling costs, but the equation can be modified if these assumptions change. For example, with a machine that has a memory latency of L (the time between making a remote request and receiving the reply), the equation is $W/P \leq T \leq W/P + L \cdot D$.

Let's return to the example of summing. Brent's equation along with our previous analysis of work and depth ($W = n-1$, $D = \log_2 n$) tell us that n numbers can be summed on P processors within the time bounds

$$\frac{(n-1)}{P} \leq T < \frac{(n-1)}{P} + \log_2 n \ .$$

For example 1,000,000 elements can be summed on 1000 processors in somewhere between 1000 ($10^6/10^3$) and 1020 ($10^6/10^3 + \log_2 10^6$) cycles, assuming we count one cycle per addition. For many parallel machines models, such as the PRAM or a set of processors connected by a hypercube network, this is indeed the case. To implement the addition, we could assign 1000 elements to each processor and sum them, which would take 999 cycles. We could then sum across the processors using a tree of depth $\log_2 1000 = 10$, so the total number of add cycles would be 1009, which is within our bounds.

Communication Costs A problem with using work and depth as cost measures is that they do not directly account for communication costs and can lead to bad predictions of running time on machines where communication is a bottleneck. To address this question, let's separate communication costs into two parts: *latency*, as defined above, and *bandwidth*, the rate at which a processor can access memory. If we assume that each processor may have multiple outstanding requests, then latency is not a problem. In particular, latency can be accounted for in the mapping of the work and depth into time for a machine (see above), and the simulation remains work-efficient (*i.e.* the processor-time product is proportional to the total work). This is based on hiding the latency by using few enough processors such that on average each processor has multiple parallel tasks (threads) to execute, and therefore has plenty to do while waiting for replies. Bandwidth is a more serious problem. For machines where the bandwidth between processors is very much less than the bandwidth to the local memory, work and depth by themselves will not in general give good predictions of running time. However, the network bandwidth available on recent parallel machines, such as the Cray T3D and SGI Power Challenge, is high enough to give reasonable predictions, and we expect the situation only to improve with rapidly improving network technology.

2 Nested Data-Parallelism and NESL

Many constructs have been suggested for expressing parallelism in programming languages, including fork-and-join constructs, data-parallel constructs, and futures, among others. The question is which of these are most useful for specifying parallel algorithms? If we look at the parallel algorithms that are described in the literature and their pseudocode, we find that nearly all are described as parallel operations over collections of values. For example "in parallel for each vertex in a graph, find its minimum neighbor", or "in parallel for each row in a matrix, sum the row". Of course the algorithms are not this simple—they usually consist of many such parallel calls interleaved with operations that rearrange the order of a collection, and can be called recursively in parallel, as in Quicksort. This ability to operate in parallel over sets of data is often referred to as *data-parallelism* [15], and languages based on it are often referred to as data-parallel languages or *collection-oriented* languages [24]. We note that many parallel languages have data-parallel features in conjunction with other forms of parallelism [10, 3, 12, 18].

Before we come to the rash conclusion that data-parallel languages are the panacea for programming parallel algorithms, we make a distinction between flat and nested data-parallel languages. In *flat data-parallel languages* a function can be applied in parallel over a set of values, but the function itself must be sequential. In *nested data-parallel languages* [4] any function can be applied over a set of values, including parallel functions. For example, the summation of each row of the matrix mentioned above could itself execute in parallel using a tree sum. We claim that the ability to nest parallel calls is critical for expressing algorithms in a way that matches our high-level intuition of how they work. In particular, nested parallelism can be used to implement nested loops and divide-and-conquer algorithms in parallel (five out of the seven algorithms described in this paper use nesting in a crucial way). The importance of allowing nesting in data-parallel languages has also been observed by others [13]. However, most existing data-parallel languages, such as High Performance Fortran (HPF) [14] or C* [21], do not have direct support for such nesting.[2]

2.1 NESL

This paper uses NESL [5] as an example of a nested data-parallel language. This section gives an overview of the language and Section 3 gives several examples of parallel algorithms described and analyzed with NESL. NESL was designed to express nested parallelism in a simple way with a minimum set of structures and was therefore designed as a language on its own rather than as an extension of an existing sequential language. The ideas, however, can clearly be used in other languages. NESL is loosely based on ML [19], a language with a powerful type system, and on SETL [22], a language designed for concisely expressing sequential algorithms. As with ML, NESL is mostly functional (has only limited forms of side effects), but this feature is tangential to the points made in this paper.

NESL supports data-parallelism by means of operations on sequences—one dimensional arrays. All elements of a sequence must be of the same type, and sequence indices are zero based (`a[0]` extracts the first element of the sequence `a`). The main data-parallel construct is *apply-to-each*, which uses a set-like notation. For example, the expression

```
{a * a :  a in [3, -4, -9, 5]};
```

squares each elements of the sequence `[3, -4, -9, 5]` returning the sequence `[9, 16, 81, 25]`. This can be read: "in parallel, for each `a` in the sequence `[3, -4, -9, 5]`, square `a`". The apply-to-each can be used over multiple sequences. The expression

```
{a + b :  a in [3, -4, -9, 5]; b in [1, 2, 3, 4]};
```

adds the two sequences elementwise returning `[4, -2, -6, 9]`. The apply-to-each construct also provides the ability to subselect elements of a sequence based on a filter. For example

```
{a * a :  a in [3, -4, -9, 5] | a > 0};
```

can be read: "in parallel, for each `a` in the sequence `[3, -4, -9, 5]` such that `a` is greater than `0`, square `a`". It returns the sequence `[9, 25]`. The elements that remain maintain their relative order. This filtering was used in the Quicksort example.

