A Provable Time and Space Efficient Implementation of NESL

Guy E. Blelloch and John Greiner
Carnegie Mellon University
{blelloch,jdg}@cs.cmu.edu

Abstract

In this paper we prove time and space bounds for the implementation of the programming language Nesl on various parallel machine models. Nesl is a sugared typed λ-calculus with a set of array primitives and an explicit parallel map over arrays. Our results extend previous work on provable implementation bounds for functional languages by considering space and by including arrays. For modeling the cost of Nesl we augment a standard call-by-value operational semantics to return two cost measures: a DAG representing the sequential dependences in the computation, and a measure of the space taken by a sequential implementation. We show that a Nesl program with w work (nodes in the DAG), d depth (levels in the DAG), and s sequential space can be implemented on a p processor butterfly network, hypercube, or CRCW PRAM using $O(w/p + d \log p)$ time and $O(s + dp \log p)$ reachable space.[1] For programs with sufficient parallelism these bounds are optimal in that they give linear speedup and use space within a constant factor of the sequential space.

1 Introduction

This paper presents a provably time and space efficient implementation of the parallel programming language Nesl [6]. Nesl is a strongly typed call-by-value functional language loosely based on ML. It has been implemented on several parallel machines [8], and has been used both for teaching parallel algorithms [9, 7], and implementing various applications [17, 4, 1]. The parallelism in the language is based on including a primitive sequence data type, a parallel map operation, and a small set of primitive operations on sequences. To be useful for analyzing parallel algorithms, Nesl was designed with rules for calculating the *work* (the total number of operations executed) and *depth* (the longest chain of sequential dependences) of a computation. These are standard cost measures in the analysis of parallel algorithms [23, 22]. In this paper we formalize these rules and give provable implementation bounds for both space and time.

[1] The implementation is based on a randomized algorithm for the fetch-and-add operator and will therefore run within the given time with high probability.

The idea of a provably efficient implementation is to add to the semantics of the language an accounting of costs, and then to prove a mapping of these costs into running time and/or space of the implementation on concrete machine models (or possibly to costs in other languages). The motivation is to assure that the costs of a program are well defined and to make guarantees about the performance of the implementation. In previous work we have studied provably time efficient parallel implementations of the λ-calculus using both call-by-value [3] and speculative parallelism [18]. These results accounted for work and depth of a computation using a profiling semantics [29, 30] and then related work and depth to running time on various machine models.

This paper applies these ideas to the language Nesl and extends the work in two ways. First, it includes sequences (arrays) as a primitive data type and accounts for them in both the cost semantics and the implementation. This is motivated by the fact that arrays cannot be simulated efficiently in the λ-calculus without arrays (the simulation of an array of length n using recursive types requires a $\Omega(\log n)$ slowdown). Second, it augments the profiling semantics with a cost measure for space and proves bounds on the space needed by the implementation based on this measure. These bounds show that for programs with sufficient parallelism, the parallel execution requires very little extra memory beyond a standard call-by-value sequential execution. These space bounds use recent results on DAG scheduling [2] and are non trivial. Although we use these extensions to prove bounds for Nesl, the techniques and results can be applied in a broader context. In particular we translate Nesl into a generic array language which could be used to express other array extensions, and the space bounds we derive can be applied with minor changes to most languages with fork-join style parallelism, including the call-by-value λ-calculus [3].

Cost model. To model time and space in the semantics, we augment a standard call-by-value operational semantics to return two cost measures. The first is a DAG which represents the sequential control dependences in the program. The number of levels in the DAG is the depth of a computation, and the number of nodes in the DAG is the work of the computation. Although the operational semantics itself is sequential, the rules for combining DAGs explicitly define what constructs are parallel. This is similar to Hudak and Anderson's [21] use of pomsets (partially ordered multisets) to add intensional information on execution order to the denotational semantics of the λ-calculus. The second cost measure is an accounting of the reachable space used by a sequential implementation of the language. This includes both space for values as well as space for any control structures—it assumes a reasonably good sequential implementation. To account for the sharing of space the semantics uses a store.

Intermediate language and machine. To simplify the proof of the implementation bounds we introduce an intermediate language and machine. The intermediate language, which we refer to as the *array language*, makes array allocation explicit and breaks certain array operations into atomic operations. In Section 4 we define the array language and give a translation from the core of NESL (Core-NESL) to the language.

Our intermediate machine, the P-CEK(q) machine, evaluates the array language. The machine works by keeping a queue of states (threads) that are ready to execute and on each step processes the first q of these states in parallel. The parameter q represents the parallelism of the machine—to allow for multithreading this is typically slightly larger than the number of processors of the target machine. The fact that only q states are processed on each step and the order in which the states are processed play a crucial role in proving the space bounds. Picking an arbitrary set of q states would not be space efficient. The P-CEK(1) machine ($q = 1$) is a sequential machine and closely corresponds to the CESK machine [13]. Section 5 defines the P-CEK(q) machine.

Results. Our results are derived, by mapping the Core-NESL costs first to the array language, then to the P-CEK(q) machine and finally to the target machines. We show that a Core-NESL program with w work, d depth and s sequential space will run in $O(w/p + d \log p)$ time and $O(s + dp \log p)$ space, on p processors of either a butterfly network, hypercube or CRCW PRAM. All machine models are randomized, so the time bounds are with high probability. The randomization is needed to implement routing of messages and the fetch-and-add operation within the specified bounds. The space bounds refer to the reachable space and do not include the cost of garbage collection. Determining whether garbage collection can be performed within the specified bounds is an interesting area of future research. More detail on our assumptions about the machines are given in Section 6.

We note that the implementation discussed in this paper does not correspond to the current implementation of NESL [8]. The current implementation is based on a technique called flattening nested parallelism [5], which has very good performance characteristics, but can be space inefficient because it generates too much parallelism. For example the NESL code

```
{count({a < b: a in s}) : b in s},
```

which calculates the rank of each key in a sequence `s`, creates n^2 parallelism ($|\mathtt{s}| = n$). In the current implementation this would require $O(n^2)$ space, while in the implementation suggested in this paper it would require only $O(n + p \log p)$ space since the depth is $O(1)$. We are currently studying whether we can combine the ideas from the two implementations.

2 The Language NESL

NESL is a nested data-parallel language designed for programming parallel algorithms at a very high level. The goal in the design was to make parallel algorithms look as close as possible to standard pseudocode. NESL has a polymorphic type system, a call-by-value semantics, and other than I/O and random number generation, is purely functional. Parallelism in NESL is explicit—it uses a set of parallel primitives on arrays and a parallel map construct. The parallel map construct maps any expression over the elements of an array in parallel. Since an expression can itself have parallel calls, this allows for the nesting of parallel calls. Such nested parallelism is crucial for expressing parallel divide-and-conquer or nested parallel loops (most data-parallel languages don't permit nesting [19, 28]). For the purpose of analyzing algorithms, the definition of NESL includes rules for calculating the work and depth of a computation. These rules specify the work and depth for the primitives, and how these costs can be composed across expressions. For example, when mapping a function over an array of data, the total work is the sum of the work of the individual applications, and the total depth is the maximum of the depths of the individual applications. Figure 1 shows an example of NESL code for the quicksort algorithm, along with the complexities that can be derived from these rules. The rules are defined somewhat informally in the original language definition [6] and are formalized in this paper.

```
function Quicksort(S) =
if (#S <= 1) then S
else
  let a  = S[#S/2];
      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];
```

$$\begin{array}{lcll} \mathit{Work} & = & O(n \log n) & \text{(expected)} \\ \mathit{Depth} & = & O(\log n) & \text{(expected)} \\ \mathit{Space} & = & O(n) & \text{(expected)} \end{array}$$

Figure 1: The Quicksort algorithm in NESL. The operator `#` returns the length of a sequence. The expression `{e in S| e < a}` subselects all elements of `S` less than `a` in parallel. This operation has constant depth and work proportional to the length of `S`. The expression `{Quicksort(v): v in [S1,S3]}` applies quicksort to `S1` and `S3` in parallel—the depth is the maximum of the depth of the two recursive calls. The function `++` appends two sequences.

To simplify the presentation of this paper we consider a language Core-NESL instead of the full NESL. Core-NESL includes a representative sample of the data types, constants, and primitive functions of NESL. For data types, it includes integers, booleans, pairs, homogeneous arrays, and functions—it omits floating-point numbers, characters, and user defined datatypes. For constants and primitive functions, we choose a set that is sufficient to simulate the full set over the included data types with only constant overhead, except for input/output and pseudo-random number generation. In NESL, input/output is not permitted in parallel calls, so it could be added to the semantics by using streams and threading them through the computation. Pseudo-random number generation can be added to the semantics by splitting the seed when making parallel calls. Core-NESL can also be strongly typed, although we do not examine any typing issues in this paper.

Core-NESL is a sugared λ-calculus extended with a set of scalar constants, a set of constant functions on arrays, and a parallel map operation. Its syntax is defined as follows:

$$
\begin{array}{rcll}
c & ::= & i \mid \textbf{true} \mid \textbf{false} \mid \textbf{fst} \mid \textbf{snd} \mid & \text{(scalar const.)} \\
 & & + \mid - \mid * \mid / \mid < \mid & \\
 & & \# \mid \textbf{elt} \mid \textbf{index} \mid \textbf{pack} \mid & \text{(array const.)} \\
 & & \texttt{<-} \mid \textbf{addscan} \mid \textbf{maxscan} & \\
e & ::= & c \mid x \mid \lambda x.e \mid (e_1, e_2) \mid \textbf{if}\ e_1\ e_2\ e_3 \mid & \\
 & & e_1\ e_2 \mid \textbf{letrec}\ x'\ x = e_1\ \textbf{in}\ e_2 \mid & \\
 & & \{e' : x\ \textbf{in}\ e\} & \text{(parallel map)}
\end{array}
$$

where i, c, and e range over the integers, constants, and expressions, respectively. The scalar constants have their standard definitions. The function **#** returns the length of an array, and the function **elt** extracts an element from an array. The function **pack** takes an array of (value,flag) pairs. It returns an array containing only the values whose corresponding flag was true. The function <- writes a set of values into an array. In particular it takes a pair of arrays in which the first is a source array and the second is an array of (index,value) pairs. It copies the source array into a destination array, and in parallel writes each value into the corresponding index of the destination array. If array indices are duplicated, the rightmost (index,value) pair will always be written, which corresponds to the priority write scheme for the CRCW PRAM models. The two scan operations execute parallel prefixes and are used as primitives to implement various other array operations (such as flattening nested sequences).

NESL also has other syntactic forms not included in Core-NESL because they have simple translations to the core syntax. For example the notation $\{e' : x \textbf{ in } e_1 \mid e_2\}$, as used in the example of Figure 1, evaluates e' using all the elements in x such that e_2 is true. This can be translated to $\{e' : x \textbf{ in pack } \{(x, e_2) : x \textbf{ in } e_1\}\}$. Similarly $x[i]$ is just syntactic sugar for **elt** x i.

3 Core-NESL Semantics

Core-NESL's extensional semantics is based on that of a standard call-by-value functional language with arrays. We define an operational semantics for the language describing the evaluation of an expression to a value. In addition, the semantics also defines the intensional aspects of interest: the computation DAG and the space. To measure space usage accurately, we explicitly model data sharing and space allocation with the use of stores.

We now define the semantic domains and notation used. A *value* v is either a constant c or a location l. We thread a *store* through the semantics to map locations to store values sv, which can be *closures*, pairs, or arrays.

$$
\begin{array}{rcl}
v & ::= & c \mid l \\
sv & ::= & cl(E, x', x, e) \mid (v_1, v_2) \mid [v_0, \ldots, v_{n-1}]
\end{array}
$$

A closure represents a potentially recursive function defined via **letrec** and consists of its definition *environment*, its name, its bound variable, and its expression body. By including the function name in the closure, we avoid complicating the semantics with recursive environments. Arrays of values, locations, and other objects are used throughout the semantics, so we introduce some convenient notation for them—array $[v_0, \ldots, v_{n-1}]$ is also denoted $\vec{v}$, where $|\vec{v}|$ is its length n. An empty mapping is denoted by $\cdot$, and the extension of a mapping with a binding of x to v is denoted $E[x \mapsto v]$, where x may be in $dom(E)$. If defined, the element bound to x in E is $E(x)$. We also extend this notation to lookup sets of variables, returning the set of corresponding values.

$space(R, \sigma) =$

$$
\sum_{l \in S} \begin{cases} |\vec{v}| & \text{if } \sigma(l) = \vec{v} \\ 2 & \text{if } \sigma(l) = (v_1, v_2) \\ |FV(e)| + 2 & \text{if } \sigma(l) = cl(E, x', x, e) \end{cases}
$$

where $S = \bigcup_{l \in R} locs(l, \sigma)$

$$
\begin{array}{rcl}
locs(c, \sigma) & = & \{\} \\
locs(l, \sigma) & = & \{l\} \cup locs(\sigma(l), \sigma) \\
locs((v_1, v_2), \sigma) & = & locs(v_1, \sigma) \cup locs(v_2, \sigma) \\
locs(\vec{v}, \sigma) & = & \bigcup_{i=0}^{n-1} locs(v_i, \sigma) \\
 & & \text{where } n = |\vec{v}| \\
locs(cl(E, x', x, e), \sigma) & = & \bigcup_{l \in L} locs(l, \sigma) \\
 & & \text{where } L = E(FV(e) - \{x, x'\})
\end{array}
$$

Figure 4: Semantics functions used in the profiling semantics for defining space. The *space* function defines the amount of space reachable from a set of roots R. It uses the *locs* function which defines the locations reachable from a location. $FV(e)$ denotes the set of free variables that appear in e.

The semantics relation, written $E, \sigma, R \vdash e \stackrel{\lambda}{\Longrightarrow} v, \sigma'; g, s$, adds the costs of evaluation to a standard extensional semantics. The context of the evaluation consists of an environment E and store σ describing the memory, and a set R of root locations pointing to data needed by the continuation. In a given context, an expression evaluates to a value v and a new store σ', with a computation DAG g using s units of space. The relation is defined by the inference rules of Figure 2, using the definitions of DAG composition given in Figure 3 and space given in Figure 4. Figure 5 gives the semantics (δ) and cost (δ_g) of constant function application.

The DAG returned by the semantics represents the dependence graph of the computation, in which the nodes of the graph represent units of computation and the edges represent the control dependences. All data dependences are subsumed by these control dependences and are therefore not made explicit in the graph. When two computations are executed sequentially, the graph from one is attached in series with the graph from the other, and when a set of computations are executed in parallel, the graphs are connected in parallel. This means that all graphs returned by the semantics will be series-parallel and will have a single source and sink. The rules for composing graphs are given in Figure 3, where $\oplus$ represents sequential composition, and $\otimes$ represents parallel composition. As can be seen from where $\otimes$ is used in Figures 2 and 5, the only parallelism in the Core-NESL semantics is in the array primitives and the EACH rule. NESL does not execute the two expressions of a function application e_1 e_2 in parallel, as does the PAL model [3]. In the DAGs the ordering among the children of a node is important (it is needed for our space bounds) so our representation of DAGs keeps this information.

Given a DAG g returned by the semantics, its *work* w is the number of nodes in g and its depth d is the number of levels in g including the source and sink.

The space s returned by the semantics represents the

$$E, \sigma, R \vdash c \xRightarrow{\lambda} c, \sigma; \mathbf{1}, space(R, \sigma) \qquad \text{(CONST)}$$

$$E, \sigma, R \vdash x \xRightarrow{\lambda} E(x), \sigma; \mathbf{1}, space(R \cup \{E(x)\}, \sigma) \qquad \text{(VAR)}$$

$$E, \sigma, R \vdash \lambda x.e \xRightarrow{\lambda} l, \sigma[l \mapsto cl(E, \_, x, e)]; \mathbf{1}, space(R \cup \{l\}, \sigma) \quad \text{where } l \notin dom(\sigma) \qquad \text{(ABSTR)}$$

$$\frac{E, \sigma, R \cup E(FV(e_2)) \vdash e_1 \xRightarrow{\lambda} v_1, \sigma_1; g_1, s_1 \qquad E, \sigma_1, R \cup \{v_1\} \vdash e_2 \xRightarrow{\lambda} v_2, \sigma_2; g_2, s_2}{E, \sigma, R \vdash (e_1, e_2) \xRightarrow{\lambda} l, \sigma_2[l \mapsto (v_1, v_2)]; \mathbf{1} \oplus g_1 \oplus g_2 \oplus \mathbf{1}, \max(s_1, s_2) + 1} \text{ where } l \notin dom(\sigma) \qquad \text{(PAIR)}$$