Any function, whether primitive or user defined, may be applied to each element of a sequence. So, for example, we could define

[2] The current HPF 1.0 has some limited support for nested calls and future versions are likely to have significantly better support.

```
function factorial(n) =
  if (n == 1) then 1
  else n*factorial(n-1);
```

and then apply it over the elements of a sequence as in

```
{factorial(i) : i in [3, 1, 7]};
```

which returns the sequence `[6, 1, 5040]`.

In addition to the parallelism supplied by apply-to-each, NESL provides a set of functions on sequences, each of which can be implemented in parallel. For example the function `sum` adds the elements of a sequence, and the function `reverse` reverses the elements of a sequence. Perhaps the most important function on sequences is `write`, which supplies the only mechanism to modify multiple values of a sequence in parallel. `write` takes two arguments: the first is the sequence to modify and the second is a sequence of integer-value pairs that specify what to modify. For each pair `(i,v)` the value `v` is inserted into position `i` of the destination sequence. For example

```
write([0, 0, 0, 0, 0, 0, 0, 0], [(4,-2),(2,5),(5,9)]);
```

inserts the `-2`, `5` and `9` into the sequence at locations 4, 2 and 5, respectively, returning

```
[0, 0, 5, 0, -2, 9, 0, 0] .
```

If an index is repeated, then one value is written non-deterministically. For readers familiar with the variants of the PRAM model, we note that the `write` function is analogous to an "arbitrary" concurrent write. NESL also includes a function `e_write` that does not allow repeated indices, and is analogous to an exclusive write. If repeated indices are used with `e_write`, the current implementation reports an error.

Nested parallelism is supplied in NESL by allowing sequences to be nested and allowing parallel functions to be used in an apply-to-each. For example, we could apply the `sum` function in parallel over a nested sequence, as in

```
{sum(a) : a in [[2,3], [8,3,9], [7]]}
```

which would return `[5, 20, 7]`. Here there is parallelism both within each sum and across the sums. The Quicksort algorithm showed another example of nested calls—the algorithm is itself used in an apply-to-each to invoke two recursive calls in parallel.

2.2 The Performance Model

We now return to the issue of performance models, this time in the context of NESL. As mentioned earlier, NESL defines work and depth in terms of the work and depth of the primitive operations and rules for composing the measures across expressions. We will use $W(e)$ and $D(e)$ to refer to the work and depth of evaluating an expression e. In most cases the work and depth of an expression are the sums of the work and depth of the subexpressions. So, for example, if we have an expression $e_1 + e_2$ where e_1 and e_2 are subexpressions then the work of the expression is

$$W(e_1 + e_2) = 1 + W(e_1) + W(e_2)$$

where the 1 is the cost of the add. A similar rule is used for depth. The interesting rules concerning parallelism are the rules for an apply-to-each expression:

$$W(\{e_1(a) : a \texttt{ in } e_2\}) = 1 + W(e_2) + \sum_{a \texttt{ in } e_2} W(e_1(a)) \tag{1}$$

$$D(\{e_1(a) : a \texttt{ in } e_2\}) = 1 + D(e_2) + \max_{a \texttt{ in } e_2} D(e_1(a)) \tag{2}$$

The first rule specifies that the work is the sum of the work of each of the applications of e_1 to a_i, plus the work of e_2, plus 1 to account for overheads. The rule for depth is similar, but takes the maximum of the depth of each application of e_1. This supports our intuition that the applications are executed in parallel and the evaluation of the apply-to-each completes when the last call completes. The other interesting rules are the rules for an `if` expression, which for work is

$$W(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) = 1 + W(e_1) + \begin{cases} W(e_2) & e_1 = \text{TRUE} \\ W(e_3) & \textit{otherwise} \end{cases} \tag{3}$$

with a similar rule for depth. The work and depth for a function call and for scalar primitives are 1 each. The costs of the NESL functions on sequences are summarized in Table 7. We note that the performance rules can be more precisely defined using an operational semantics [7].

As an example of composing work and depth, consider evaluating the expression

$e =$ `{factorial(n) : n in a}`

where `a = [3,1,5,2]`. Using the rules for work and the code for `factorial` given earlier, we can write the following equation for work:

$$W_{\texttt{fact}}(n) = 1 + 1 + W_{\texttt{==}} + \begin{cases} 0 & n = 1 \\ W_{\texttt{*}} + W_{\texttt{-}} + W_{\texttt{fact}}(n-1) & n > 1 \end{cases}$$

where $W_{\texttt{==}}$, $W_{\texttt{*}}$, and $W_{\texttt{-}}$ are the work for `==`, `*`, and `-`, and are all 1. The two unit constants come from the cost of the function call and the *if-then-else* rule. Adding up the terms and solving the recurrence gives $W_{\texttt{fact}}(n) = 5n - 2$. Since there is no parallelism in the factorial function, the depth is the same as the work. To calculate work and depth for the full expression `{factorial(n) : n in a}` we can use equations 1 and 2. This calculation is shown in Figure 6.