$$\frac{E, \sigma, R \cup E(FV(e_2)) \cup E(FV(e_3)) \vdash e_1 \xRightarrow{\lambda} \mathbf{true}, \sigma_1; g_1, s_1 \qquad E, \sigma_1, R \vdash e_2 \xRightarrow{\lambda} v, \sigma_2; g_2, s_2}{E, \sigma, R \vdash \mathbf{if}\ e_1\ e_2\ e_3 \xRightarrow{\lambda} v, \sigma_2; \mathbf{1} \oplus g_1 \oplus g_2, \max(s_1 + 1, s_2)} \qquad \text{(IF-TRUE)}$$

$$\frac{E, \sigma, R \cup E(FV(e_2)) \cup E(FV(e_3)) \vdash e_1 \xRightarrow{\lambda} \mathbf{false}, \sigma_1; g_1, s_1 \qquad E, \sigma_1, R \vdash e_3 \xRightarrow{\lambda} v, \sigma_3; g_3, s_3}{E, \sigma, R \vdash \mathbf{if}\ e_1\ e_2\ e_3 \xRightarrow{\lambda} v, \sigma_3; \mathbf{1} \oplus g_1 \oplus g_3, \max(s_1 + 1, s_3)} \qquad \text{(IF-FALSE)}$$

$$\frac{E[x' \mapsto l], \sigma[l \mapsto cl(E, x', x, e_1)], R \vdash e_2 \xRightarrow{\lambda} v, \sigma'; g, s}{E, \sigma, R \vdash \mathbf{letrec}\ x'\ x = e_1\ \mathbf{in}\ e_2 \xRightarrow{\lambda} v, \sigma'; \mathbf{1} \oplus g, s + 1} \text{ where } l \notin dom(\sigma) \qquad \text{(LETREC)}$$

$$\frac{E, \sigma, R \cup E(FV(e_2)) \vdash e_1 \xRightarrow{\lambda} c, \sigma_1; g_1, s_1 \qquad E, \sigma_1, R \vdash e_2 \xRightarrow{\lambda} v_2, \sigma_2; g_2, s_2 \qquad \delta(c, v_2, \sigma_2) = v, \sigma_3}{E, \sigma, R \vdash e_1\ e_2 \xRightarrow{\lambda} v, \sigma_2 \cup \sigma_3; \mathbf{1} \oplus g_1 \oplus g_2 \oplus \delta_g(c, v_2, \sigma_2), \max(s_1 + 1, s_2 + 1, space(R \cup \{v\}, \sigma_3))} \qquad \text{(APPC)}$$

$$\frac{\begin{array}{c} E, \sigma, R \cup E(FV(e_2)) \vdash e_1 \xRightarrow{\lambda} l, \sigma_1; g_1, s_1 \qquad E, \sigma_1, R \cup \{l\} \vdash e_2 \xRightarrow{\lambda} v_2, \sigma_2; g_2, s_2 \\ \sigma_1(l) = cl(E', x', x, e) \qquad E'[x' \mapsto l][x \mapsto v_2], \sigma_2, R \vdash e \xRightarrow{\lambda} v, \sigma_3; g, s \end{array}}{E, \sigma, R \vdash e_1\ e_2 \xRightarrow{\lambda} v, \sigma_3; \mathbf{1} \oplus g_1 \oplus g_2 \oplus \mathbf{1} \oplus g, \max(s_1 + 1, s_2 + 1, s)} \qquad \text{(APP)}$$

$$\frac{\begin{array}{c} E, \sigma, R \cup E(FV(e')) \vdash e \xRightarrow{\lambda} l, \sigma_0; g, s \qquad \sigma_0(l) = \vec{v} \qquad n = |\vec{v}| \\ E[x \mapsto v_j], \sigma_j, R \cup E(FV(e') - \{x\}) \vdash e' \xRightarrow{\lambda} v'_j, \sigma_{j+1}; g_j, s'_j \qquad \forall j \in \{0, \ldots, n-1\} \end{array}}{E, \sigma, R \vdash \{e' : x\ \mathbf{in}\ e\} \xRightarrow{\lambda} l', \sigma_n[l' \mapsto \vec{v'}]; g \oplus (\bigotimes_{j=0}^{n-1} \mathbf{1}) \oplus (\bigotimes \vec{g}), \max(s, n + \max(\vec{s'}))} \text{ where } l' \notin dom(\sigma) \qquad \text{(EACH)}$$

Figure 2: Profiling semantics of Core-NESL.

DAG g	$\mathbf{1}$	$(n_1, n'_1, D_1) \oplus (n_2, n'_2, D_2)$	$\bigotimes_{j=0}^{m-1} (n_j, n'_j, D_j)$
(n, n', D)	$(n, n, \{\})$ where n is new	$(n_1, n'_2, \{(n'_1, [n_2])\} \cup D_1 \cup D_2)$	$(n, n', \{(n, \vec{n})\} \cup \bigcup_{j=0}^{m-1} \{(n'_j, [n'])\} \cup \bigcup_{j=0}^{m-1} D_j)$ where n, n' are new

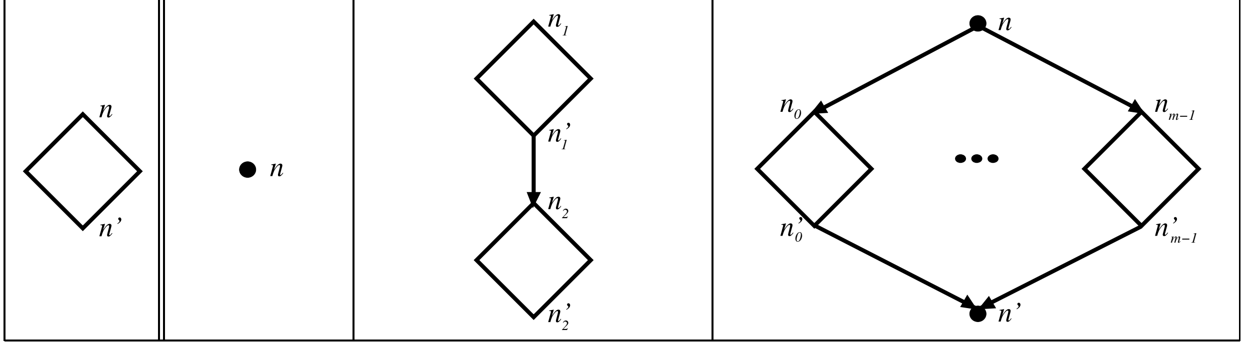

Figure 3: Definition of the unitary DAG ($\mathbf{1}$) and serial ($\oplus$) and parallel ($\otimes$) composition of DAGs. DAGs are represented as a triple (n, n', D) of the source node, the sink node, and a set of pairs, each consisting of a node and an ordered set (array) of its children. The newness conditions could be formalized easily by labelling nodes by the expressions they represent.

c	v	$\delta(c, v, \sigma) \ = \ v', \sigma'$	$\delta_g(c, v, \sigma)$
$+$	l	$i_1 + i_2 \quad \sigma$ if $\sigma(l) = (i_1, i_2)$	$\mathbf{1}$
elt	l	$v_j \quad \cdot$ if $\sigma(l) = (l', j),\ \sigma(l') = \vec{v}$	$\mathbf{1}$
#	l	$\lvert\vec{v}\rvert \quad \cdot$ if $\sigma(l) = \vec{v}$	$\mathbf{1}$
addscan	l	$l' \quad [l' \mapsto [\sum_{j=0}^{0} i_j, \ldots, \sum_{j=0}^{n-1} i_j]]$ if $\sigma(l) = \vec{i},\ n = \left\lvert\vec{i}\right\rvert,\ l' \notin dom(\sigma)$	$\bigotimes_{j=0}^{n-1} \mathbf{1}$
index	i	$l' \quad [l' \mapsto [0, \ldots, i-1]]$ if $l' \notin dom(\sigma)$	$\bigotimes_{j=0}^{i-1} \mathbf{1}$
pack	l	$l' \quad [l' \mapsto [v_{i_0}, \ldots, v_{i_{n-1}}]]$ if $\sigma(l) = \vec{l},\ n = \left\lvert\vec{l}\right\rvert,\ \sigma(l_j) = (v_j, c_j),$ $\{i_0, \ldots, i_{n'-1}\} = \{i \mid c_i = \mathbf{true}\},\ i_0 < \cdots < i_{n'-1},\ l' \notin dom(\sigma)$	$\bigotimes_{j=0}^{n-1} \mathbf{1}$
<-	l	$l' \quad [l' \mapsto \vec{v}[v'_0/i_0, \ldots, v'_{n'-1}/i_{n'-1}]]$ if $\sigma(l) = (l_1, l_2),\ \sigma(l_1) = \vec{v},\ n = \lvert\vec{v}\rvert,\ \sigma(l_2) = \vec{l'},\ n' = \left\lvert\vec{l'}\right\rvert,$ $\sigma(l'_j) = (i_j, v'_j),\ 0 \leq i_0, \ldots, i_{n'-1} < \lvert\vec{v}\rvert,\ l' \notin dom(\sigma)$	$(\bigotimes_{j=0}^{n-1} \mathbf{1}) \oplus (\bigotimes_{j=0}^{n'-1} \mathbf{1})$

Figure 5: The semantics (δ) and cost (δ_g) of constant function application. The semantics of the remaining primitives, -, *, /, <, **fst**, **snd**, and **maxscan**, are similar to those given. The substitution in the definition of <- gives priority to the last occurrence of any duplicate indices.

maximum reachable space during the computation. This is accounted for by tracking all the values that might be needed in the continuation (these are kept as a set of labels into the store). The R in the rules in Figure 2 keep these labels. For example, in a function application $e_1\ e_2$ when executing e_1 the semantics adds to the current set of labels the labels for the free variables in e_2. Given a set of root labels, the space required by the data is measured by finding all the locations reachable from the root locations R, and summing the space for each object stored at these labels (see Figure 4). Space is only measured at the leaves of the evaluation tree (in the rules CONST, VAR, ABSTR, and APPC). The addition of 1 to the space for many of the other rules represents space that is needed for control information and will be justified in Section 6.

In the EACH rule (for $\{e' : x \textbf{ in } e\}$) the expression e is evaluated to return a sequence, and the body is then evaluated for each element of the sequence. The store is threaded through the evaluations of body so as to specify a sequential order for which we will measure space (the space requirements can be different for different orders of execution). The execution of the subcomputations, however, are independent and therefore can be executed in parallel, at a cost of some extra space. The challenge is to show that the parallel execution does not require much more space than the specified sequential execution.

4 The Array Language

To simplify our abstract machine (Section 5) we translate the array instructions of Core-NESL to a lower-level language, which we refer to as the array language. We do this to make the memory allocation explicit, and to break up the array operations that take non-constant work into a set of tasks that each do constant work. The array language includes explicit side-effects since it needs to atomically update elements of arrays. The syntax of this language is the same as Core-NESL, except that we replace **index**, **pack**, <-, **addscan**, **maxscan**, and the parallel map expression with the constants **store**, **new**, **fork**, and **scanadd**, **scanmax**. Applying **store** to an array, an array of indices, and a value writes the value into the indexed locations of the array. Applying **new** to an integer creates and returns a new array of that length. And applying **fork** to an integer i and a function applies the function to each of $0, \ldots, i-1$ and returns a dummy value 0. Since the **fork** function returns a dummy value, it is only useful for any side-effects. The new scan operations compute the same as their counterparts, but allocate memory differently.

The semantics relation for the array language, written $E, \sigma, R \vdash e \stackrel{a}{\Longrightarrow} v, \sigma'; g, s$, is defined like that of Core-NESL, with the additions given in Figure 6.

The translation from Core-NESL array operations to the array language is given in Figure 7. The translation of the parallel map ($\{e' : x \textbf{ in } e\}$), for example, evaluates e, allocates a result array, and then forks n threads each of which applies e' to the appropriate element of e's value and writes its result in the appropriate element of the result array. The motivation for executing a **fork** in the definition of **allocate** is for bounding memory use (see the proof of Theorem 3). The following theorem states that the translation only affects the work, depth, and space by a constant factor.

Theorem 1 *If in* Core-NESL, $\cdot, \cdot, \{\} \vdash e \stackrel{\lambda}{\Longrightarrow} v, \sigma; g, s$, *then in the array language*, $\cdot, \cdot, \{\} \vdash T[\![e]\!] \stackrel{a}{\Longrightarrow} v', \sigma'; g', s'$ *such that*

1. *v' is the same as v, and σ' is an extension of σ, except that each may point to environments larger than those found in v and σ,*

2. *g' contains at most k more nodes and levels than g, and*

3. *$s' \leq ks$, for some constant k.*

c	v	$\delta(c, v, \sigma)$		$\delta_g(c, v, \sigma)$
store	l	v'	$\sigma[l_1 \mapsto \vec{v}[v'/i]]$	**1**
			if $\sigma(l) = (l_1, l_2),\ \sigma(l_1) = \vec{v},\ \sigma(l_2) = (i, v')$	
new	i	l	$\sigma[l \mapsto array(i)]$	**1**
			if $l \notin dom(\sigma)$	

$$\frac{\begin{array}{c} E, \sigma, R \cup E(FV(e_2)) \vdash e_1 \stackrel{a}{\Longrightarrow} \mathbf{fork}, \sigma'; g_1, s_1 \qquad E, \sigma', R \vdash e_2 \stackrel{a}{\Longrightarrow} l, \sigma_0; g_2, s_2 \\ \sigma_0(l) = (i, l') \qquad \sigma_0(l') = cl(E', x', x, e) \\ E'[x' \mapsto l'][x \mapsto j], \sigma_j, R \cup \{l'\} \vdash e \stackrel{a}{\Longrightarrow} v_j', \sigma_{j+1}; g_j', s_j' \qquad \forall j \in \{0, \ldots, i-1\} \end{array}}{E, \sigma, R \vdash e_1\ e_2 \stackrel{a}{\Longrightarrow} 0, \sigma_n; \mathbf{1} \oplus g_1 \oplus g_2 \oplus (\bigotimes \vec{g'}), \max(s_1, s_2, \vec{s'})} \quad \text{(FORK)}$$

Figure 6: The semantics rules for the constants of the array language. The function *array* creates and returns a new array of the specified length containing dummy values.

$$\begin{array}{l}
T[\![\{e' : x \textbf{ in } e\}]\!] = \\
\quad \textbf{let } x = T[\![e]\!] \\
\quad\quad n = \#\ x \\
\quad\quad r = \textbf{allocate } n \\
\quad\quad _ = \textbf{fork } (n, \lambda i.r[i] := (\lambda x.T[\![e']\!])\ x[i]) \\
\quad \textbf{in } r
\end{array}$$

$$\begin{array}{l}
T[\![\textbf{index}]\!] = \lambda x. \\
\quad \textbf{let } r = \textbf{allocate } x \\
\quad\quad _ = \textbf{fork } (n, \lambda i.r[i] := i) \\
\quad \textbf{in } r
\end{array}$$

$$\begin{array}{l}
T[\![\textbf{pack}]\!] = \lambda x. \\
\quad \textbf{let } z = T[\![\textbf{addscan } \{\textbf{if snd } (x')\ 1\ 0 : x' \textbf{ in } x\}]\!] \\
\quad\quad n = \#\ z \\
\quad\quad r = \textbf{allocate } z[n-1] \\
\quad\quad _ = \textbf{fork } (n, \lambda i.\textbf{if snd } (a[i])\ r[z[i]] := \textbf{fst } (a[i])\ 0) \\
\quad \textbf{in } r
\end{array}$$

$$\begin{array}{l}
T[\![\texttt{<-}]\!] = \lambda x. \\
\quad \textbf{let } a = \textbf{fst } x \\
\quad\quad ia = \textbf{snd } x \\
\quad\quad n = \#\ a \\
\quad\quad r = \textbf{allocate } n \\
\quad\quad _ = \textbf{fork } (n, \lambda i.r[i] := a[i]) \qquad \text{(copy dest. array)} \\
\quad\quad _ = \textbf{fork } (\#\ ia, \lambda i.r[\textbf{fst } ia[i]] := \textbf{snd } ia[i]) \\
\quad \textbf{in } r
\end{array}$$

$$\begin{array}{l}
T[\![\textbf{addscan}]\!] = \lambda x. \\
\quad \textbf{let } a = \textbf{allocate } (\#\ x) \\
\quad \textbf{in scanadd } (x, a)
\end{array}$$

$$\begin{array}{lrcl}
\text{where} & \textbf{allocate } n & = & \textbf{let } _ = \textbf{fork } (n, \lambda i.0) \textbf{ in new } n \\
& e_1[e_2] & = & \textbf{elt } (e_1, e_2) \\
& e_1[e_2] := e_3 & = & \textbf{store } (e_1, (e_2, e_3))
\end{array}$$

Figure 7: Translation from the Core-NESL array operations to the array language. We use a multiassignment **let** statement that executes the assignments in sequential order, which can also be translated into Core-NESL.