3 Examples of Parallel Algorithms in NESL

This section describes and analyzes several parallel algorithms. These algorithms provide examples of how to analyze algorithms in terms of work and depth and of how to use nested data-parallel constructs. They also introduce some important ideas concerning parallel algorithms. We mention again that the main goals are to have the code closely match the high-level intuition of the algorithm, and to make it easy to analyze the asymptotic performance from the code.

3.1 Primes

Our first algorithm finds all the prime numbers less than n. This example demonstrates a common technique used in parallel algorithms—solving a smaller case of the same problem to speed the solution of the full problem. We also use the example to introduce the notion of work efficiency. An important aspect of developing a good parallel algorithm is designing one whose work is close to the time for a good sequential algorithm that solves the same problem. Without this condition we cannot hope to get good speedup of the parallel algorithm over the sequential algorithm. Parallel algorithms are referred to as *work-efficient* relative to a sequential algorithm if their work is within a constant factor of the time of the sequential algorithm. All the algorithms we have discussed so far are work efficient relative to the best sequential algorithms. In particular summing n numbers took $O(n)$ work and parallel Quicksort took $O(n \log n)$ expected work, both of which are the same as required sequentially. For finding primes our goal should again be to develop a work-efficient algorithm. We therefore start by looking at efficient sequential algorithms.

The most common sequential algorithm for finding primes is the sieve of Eratosthenes, which is specified in Figure 8. The algorithm returns an array in which the i^{th} position is set to true if i is a prime and to false otherwise. The algorithm works by initializing the array A to TRUE and then setting to FALSE all multiples of each prime it finds. It starts with the first prime, 2, and works it way up to $\sqrt{n}$. The algorithm only needs go up to $\sqrt{n}$ since all composite numbers (non-primes) less than n must have a factor less or equal to $\sqrt{n}$. If line 7 is implemented by looping over the multiples, then the algorithm can be shown to take $O(n \log\log n)$ time and the constant is small. The sieve of Eratosthenes is not the theoretically best algorithm for finding primes, but it is close and we would be happy to derive a parallel algorithm that is work-efficient relative to it (*i.e.* does $O(n \log\log n)$ work).

It turns out that the algorithm as described has some easy parallelism. In particular, line 7 can be implemented in parallel. In NESL the multiples of a value i can be generated in parallel with the expression

```
[2*i:n:i]
```

and can be written into the array A in parallel with the `write` function. Using the rules for costs (see Table 7), the depth of these operations is constant and the work is the number of multiples, which is the same as the time of the sequential version. Given the parallel implementation of line 7, the total work of the algorithm is the same as the sequential algorithm since it does the same number of operations, and the depth of the algorithm is $O(\sqrt{n})$ since each iteration of the loop in lines 5-8 has constant depth and the number of iterations is $\sqrt{n}$. Note that thinking of the algorithm in terms of work and depth allows a simple analysis (assuming we know the running time of the sequential algorithm) without having to worry about how the parallelism maps onto a machine. In particular, the amount of parallelism varies greatly from the first iteration, in which we have $n/2$ multiples of 2 to knock out in parallel, to the last iteration where we have only $\sqrt{n}$ multiples. This varying parallelism would make it messy to program and analyze on a processor based model.

We now consider improving the depth of the algorithm without giving up any work. We note that if we were given all the primes from 2 up to $\sqrt{n}$, we could then generate all the multiples of these primes at once. The NESL code for generating all the multiples is

```
{[2*p:n:p]:  p in sqr_primes};
```

where `sqr_primes` is a sequence containing all the primes up to $\sqrt{n}$. This computation has nested parallelism, since there is parallelism across the `sqr_primes` (outer parallelism) and also in generating the multiples of each prime (inner parallelism). The depth of the computation is constant since each subcall has constant depth, and the work is $O(n \log\log n)$ since the total number of multiples when summed across the subcalls is the same as the number of multiples used by the sequential version.

We have assumed that `sqr_primes` was given, but to generate these primes we can simply call the algorithm recursively on $\sqrt{n}$. Figure 9 shows the full algorithm for finding primes based on this idea. Instead of returning a sequence of flags, the algorithm returns a sequence with the values of the primes. For example `primes(10)` would return the sequence `[2,3,5,7]`. The algorithm recursively calls itself on a problem of size $\sqrt{n}$ and terminates when a problem of size 2 is reached. The work and depth can be analyzed by looking at the picture at the bottom of Figure 9. Clearly most of the work is done at the top level of recursion, which does $O(n \log\log n)$ work. The total work is therefore also $O(n \log\log n)$. Now let's consider the depth. Since each recursion level has constant depth, the total depth is proportional to the number of levels. To calculate this number we note that the size of the problem at level i is $n^{1/2^i}$, and that when the size is 2, the algorithm terminates. This gives us the equation $n^{1/2^d} = 2$, where d is the depth we seek. Solving for d, this method gives $d = \log\log n$. The costs are therefore:

$$\begin{aligned} Work &= O(n \log\log n) \\ Depth &= O(\log\log n) \end{aligned}$$

This algorithm remains work efficient relative to the sequential sieve of Eratosthenes and greatly improves the depth. It is actually possible to improve the depth to a constant, but we will leave this as an exercise for the reader.

3.2 Sparse Matrix Multiplication

Sparse matrices, which are common in scientific applications, are matrices in which most elements are zero. To save space and running time it is critical to only store the nonzero elements. A standard representation of sparse matrices in sequential languages is to use an array with one element per row each of which contains a linked-list of the nonzero values in that row along with their column number. A similar representation can be used in parallel. In NESL a sparse matrix can be represented as a sequence of rows, each of which is a sequence of `(column-number, value)` pairs of the nonzero values in the row. The matrix

$$A = \begin{bmatrix} 2.0 & -1.0 & 0 & 0 \\ -1.0 & 2.0 & -1.0 & 0 \\ 0 & -1.0 & 2.0 & -1.0 \\ 0 & 0 & -1.0 & 2.0 \end{bmatrix}$$

is represented in this way as

```
A = [[(0, 2.0), (1, -1.0)],
     [(0, -1.0), (1, 2.0), (2, -1.0)],
     [(1, -1.0), (2, 2.0), (3, -1.0)],
     [(2, -1.0), (3, 2.0)]]
```

where `A` is a nested sequence. This representation can be used for matrices with arbitrary patterns of nonzero elements since each subsequence can be of a different size.

A common operation on sparse matrices is to multiply them by a dense vector. In such an operation, the result is the dot-product of each sparse row of the matrix with the dense vector. The NESL code for taking the dot-product of a sparse row with a dense vector `x` is:

```
sum({v * x[i] :  (i,v) in row})
```

This code takes each index-value pair `(i,v)` in the sparse row, multiplies `v` with the $\mathtt{i}^{th}$ value of `x`, and sums the results. The work and depth is easily calculated using the performance rules. If n is the number of non-zeros in the row, then the depth of the computation is the depth of the `sum`, which is $O(\log n)$, and the work is the sum of the work across the elements, which is $O(n)$.

The full code for multiplying a sparse matrix A represented as above by a dense vector `x` requires that we apply the above code to each row in parallel, which gives

```
{sum({v * x[i] :  (i,v) in row})
  :  row in A}
```

This example has nested parallelism since there is parallelism both across the rows and within each row for the dot products. The total depth of the code is the maximum of the depth of the dot products, which is the logarithm of the size of the largest row. The total work is proportional to the total number of nonzero elements.

3.3 Planar Convex-Hull

Our next example solves the planar convex hull problem: Given n points in a plane, find which of them lie on the perimeter of the smallest convex region that contains all points. This example shows another use of nested parallelism for divide-and-conquer algorithms. The algorithm we use is a parallel *Quickhull* [20], so named because of its similarity to the Quicksort algorithm. As with Quicksort, the strategy is to pick a "pivot" element, split the data based on the pivot, and recurse on 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 requires $O(n^2)$ work, however in practice the algorithm is often very efficient.

Figure 10 shows the code and an example of the Quickhull algorithm. 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 that lie on the hull clockwise from `p1` to `p2`, inclusive of `p1`, but not of `p2`. In Figure 10, given all the points `[A, B, C, ..., P]`, `p1 = A` and `p2 = P`, `hsplit` would return the sequence `[A, 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 `[P, N, C]`.

The `hsplit` function first removes all the elements that cannot be on the hull because they lie below the line between `p1` and `p2` (which we denote by `p1-p2`). This is done by removing elements whose cross product with the line between `p1` and `p2` is negative. In the case `p1 = A` and `p2 = P`, the points `[B, D, F, G, H, J, K, M, O]` would remain and be placed in the sequence `packed`. The algorithm now finds the point, `pm`, furthest from the line `p1-p2`. The 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` flattens the result, thereby appending the two subhulls.

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 constant depth and $O(n)$ work. However, since many points might be deleted on each step, the work could be significantly less. As with Quicksort, the worst case costs are $W = O(n^2)$ and $D = O(n)$. For m hull points the best case times are $O(\log m)$ depth and $O(n)$ work. It is hard to state the average case time since it depends on the distribution of the inputs. Other parallel algorithms for the convex-hull problem run in $D = O(\log n)$, and $W = O(n)$ in the worst case [16], but have larger constants.

3.4 Three Other Algorithms

We conclude our examples with brief discussions of three other algorithms: the fast Fourier transform (FFT), the scan operation (all prefix sums), and an algorithm for finding the k^{th} smallest element of a set. All the code is shown in Figure 11. These algorithms give more demonstration of the conciseness of nested data parallel constructs.