Proof Outline: This is proved by structural induction on the Core-NESL derivation, after generalizing the statement to hold for all evaluation contexts. □

5 The P-CEK(q) Machine

Our implementation of Core-NESL uses an intermediate abstract machine we call the P-CEK(q) machine. The P-CEK(q) machine executes a sequence of steps in which each step takes a queue of states and a store and processes a subset (of size at most q) of these states in parallel to generate a new queue of states and a modified store. Each state is processed by a transition similar to that of the CESK machine [13] (a variant of the SECD machine). The number of states grows as threads fork and shrinks as threads finish. In this section we define the machine and in the next section we prove bounds on the time and space it requires as a function of the work, depth and space of the Core-NESL semantics.

The P-CEK(q) machine is motivated by the P-ECD machine [3], which was used for proving bounds on the parallel implementation of the call-by-value λ-calculus (PAL model). However, because of the need to be space efficient and handle arrays, it has three important extensions. First, instead of processing the full queue of states on each step it only processes the first q states from the queue on each step. As will be shown in the next section, this modification greatly reduces the memory needs of the machine by reducing the parallelism. Second, the P-CEK(q) machine uses an explicit store. This is needed both to allow us to model the sharing of data in the accounting of space and to allow the machine to update array contents. Third, it allows for a state to fork an arbitrary number of children states in a single step instead of just a pair. This changes how threads synchronize since they have to synchronize as a group rather than as a pair. A fourth minor point is that in NESL function application $e_1\ e_2$ is defined to execute sequentially (as opposed to in parallel in the PAL model), and the machine reflects this. Modifying the P-CEK(q) machine to execute function application in parallel, however, would be a minor change.

The P-CEK(q) machine executes programs of the array language. Each machine step i is a transition of the form

$$Q_i, \sigma_i \stackrel{P(q)}{\Longrightarrow} Q_{i+1}, \sigma_{i+1}$$

where σ_i is a store, and Q_i is a queue of *substates*. Each substate S is a (C, E, K) triple which, together with the current store, corresponds roughly to state of the sequential CESK machine (C is an expression, E is an environment, and K is a *continuation*). The possible continuations are defined by

$$K \quad ::= \quad \mathsf{nil} \mid \langle \mathsf{arg}\ e\ E \rangle :: K \mid \langle \mathsf{fun}\ v \rangle :: K \mid \langle \mathsf{end}\ l \rangle :: K$$

These represent the empty continuation, a function's argument to be evaluated, a function to be applied, and a finishing thread, respectively.

To evaluate a program e, the machine starts with an empty store and a queue with a single substate $(e, \cdot, \mathsf{nil})$. On each step the P-CEK(q) machine takes $\min(q, |Q|)$ substates from the front of Q and processes a substate transition on each in parallel. Each substate can transition into a single new substate, fork off a group of new substates, or terminate. The new substates are collected together, maintaining their order relative to the original substates, and added back to the front of the queue. Maintaining the order is important for the space bounds. Substate transition can also return modifications for the store. These modifications are merged into the store for the next step. In addition to data the store is used to keep synchronization counters for detecting when a set of forked threads completes. The P-CEK(q) machine terminates with the single result substate $(\mathsf{exit}\ v, \cdot, \mathsf{nil})$, where v is the result value.

The transition rules for the P-CEK(q) are given in Figure 8. Each step is broken into two parts $\overset{P}{\Longrightarrow}_1$ and $\overset{P}{\Longrightarrow}_2$. The transition $\overset{P}{\Longrightarrow}_1$ is applied to each of the selected substates in parallel. It returns a new set of states and any modifications that need to be made to the store. The first five rules are minor modifications of standard CESK rules. They also restrict the domain of the environments to the variables that occur in their associated expression. This is needed to assure the space bounds. We leave out the rules for pairs, conditionals, and recursive bindings since these are minor modifications to the standard sequential semantics. The fork rule creates a set of i new substates (threads) and adds a synchronization counter l' to the store. Whenever a thread terminates it decrements the synchronization counter, and if the count goes to zero, then it applies the continuation. We note that in the fork and scanadd rules all but the last new substates created are marked with a prime. We will use these markings in our space bounds in the next section and otherwise they do not effect the semantics of the machine. The function *appK* applies a continuation, either creating a new state or an End object to be processed in the following transition. Any new locations created are guaranteed to be distinct from existing locations and from each other. We will account for the cost of getting new labels in the implementation.

The transition $\overset{P}{\Longrightarrow}_2$ is used for the "sequential" processing of the states—decrementing the synchronization counters and modifying the shared store. The store is therefore threaded though the substate transitions. As shown in the next section, however, these transitions can actually be implemented in parallel using a fetch-and-add operation and a priority write, and are therefore only threaded for the sake of the semantics.

The P-CEK(1) machine will always return the same value as the sequential semantics of the array language, but in general the P-CEK(q) machine ($q > 1$) can return a different value. This can happen if threads interact—for example, if one thread writes a value into an array that another parallel thread reads. The P-CEK(q) machine can possibly execute these in a different order than the P-CEK(1) machine. We claim, however, that this cannot happen with programs in the array language that are generated from Core-NESL. This is because the translations are specified so that each forked thread writes to a different array location and none of the locations are read by other threads. The one exception is in the rule for **store**. In this rule, however, the modifications will always occur in order (left to right) since in the semantics there is no way for an assignment on the right to execute before an assignment on the left.

6 Bounds on Time and Space

In this section we prove bounds on the space and time taken by the P-CEK(q) machine as a function of the work (size of DAG), depth (levels in DAG), and space measures returned by the array language profiling semantics. In particular we show the following:

1. **Sequential Space.** The space returned by the array language semantics is the same within a constant factor as the space required by the P-CEK(1) machine (the machine that only processes one state at a time). To prove this we formally define the space reachable by the P-CEK(q) machine. We note that in this paper we only consider the reachable space and therefore do not consider the cost of garbage collecting.

2. **Parallel Space.** There is a one-to-one correspondence between substates processed by the P-CEK(q) machine and nodes of the DAG returned by the array language semantics. Furthermore, we show that the P-CEK(q) machine executes a p depth-first traversal (p-DFT) of the DAG. This allows us to use previously results on DAG scheduling to show that the P-CEK(q) machine never schedules too many substates prematurely relative to the P-CEK(1) machine. This, in turn, allows us to bound the extra reachable space required by the P-CEK(q) machine. It also allows us to bound the number of steps taken by the P-CEK(q) machine.

3. **Parallel Time.** We show that each step of a P-CEK($p \log p$) machine can be implemented on the machine models (butterfly, hypercube, and PRAM) in $O(\log p)$ time, with high probability. Since we have a bound on the number of steps required by the machine, this allows us to bound the total running time for these machines.

By Theorem 1, all these bounds also hold for the Core-NESL semantics.

For brevity, we assume that each stage of the implementation preserves extensional correctness, *i.e.*, that evaluating an expression in each model results in the same value. Proving this would be straightforward using standard techniques. Furthermore, the following theorems assume that the array language evaluation derivation in question use expressions derived by the translation T. This is needed to ensure memory is allocated appropriately for the space bounds and to ensure the determinacy of the P-CEK(q) evaluations for the extensional correctness of these theorems (not shown here).

$$
\begin{array}{llllll}
(c, & E,\ K),\ \sigma & \overset{P}{\Longrightarrow}_1 & \mathit{appK}\ c\ K, & \cdot & \text{constant} \\
(x, & E,\ K),\ \sigma & \overset{P}{\Longrightarrow}_1 & \mathit{appK}\ E(x)\ K, & \cdot & \text{variable} \\
(\lambda x.e, & E,\ K),\ \sigma & \overset{P}{\Longrightarrow}_1 & \mathit{appK}\ l\ K, & [l \mapsto \mathit{cl}(E', \_, x, e)] & \text{abstraction} \\
 & & & \quad \text{if } l \notin \mathit{dom}(\sigma),\ E' = \mathit{restr}(E, e) & & \\
(e_1\ e_2, & E,\ K),\ \sigma & \overset{P}{\Longrightarrow}_1 & [[(e_1, E, \langle \mathsf{arg}\ e_2\ \mathit{restr}(E, e_2)\rangle :: K)]], & \cdot & \text{apply} \\
(@\ l\ v, & \cdot,\ K),\ \sigma & \overset{P}{\Longrightarrow}_1 & [[(e, E'[x' \mapsto l][x \mapsto v], K)]], & \cdot & \text{func-call} \\
 & & & \quad \text{if } \sigma(l) = \mathit{cl}(E', x', x, e) & & \\
(@\ \mathbf{fork}\ l, & \cdot,\ K),\ \sigma & \overset{P}{\Longrightarrow}_1 & [(@\ v\ 0, \cdot, K')', (@\ v\ 1, \cdot, K')', \ldots, & & \text{fork} \\
 & & & \ (@\ v\ (i-1), \cdot, K')], & [l' \mapsto i] & \\
 & & & \quad \text{if } \sigma(l) = (i, v),\ K' = \langle \mathsf{end}\ l'\rangle :: K,\ l' \notin \mathit{dom}(\sigma) & & \\
(@\ \mathbf{scanadd}\ l, & \cdot,\ K),\ \sigma & \overset{P}{\Longrightarrow}_1 & [(\Sigma\ l_1\ l_2\ l'\ 0, \cdot, K)', (\Sigma\ l_1\ l_2\ l'\ 1, \cdot, K)', \ldots, & & \text{scanadd} \\
 & & & \ (\Sigma\ l_1\ l_2\ l'\ (n-1), \cdot, K)], & [l' \mapsto 0] & \\
 & & & \quad \text{if } \sigma(l) = (l_1, l_2),\ n = |\sigma(l_1)|,\ l' \notin \mathit{dom}(\sigma) & & \\
(@\ c\ v, & \cdot,\ K),\ \sigma & \overset{P}{\Longrightarrow}_1 & \mathit{appK}\ v'\ K, & \sigma' & \text{prim-call} \\
 & & & \quad \text{if } \delta(c, v, \sigma) = v', \sigma' & & \\
(\Sigma\ l_1\ l_2\ l'\ i, & \cdot,\ K),\ \sigma & \overset{P}{\Longrightarrow}_1 & \mathbf{\Sigma}(l_1, l_2, l', i, K), & \cdot & \text{scanadd'}
\end{array}
$$