We use the standard recursive version for the fast Fourier transform (FFT) [11]. The second argument `w` is a sequence of the same length as `a` containing all the complex nth roots of unity. The FFT is called recursively on the odd and even elements of `a`. The results are then combined using `cadd` and `cmult` (complex addition and multiplication). Assuming that `cadd` and `cmult` take constant work and depth, then the recursion gives us the costs:

$$\begin{aligned} W(n) &= 2W(n/2) + kn = O(n \log n) \\ D(n) &= D(n/2) + k = O(\log n) \end{aligned}$$

The plus-scan operation (also called all-prefix-sums) takes a sequence of values and returns a sequence of equal length for which each element is the sum of all previous elements in the original sequence. For example, executing a plus-scan on the sequence `[3, 5, 3, 1, 6]` returns `[0, 3, 8, 11, 12]`. This can be implemented as shown in Figure 11. The algorithm works by elementwise adding the odd and even elements and recursively solving the problem on these sums. The result of the recursive call is then used to generate all the prefix sums. The costs are:

$$\begin{aligned} W(n) &= W(n/2) + kn = O(n) \\ D(n) &= D(n/2) + k = O(\log n) \end{aligned}$$

The particular code shown only works on sequences that have a length equal to a power of two, but it is not hard to generalize it to work on sequences of any length.

A variation of Quicksort can be used to find the $\mathtt{k}^{th}$ smallest element of a sequence [11]. This algorithm only calls itself recursively on the set of elements containing the result. Here we consider a parallel version of this algorithm. After selecting the `lesser` elements, if `#lesser` is greater than `k`, then the $\mathtt{k}^{th}$ smallest element must belong to that set. In this case, the algorithm calls `kth_smallest` recursively on `lesser` using the same `k`. Otherwise, the algorithm selects the elements that are greater than the pivot, and can similarly find if the $\mathtt{k}^{th}$ element belongs in `greater`. If it does belong in `greater`, the algorithm calls itself recursively, but must now readjust `k` by subtracting the number of elements less than or equal to the pivot. If the $\mathtt{k}^{th}$ element belongs in neither `lesser` nor `greater`, then it must be the pivot, and the algorithm returns this value. For sequences of length n the expected work of this algorithm is $O(n)$, which is the same as the time of the serial version. The expected depth is $O(\log n)$ since the expected depth of recursion is $O(\log n)$.

4 Summary

The NESL language was designed to be useful for programming and teaching parallel algorithms. For these purposes it was important that it allowed for simple descriptions of algorithms that closely match our high level intuition, and also supplied a well-defined model for analyzing performance. We believe the language has successfully achieved these goals. There are many aspects of NESL, and the purpose of this paper was to extract the two features that are most important for programming parallel algorithms. They are:

1. A performance model based on work and depth. An important aspect is that the model is defined directly in terms of language constructs rather than trying to appeal to any intuition of a machine. As discussed, the model is a virtual one for which we give mappings onto running times for various physical machine models.

2. The use of data-parallel constructs for expressing parallelism, and the ability to nest such constructs. We certainly do not mean to exclude any other parallel constructs, but having some way of mapping a function over a set of values in parallel seems critical for expressing many parallel algorithms.

This paper is suggesting a change in the underlying models we use for analyzing parallel algorithms. In particular it suggests that we move away from using theoretical performance models based on machines to models based on languages. As mentioned in the paper, some reference works already informally analyze parallel algorithms in terms of work and depth before mapping them onto a PRAM [17, 16]. We suggest that the extra step is taken of formalizing a model based on work and depth. With this formal model the PRAM can be cut out of the loop, directly mapping the model onto more realistic machines. We furthermore argue that language-based models seem to be the most reasonable way to define a programming model based on work and depth.

A full implementation of NESL is currently available on the Web. The compiler is based on a technique called flattening nested parallelism [4], and compiles to an intermediate language called VCODE. Benchmark results for this implementation for the Connection Machines CM-2 and CM-5 and the Cray C90 are described in [6]. These results show that NESL's performance is competitive with that of machine-specific codes for those benchmarks.

Acknowledgments

I would like to thank Marco Zagha, Uzi Vishkin, Jay Sipelstein, Margaret Reid-Miller, Takis Metaxas, Bob Harper, Jonathan Hardwick, John Greiner, Jacques Cohen, and Siddhartha Chatterjee for many helpful comments on this article. Siddhartha Chatterjee, Jonathan Hardwick, Jay Sipelstein, and Marco Zagha helped in the design of NESL and did all the work implementing the intermediate languages VCODE and CVL.

This research was sponsored in part by the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330, and in part by an NSF Young Investigator Award.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science.* Computer Science Press, New York, 1992.

[3] 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.

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

[5] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[6] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[7] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the Symposium on Functional Programming and Computer Architecture*, June 1995.

[8] Guy E. Blelloch and Jonathan C. Hardwick. Class notes: Programming parallel algorithms. Technical Report CMU-CS-93-115, School of Computer Science, Carnegie Mellon University, February 1993.

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

[10] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

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

[12] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[13] Phil Hatcher, Walter F. Tichy, and Michael Philippsen. A critique of the programming language C*. *Communications of the ACM*, 35(6):21–24, June 1992.

[14] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[15] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12), December 1986.

[16] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, Reading, Mass., 1992.

[17] Richard M. Karp and Vijaya 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.

[18] 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 UNC-CH TR90-041, Computer Science Department, University of North Carolina, 1990.