$$
\begin{array}{llcl}
\text{where} & \mathit{appK}\ v\ \ \mathsf{nil} & = & [[(\mathsf{exit}\ v, \cdot, \mathsf{nil})]] \\
 & \mathit{appK}\ v\ \ \langle \mathsf{arg}\ e\ E\rangle :: K & = & [[(e, E, \langle \mathsf{fun}\ v\rangle :: K)]] \\
 & \mathit{appK}\ v_2\ \langle \mathsf{fun}\ v_1\rangle :: K & = & [[(@\ v_1\ v_2, \cdot, K)]] \\
 & \mathit{appK}\ v\ \ \langle \mathsf{end}\ l\rangle :: K & = & \mathsf{End}(l, K)
\end{array}
$$

$\mathit{restr}(E, e) =$ the environment E restricted to the free variables in e.

$$
\begin{array}{llll}
(\mathsf{End}(l, K), & \sigma'),\ A,\ \sigma & \overset{P}{\Longrightarrow}_2 & \begin{cases} A +\!\!+ [[(0, \cdot, K)]], \sigma \cup \sigma' & \sigma(l) = 1 \quad \text{(last thread of parallel map)} \\ A, \sigma[l \mapsto \sigma(l) - 1] \cup \sigma' & \sigma(l) \neq 1 \end{cases} \\
(\mathbf{\Sigma}(l_1, l_2, l', i, K), & \cdot),\ A,\ \sigma & \overset{P}{\Longrightarrow}_2 & \begin{cases} A +\!\!+ [[(x, [x \mapsto l_2], K)]], \sigma' & i = \left|\vec{i'}\right| - 1 \quad \text{(last element to sum)} \\ A, \sigma' & i \neq \left|\vec{i'}\right| - 1 \end{cases} \\
 & & & \text{where } \sigma(l_1) = \vec{i'},\ \sigma(l') = j,\ \sigma' = \sigma[l_2 \mapsto \sigma(l_2)[j/i]][l' \mapsto j + i'_i] \\
(X, & \sigma'),\ A,\ \sigma & \overset{P}{\Longrightarrow}_2 & A +\!\!+ X, \sigma \cup \sigma'
\end{array}
$$

where $\sigma \cup \sigma'$ is the union of the stores such that if a location appears in both stores, the binding from the right store is kept.

$$
[S_1, \ldots, S_n], \sigma \overset{P(q)}{\Longrightarrow} A_{q'} +\!\!+ [S_{q+1}, \ldots, S_n],\ \sigma_{q'}
$$

where $q' = \min(q, n)$, $A_0 = []$, $\sigma_0 = \sigma$, and for $1 \leq j \leq q'$,

$$
\begin{array}{rcll}
S_j, \sigma & \overset{P}{\Longrightarrow}_1 & X_j, \sigma'_j & \text{(Execute these independently in parallel)} \\
(X_j, \sigma'_j), A_{j-1}, \sigma_{j-1} & \overset{P}{\Longrightarrow}_2 & A_j, \sigma_j & \text{(Execute these ``sequentially'')}
\end{array}
$$

where the new locations of $\sigma'_0, \ldots, \sigma'_j$ are renamed if necessary to ensure they are distinct.

Figure 8: Definition of the P-CEK(q) machine, omitting the definition of the evaluation of **scanmax**, pairs, conditionals, and recursive bindings. Arrays, pairs, closures, and synchronization counters are kept in the store. The expressions $@\ v\ v'$ and $\Sigma\ l_1\ l_2\ l'\ i$ are for internal use in evaluating applications and scans, respectively. In the latter, l_1 and l_2 point to the source and destination arrays, respectively, l' is the running total, and i is the index.

$$L(Q) = \bigcup_{(e,E,K)\in Q} (L_E(E) \cup L_K(K))$$

$$\begin{aligned}
L_K(\mathsf{nil}) &= \{\} \\
L_K(\langle \mathsf{arg}\ e\ E\rangle :: K) &= L_E(E) \cup L_K(K) \\
L_K(\langle \mathsf{fun}\ v\rangle :: K) &= \{v\} \cup L_K(K) \\
L_K(\langle \mathsf{end}\ l\rangle :: K) &= L_K(K)
\end{aligned}$$

Figure 9: Definitions for the *root values* $L(Q)$ of a step of the P-CEK(q) machine. This is a set of values, where labels act as roots into the store. The function $L_E(E)$ returns the range of the environment, and $L_K(K)$ returns the root values of a continuation.

Sequential Space. To capture the space that is required to implement the P-CEK(q) machine we need to consider on each step both the space that is required for the queue (Q) as well as any space in the store σ that can be reached via some label in the queue. As mentioned in the previous section, certain substates in the queue are marked with a prime (see the fork rule in Figure 8). We assign no space for these substates since in our implementation we will store a set of forked states in a compacted form—as a record with a start count, end count, and pointers to the common function to apply, environment and continuation. We assign the space for this structure to the last of the forked substates, which the semantics does not put a prime on, and therefore do not include any space for the other substates. This is necessary for the parallel space bounds. For each of the non primed CEK substates we include the following space: for the command (C) we include constant space, for the environment (E) we include space proportional to the size of the domain, and for the stack of continuations (K) we include space proportional to the number of entries in the stack (here we are just accounting for space required by the queue itself and not for any values that are in the store). To find the root labels into the store we consider all labels accessible either though an environment or continuation of any of the substates.

Definition 1 *The* reachable space $S_r(Q_i, \sigma_i)$ *of a step* i *of the* P-CEK(q) *machine is the sum of:*

1. *the* queue space $S_q(Q_i)$ *is the sum of* $(1 + |E| + |K|)$ *for those non-primed states* $(e, E, K) \in Q_i$*, and*
2. *the* store space $S_\sigma(Q_i, \sigma_i) = space(L(Q_i), \sigma_i)$

where $space(\cdot,\cdot)$ *and* $L(\cdot)$ *are defined in Figures 4 and 9, respectively,* $|E|$ *is the size of* $dom(E)$*, and* $|K|$ *is the length of the continuation stack* K*.*

We note that the space for the synchronization counters is included in $S_q(Q_i)$ and not $S_\sigma(Q_i, \sigma_i)$ even though the counters are kept in the store (the rule $L_K(\langle \mathsf{end}\ l\rangle :: K)$ in Figure 9 does not add l to the labels). The motivation for this is to allow an exact correspondence between the locations in the array language semantics and the locations in the P-CEK(1) machine.

Theorem 2 *If* $\cdot,\cdot,\{\} \vdash e \stackrel{a}{\Longrightarrow} v, \sigma; g, s$*, and* e *is a translation of a* Core-NESL *expression, then during the* i *steps of the* P-CEK(1) *execution*

$$[(e, \cdot, \mathsf{nil})], \cdot \quad \stackrel{P(1)^*}{\Longrightarrow} \quad [(\mathsf{exit}\ v, \cdot, \mathsf{nil})], \sigma'$$

the following bound holds on the reachable space for some constant k*:*

$$\max_{i=1}^{n} S_r(Q_i, \sigma_i) \quad \leq \quad ks$$

To prove this we first generalize the statement. The following lemma considers the steps of P-CEK(1) required to evaluate an expression in some general context and bounds the reachable space during those steps by the space specified by the Core-NESL semantics plus the queue space at the beginning of the evaluation.

Lemma 1 *If* $E, \sigma, R \vdash e \stackrel{a}{\Longrightarrow} v, \sigma'; g, s$*, where* e *is a subexpression of a translated* Core-NESL *expression, and for a step* i *of the* P-CEK(1) *machine*

1. $Q_i = [(e, E, K), \ldots]$,
2. *the semantics and machine can access the same locations:* $L(Q_i) = R \cup E(FV(e))$*, and*
3. *... and these locations have the same values:* $\forall_{l \in locs(L(Q_i),\sigma_i)} \sigma(l) = \sigma_i(l)$,

then on some future step $j \geq i$*, the machine will execute* $appK\ v\ K$*, and for some constant* k*,*

$$\max_{n=i}^{j} S_r(Q_n, \sigma_n) \quad \leq \quad S_q(Q_i) + ks$$

Proof: We prove this by structural induction on the array language evaluation derivation and show a representative set of the cases. The remaining cases are similar.

case VAR, $e = x$**:** By VAR, $s = space(R \cup \{E(x)\}, \sigma)$, and by the definition of the P-CEK(q) machine, $j = i$. Thus,

$$\begin{aligned}
& \max\nolimits_{n=i}^{j} S_r(Q_n, \sigma_n) & \\
= \ & S_q(Q_i) + space(L(Q_i), \sigma_i) & \text{(Def. 1)} \\
= \ & S_q(Q_i) + space(L(Q_i), \sigma) & \text{(3rd assump.)} \\
= \ & S_q(Q_i) + s & \text{(2nd assump.)}
\end{aligned}$$

The other base cases, CONST, ABSTR, and LETREC, are similar.

case APP, $e = e_1\ e_2$**:** By induction, we can assume that the lemma holds for e_1, e_2, and the appropriate function body e_3. The steps of the P-CEK(1) machine corresponding to these subderivations are numbered i_1 to j_1, etc. By definition of the P-CEK(q) machine, this implies that $i_1 = i+1$, $i_2 = j_1+1$, and $i_3 = j_2+2$, and step $j_2 + 1$ is the appropriate func-call transition.

Let's look at the queues at these important iterations:

$$\begin{aligned}
Q_i &= [(e_1\ e_2, E, K)] +\!\!+ Q \\
Q_{i_1} &= [(e_1, E, \langle \mathsf{arg}\ e_2\ restr(E, e_2)\rangle :: K)] +\!\!+ Q \\
Q_{i_2} &= [(e_2, E, \langle \mathsf{fun}\ l\rangle :: K)] +\!\!+ Q \\
Q_{j_2+1} &= [(@\ l\ v_2, \cdot, K)] +\!\!+ Q \\
Q_{i_3} &= [(e_3, E'[x' \mapsto l][x \mapsto v_2], K)] +\!\!+ Q
\end{aligned}$$

where l is the value of e_1, $\sigma_{j_1}(l) = cl(E', x, x', e_2)$, and $dom(E')$ is restricted to the free variables of e_2.

Examining the definition of the P-CEK(q) machine and using Definition 1, we have

$$\begin{aligned} S_r(Q_i, \sigma_i) &= S_r(Q_{i_1}, \sigma_{i_1}) \\ S_r(Q_{j_2+1}, \sigma_{j_2+1}) &\leq S_r(Q_i, \sigma_i) \end{aligned}$$

So the reachable space in those two steps is not greater than in the others, and using induction we have

$$\max_{n=i}^{j_3} S_r(Q_n, \sigma_n) \quad \leq \quad \max_{n=1}^{3} \left(S_q(Q_{i_n}) + ks_n\right)$$

Relating the queue spaces $S_q(Q_{i_n})$ to $S_q(Q_i)$ we see

$$S_q(Q_{i_1}) = S_q(Q_{i_2}) = S_q(Q_i) + 1$$

For $S_q(Q_{i_1})$, first observe that $|E'| + 2 \leq s_1$ by the definition of $space(\cdot, \cdot)$ since the closure with E' must have been the result of a subderivation of e_1. Thus,

$$S_q(Q_{i_3}) + ks_3 \quad \leq \quad S_q(Q_i) + ks$$

and the conclusion holds.

The PAIR, IF-TRUE, IF-FALSE, and APPC cases follow in similar, but generally simpler, inductive manners.

The FORK case is also similar except that a function is applied to many values instead of just one. But since the semantics threads the stores through these applications, induction can be used as in the APP case.

□

For the proof of Theorem 2 we specialize the above lemma starting with an empty environment, store, and roots.

Parallel Space. Given the space taken by the P-CEK(1) machine we are now concerned with the space taken by the P-CEK(q) machine for a general q. The P-CEK(q) machine can require more space both because it can create many more simultaneous parallel threads (the queue can become much larger), and because it can have simultaneous access to many more locations in the store. Our aim is to place bounds on how much extra space is needed.

As mentioned, the idea behind the proof is to show that the P-CEK(q) executes a p-DFT traversal of the DAG g returned by the semantics, then use previous results on the number of nodes scheduled prematurely in a p-DFT [2], and finally use these results to bound the space. By the machine traversing the DAG we mean there is a one-to-one correspondence between substate transitions and nodes in the DAG. This implies that each parallel step of the P-CEK(q) processes $\min(q, |Q|)$ nodes of the DAG, and the total number of substates processed is equal to the size of the DAG (*i.e.*, the work). The appendix gives a more formal definition of a traversal and a p-DFT. The following theorem shows the correspondence.

Lemma 2 *If* $\cdot, \cdot, \{\} \vdash e \stackrel{a}{\Longrightarrow} v, \sigma; g, s$, *and* e *is a translation of a* Core-NESL *expression, then there is a one-to-one correspondence between the nodes in* g *and the CEK substates processed in the* P-CEK(q) *transitions*

$$[(e, \cdot, \mathsf{nil})], \cdot \quad \stackrel{P(q)}{\Longrightarrow}^* \quad [(\mathsf{exit}\ v, \cdot, \mathsf{nil})], \sigma'$$

such that the single CEK substate in $|Q_0|$ *corresponds to the root of* g *and for every step* i *of the* P-CEK(q) *all the ready children of the* $\min(q, |Q_i|)$ *CEK substates processed on that step will appear in order at the front of* Q_{i+1}.

Proof Outline: The proof of this is similar to the proof of Theorem 2. In particular we generalize the statement to consider an arbitrary context and then prove by induction on the rules of the semantics. □

This theorem together with Theorem 6 in the appendix and the fact that g is series-parallel imply that the P-CEK(q) executes a p-DFT of g with parameter q. Theorem 5 then bounds the number of premature nodes on any given step of the P-CEK(q). A premature node is a node that gets executed out of order (prematurely) relative to a sequential traversal (*i.e.*, the traversal executed by the P-CEK(1) machine). Having a bound on the number of premature nodes, we can bound the memory used by these nodes.

Theorem 3 *If* $\cdot, \cdot, \{\} \vdash e \stackrel{a}{\Longrightarrow} v, \sigma; g, s$, *where* e *is a subexpression of a translated* Core-NESL *expression, and the number of levels in* g *is* d, *then for the* i *steps of the* P-CEK(q) *execution*

$$[(e, \cdot, \mathsf{nil})], \cdot \quad \stackrel{P(q)}{\Longrightarrow}^* \quad [(\mathsf{exit}\ v, \cdot, \mathsf{nil})], \sigma'$$

the following bound on the reachable space holds for some constant k*:*

$$\max_{n=1}^{i} S_r(Q_n, \sigma_n) \quad \leq \quad k(s + dq)$$

Proof: Since the P-CEK(q) machine executes a p-DFT of g with parameter q, on any step of the P-CEK(q) there can be at most dq nodes executed prematurely relative to the P-CEK(1) machine (see Theorem 5 in the appendix). If each substate transition in a step i of a P-CEK(q) machine added at most constant space to the next state of the machine, then the proof would be easy. In particular since the maximum space taken by any step of the P-CEK(1) machine is ks, and on any step of the P-CEK(q) machine there are at most dq substate transitions that were executed prematurely relative to some step of the P-CEK(1) machine, each of which allocated at most constant space, the total space will be $k(s + dq)$. The reason for using the compact representation of forked threads discussed earlier is to guarantee that the fork transition in Figure 8 only creates constant space.