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

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

[21] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. In *Proceedings Second International Conference on Supercomputing, Vol. 2*, pages 2–16, May 1987.

[22] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, 1986.

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

[24] Jay Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[25] Uzi Vishkin. Parallel-design distributed-implementation (PDDI) general purpose computer. *Theoretical Computer Science*, 32:157–172, 1984.

Author Information Guy Blelloch is an associate professor in the School of Computer Science at Carnegie Mellon University. His research interests is concerned with practical issues in the design and implementation of parallel algorithms. Issues of interest include parallel primitives, languages, compilers and implementation techniques.

Present Address:
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
email: blelloch@cs.cmu.edu

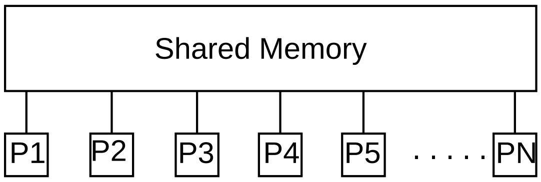


Figure 1: A diagram of a Parallel Random Access Machine (PRAM). It is assumed in this model that all the processors can access a memory location in the shared memory simultaneously in unit time.

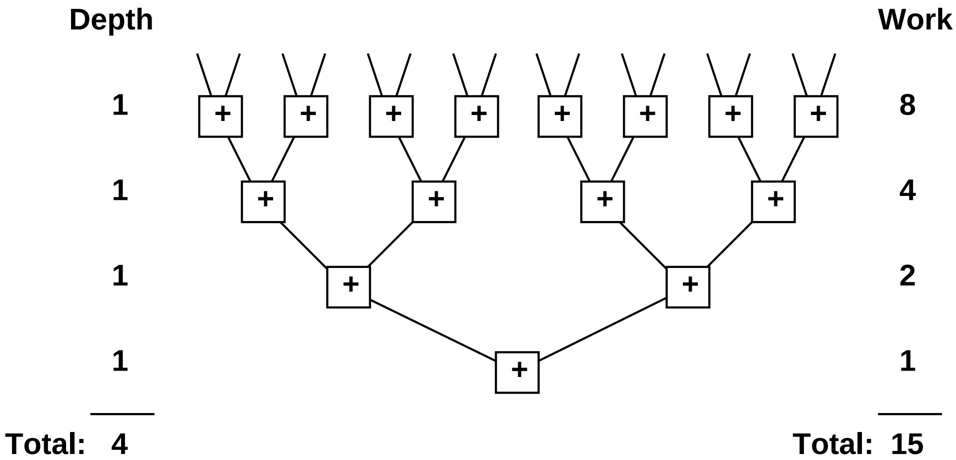
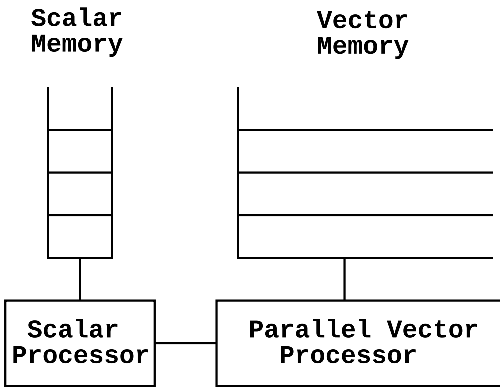


Figure 2: Summing 16 numbers on a tree. The total depth (longest chain of dependencies) is 4 and the total work (number of operations) is 15.



procedure SUM(V):
n = `length`(V);
for $i = 1$ **to** $log_2 n$
  begin
    V_o = `odd_elts`(V);
    V_e = `even_elts`(V);
    V = `vector_add`(V_o, V_e);
  end
return V

Figure 3: A diagram of a Vector Random Access Machine (VRAM) and pseudocode for summing n numbers on the machine. The vector processor acts as a slave to the scalar processor. The functions `odd_elts` and `even_elts` extract the odd and even elements from a vector, respectively. The function `vector_add` elementwise adds two vectors. On each iteration through the loop the length of the vector `V` halves. The code assumes n is a power of 2, but it is not hard to generalize the code to work with any n. The total work done by the computation is $O(n + n/2 + n/4 + ..) = O(n)$ and the depth is a constant times the number of iterations, which is $O(\log n)$.

procedure QUICKSORT(S):
if S contains at most one element **then return** S
else
  begin
    choose an element a randomly from S;
    let S_1, S_2 and S_3 be the sequences of elements in S less
      than, equal to, and greater than a, respectively;
    return (QUICKSORT(S_1) followed by S_2 followed by
      QUICKSORT(S_3))
  end

Figure 4: Pseudocode for Quicksort from Aho, Hopcroft and Ullman, "The Design and Analysis of Computer Algorithms" [1]. Although originally described as a sequential algorithm, the algorithm as stated is not hard to parallelize.

```
function Quicksort(S) =
if (#S <= 1) then S
else
  let a  = S[rand(#S)];
      S1 = {e in S| e < a};
      S2 = {e in S| e == a};
      S3 = {e in S| e > a};
      R =  {Quicksort(v):  v in [S1,S3]};
  in R[0] ++ S2 ++ R[1];
```

$$\begin{aligned} Work &= O(n \log n) \text{ (expected)} \\ Depth &= O(\log n) \text{ (expected)} \end{aligned}$$

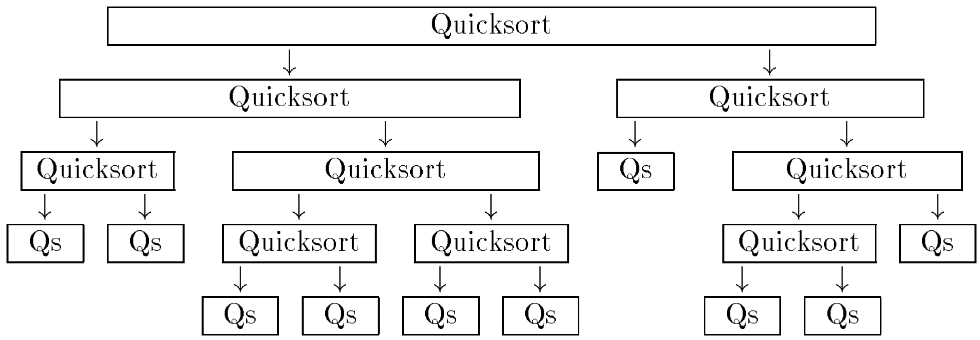
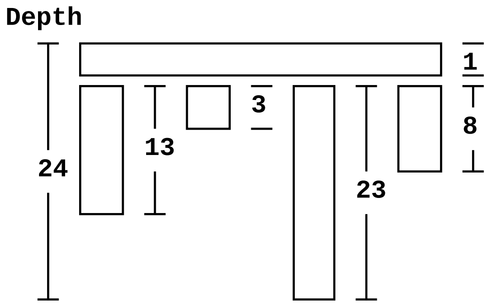


Figure 5: The Quicksort algorithm in NESL. The operator `#` returns the length of a sequence. The function `rand(n)` returns a random number between 0 and `n` (the expression `S[rand(#S)]` therefore returns a random element of `S`). The notation `{e in S| e < a}` is read: "in parallel find all elements `e` in `S` for which `e` is less than `a`". This operation has constant depth and work proportional to the length of `S`. The notation `{Quicksort(v): v in [S1,S3]}` is read: "in parallel for `v` in `S1` and `S3`, `Quicksort v`". The results are returned as a pair. The function `++` appends two sequences.



$$W_{\texttt{fact}}(n) = D_{\texttt{fact}}(n) = 5n - 2$$

$$\begin{aligned}
Work &= 1 + \ \textbf{sum}(W_{\texttt{fact}}(3), W_{\texttt{fact}}(1), W_{\texttt{fact}}(5), W_{\texttt{fact}}(2)) \\
&= 1 + \ \textbf{sum}(13, 3, 23, 8) \\
&= 48
\end{aligned}$$

$$\begin{aligned}
Depth &= 1 + \ \textbf{max}(D_{\texttt{fact}}(3), D_{\texttt{fact}}(1), D_{\texttt{fact}}(5), D_{\texttt{fact}}(2)) \\
&= 1 + \ \textbf{max}(13, 3, 23, 8) \\
&= 24
\end{aligned}$$

Figure 6: Calculating the work and depth of {`factorial(n) : n in [3,1,5,2]`}.

Operation	Description	Work	Depth
`dist(a,l)`	*Create a sequence of* `a`*s of length* `l`.	1	1
`#a`	*Return length of sequence* `a`.	1	1
`a[i]`	*Return element at position* `i` *of* `a`.	1	1
`[s:e]`	*Return integer sequence from* `s` *to* `e`.	`(e - s)`	1
`[s:e:d]`	*Return integer sequence from* `s` *to* `e` *by* `d`.	`(e - s)/d`	1
`sum(a)`	*Return sum of sequence* `a`.	L(`a`)	log L(`a`)
`write(d, a)`	*Place elements* `a` *in* `d`.	L(`a`)	1
`a ++ b`	*Append sequences* `a` *and* `b`.	L(`a`) + L(`b`)	1
`drop(a,n)`	*Drop first* `n` *elements of sequence* `a`.	L(`result`)	1
`interleave(a,b)`	*Interleave elements of sequences* `a` *and* `b`.	L(`result`)	1
`flatten(a)`	*Flatten nested sequence* `a`.	L(`result`)	1

Figure 7: List of some of the sequence functions supplied by NESL. The work required by each function is given in the Work column: L(`v`) refers to the length of the sequence `v`. The work of the write(d, a) function actually depends on whether the argument `d` needs to be copied or not, but in the examples in this paper the difference has no effect.

```
procedure PRIMES(n):
let A be an array of length n
set all but the first element of A to TRUE
for i from 2 to √n
    begin
        if A[i] is TRUE
        then set all multiples of i up to n to FALSE
    end
```

Figure 8: Pseudocode for the sieve of Eratosthenes

```
function primes(n) =
if n == 2 then ([] int)
else
  let sqr_primes  = primes(isqrt(n));
      composites  = {[2*p:n:p]:  p in sqr_primes};
      flat_comps  = flatten(composites);
      flags       = write(dist(true, n), {(i,false):  i in flat_comps});
      indices     = {i in [0:n]; fl in flags | fl}
  in drop(indices, 2);
```

Example for `primes(20)`:

```
sqr_primes   =   [2, 3]
composites   =   [[4,6,8,10,12,14,16,18],[6,9,12,15,18]]
flat_comps   =   [4,6,8,10,12,14,16,18,6,9,12,15,18]
flags        =   [t,t,t,t,f,t,f,t,f,f,f,t,f,t,f,f,f,t,f,t]
indices      =   [0,1,2,3,5,7,11,13,17,19]
result       =   [2,3,5,7,11,13,17,19]
```

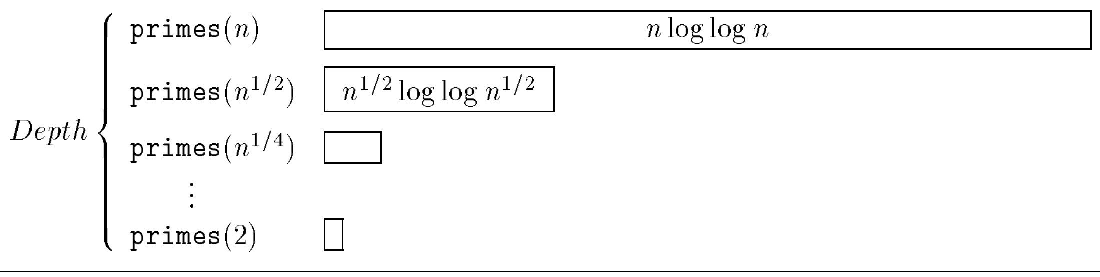
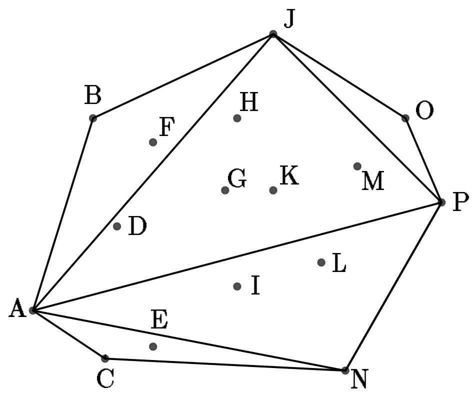


Figure 9: The code for the primes algorithm, an example of one level of the recursion, and a diagram of the work and depth. In the code `[] int` indicates an empty sequence of integers. The function `isqrt` takes the square root of an integer. The function `flatten` takes a nested sequence and flattens it. The function `dist(a,n)` distributes the value `a` to a sequence of length `n`. The expression `{i in [0:n]; fl in flags | fl}` can be read as "for each `i` from 0 to `n` and each `fl` in `flags` return the `i` if the corresponding `fl` is true". The function `drop(a,n)` drops the first `n` elements of the sequence `a`.

```
function cross_product(o,line) =
let (xo,yo) = o;
    ((x1,y1),(x2,y2)) = line
in (x1-xo)*(y2-yo) - (y1-yo)*(x2-xo);

function hsplit(points,p1,p2) =
let cross = {cross_product(p,(p1,p2)):  p in points};
    packed = {p:p in points; c in cross | plusp(c)}
in if (#packed < 2) then [p1] ++ packed
   else
     let pm = points[max_index(cross)]
     in flatten({hsplit(packed,p1,p2):
                 p1 in [p1,pm]; p2 in [pm,p2]});

function convex_hull(points) =
let x = {x :  (x,y) in points};
    minx = points[min_index(x)];
    maxx = points[max_index(x)]
in hsplit(points,minx,maxx) ++ hsplit(points,maxx,minx);
```



[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 10: Code and example of the *Quickhull* algorithm. Each sequence in the example 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.

```
function fft(a,w) =
if #a == 1 then a
else
  let r = {fft(b, even_elts(w)):
           b in [even_elts(a),odd_elts(a)]}
  in {cadd(a, cmult(b, w)):
      a in r[0] ++ r[0];
      b in r[1] ++ r[1];
      w in w};
```

$$\begin{aligned} Work &= O(n \log n) \\ Depth &= O(\log n) \end{aligned}$$

```
function scan(a) =
if #a == 1 then [0]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan({e + o:  e in e; o in o})
  in interleave(s,{s + e:  s in s; e in e});
```

$$\begin{aligned} Work &= O(n) \\ Depth &= O(\log n) \end{aligned}$$

```
function kth_smallest(s, k) =
let pivot  = s[#s/2];
    lesser = {e in s| e < pivot};
    greater = {e in s| e > pivot};
in if (k < #lesser) then
     kth_smallest(lesser, k)
   else if (k >= #s - #greater) then
     kth_smallest(greater, k - (#s - #greater))
   else pivot;
```

$$\begin{aligned} Work &= O(n) (\text{expected}) \\ Depth &= O(\log n) (\text{expected}) \end{aligned}$$

Figure 11: Code for the fast Fourier transforms, the scan operation, and for finding the k^{th} smallest element of a set.