The one transition that creates more than constant space is **@ new** i since it allocates an array of size i. However, **new** is only used in the translation from Core-NESL in the rule

$$\textbf{allocate}\ n \quad = \quad \textbf{let}\ \_ = \textbf{fork}\ (n, \lambda i.0)\ \textbf{in new}\ n$$

For the **new** transition to be premature, all the n forked threads would also need to be premature. We can then account for the n space required by the allocation of the array against these n forked threads. Our bounds therefore still hold even though this one transition creates more than constant space. □

Parallel Time. Our final goal is to prove bounds on the time taken by each step of the P-CEK(q) machine on the butterfly, hypercube, and PRAM machine models, each with p processors. To account for memory latency in the butterfly and hypercube, and for the latency in the fetch-and-add operation for all three machines, we process $p \log p$ states on each step instead of just p (*i.e.*, we use a P-CEK($p \log p$) machine).

Our simulation uses the *fetch-and-add* operation [16] (or multiprefix [26]). In this operation, each processor has an address and an integer value i. In parallel all processors can atomically fetch the value from the address while incrementing the value by i. In our case it is important that the fetch and add is stable—if two processors make a request simultaneously, the processor with the smaller ID will access the counter first. The stable fetch-and-add operation can be implemented in a butterfly or hypercube network by combining requests as they go through the network [26], and on a PRAM by various other techniques [24, 15]. For all machines, if each processor makes up to m fetch-and-add requests, all requests can be processed in $O(m + \log p)$ time with high probability (the bounds can be slightly improved on the CRCW PRAM [15]). These bounds assume the butterfly has $p \log_2 p$ switches which can do the combining, the hypercube can communicate and combine over all wires simultaneously (multiport version), and that the CRCW PRAM is the priority write version. The fetch-and-add operation is used in four places in our implementation: decrementing the synchronization variables, the **scanadd** operation, allocating tasks to processors, and memory allocation for creating new arrays and getting new labels.

The following theorem assumes that program expressions are of constant size and therefore does not include the time for looking up variables in the environment. It would be easy to generalize to account for variable lookup [3].

Theorem 4 *A step of a* P-CEK($p \log p$) *machine can be simulated in* $O(\log p)$ *time on the butterfly, hypercube, or CRCW PRAM machine models.*

Proof: Each processor takes $\log p$ elements from the queue and executes the transition $\stackrel{P}{\Longrightarrow}_1$. Since we store the queue in a compacted form (*i.e.*, a set of forked threads are stored in constant space) we use a fetch-and-add operation to assign tasks to processors. We make sure that they are assigned to processors in order (lower numbered processors get lower numbered states). Each of the substate transitions can be executed with a constant number of memory references and local operations. This assumes that environment lookup takes constant time, as stated above, and that when forking a set of threads the forked threads are represented in compacted form (otherwise forking n threads would take n time). We also note that any memory allocation that is required can be executed with a fetch-and-add to a global queue. Given these conditions, each processor makes a total of $k \log p$ memory and fetch-and-add requests, taking $O(\log p)$ time using the above stated bounds.

The second substate transition $\stackrel{P}{\Longrightarrow}_2$ is more involved since we have to update the synchronization counters and merge the stores as if they were done sequentially. To update the synchronization counters we use the fetch-and-add operation. Since each processor can have at most $\log p$ requests, this takes $O(\log p)$ time. The fetch-and-add can also be used for the transition on $\Sigma(\vec{i}, \vec{l'}, l'', i, K)$. For merging the stores the only operation that could conflict is a **store** instruction as part of implementing the `<-` operation. However since the substates have the same order as the processors, a priority concurrent write (with higher numbered processors given the higher priority) will guarantee that rightmost value will be written.

To finish the step of the P-CEK(q) we need to merge the states and put them back on the front of the queue. This can be implemented with a fetch-and-add. □

To determine the total running time we use the result that a P-DFT with parameter q on a DAG with w nodes and d levels takes $O(w/q + d)$ steps [2].

Corollary 1 *If* $\cdot, \cdot, \{\} \vdash e \stackrel{a}{\Longrightarrow} v, \sigma; g, s$, *and* e *is a translation of a* Core-NESL *expression, then the* P-CEK($p \log p$) *steps*

$$[(e, \cdot, \mathsf{nil})], \cdot \quad \stackrel{P(p \log p)^*}{\Longrightarrow} \quad [(\mathsf{exit}\ v, \cdot, \mathsf{nil})], \sigma'$$

can be simulated using p *processors of a CRCW PRAM in* $O(w/p + d \log p)$ *time, where* w *and* d *are the numbers of nodes and levels of* g, *respectively.*

Proof: Lemma 2 relates the DAG g to the P-CEK(q) computation, where $q = p \log p$. There are $w/q + d$ steps, and each step takes $O(\log p)$ time. □

7 Related Work and Discussion

Several researchers have used cost-augmented semantics for automatic time analysis or definitional purposes [29, 30, 31, 27, 33, 14]. Hudak and Anderson [21] used partially ordered multisets (pomsets) to model the dependences in various implementations of the λ-calculus. Because of the relationship between partially ordered sets and DAGs, these are quite similar in concept to our DAGS. For our bounds, however, we also need to keep an order among the children of each node, which cannot be represented within the single pomset. None of the above work includes costs that model space or relates the costs of the modeled language to those in machine models. There have been a handful of studies that use semantics to model the reachable space of sequential computations, in the context of both garbage collection (*e.g.*, [25]) and copy avoidance (*e.g.*, [20]). None of this work, however, has considered the extra reachable space required by a parallel evaluation. There have been a sequence of studies that place space bounds on implementations of parallel languages [11, 10, 12, 2]. For a shared memory model, which is required to efficiently simulate the λ-calculus because of shared pointers, the best results are those by Blelloch, Gibbons, and Matias [2], which are the results we use in this paper.

Provable time bounds for mapping nested data-parallel languages onto the PRAM were considered by Blelloch [5] and in the definition of NESL [6], but the time bounds are restricted to a class of program that are called *contained*. Similar results were shown by Suciu and Tannen for a parallel language based on while loops and map recursion [32].

Practical issues. The design of the intermediate language and machine were optimized to simplify the proofs rather than for practical considerations. Here we briefly discuss some modifications to make the implementation more practical without affecting the asymptotic bounds (although they would complicate the analysis). The first modification is to cluster the computations into larger blocks, and then have the substate transitions of the P-CEK(q) machine each execute one of these blocks. This would have three advantages: (1) it would allow the use of standard sequential compiler to compile and optimize each block; (2) it would reduce the scheduling overhead and need for synchronization; and (3) it would improve cache performance since a given block is likely to have repeated accesses to the same memory location. A second modification to the P-CEK(q) is to schedule the sequence of state transitions that correspond to a given thread on the same processor (assuming it gets scheduled on consecutive transitions). This would further improve cache performance.

In terms of the practicality of our target machine models, some readers might complain that they do not properly account for communication costs. First we note that our model does account for communication latency. We already use multithreading for hiding the $\log p$ latency in the butterfly and hypercube networks. The effect of having a larger latency L would simply require a higher degree of multithreading and would appear in our time bounds as $O(w/p + Ld)$, and similarly would replace the $\log p$ in the space bounds. Of course this requires a machine that can properly hide latency. In terms of throughput we note that on the most recent parallel machines, such as the T3E, with proper latency hiding, global bandwidth is no more of an issue than local memory bandwidth on a sequential machine. In both cases performance relies heavily on effective use of the cache. Finally, our implementation uses a fetch-and-add operation. Although this is not very practical on many machines, it is likely that the need could be removed using the techniques discussed in [2].

References

[1] G. Blelloch, G. L. Miller, and D. Talmor. Developing a practical projection-based parallel Delaunay algorithm. In *Proceedings ACM Symposium on Computational Geometry*, May 1996.

[2] Guy Blelloch, Phil Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *ACM Symposium on Parallel Algorithms and Architectures*, July 1995.

[3] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 226–237, June 1995.

[4] Guy Blelloch and Girija Narlikar. A comparison of two n-body algorithms. In *Dimacs implementation challenge workshop*, October 1994.

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

[6] Guy E. Blelloch. NESL: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, 1995.

[7] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, March 1996.

[8] 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.

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

[10] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 362–371, May 1993.

[11] F. Warren Burton. Storage management in virtual tree machines. *IEEE Trans. on Computers*, 37(3):321–328, 1988.

[12] F. Warren Burton and David J. Simpson. Space efficient execution of deterministic parallel programs. Manuscript, December 1994.

[13] Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Proceedings 13th ACM Symposium on Principles of Programming Languages*, pages 314–325, January 1987.

[14] Cormac Flanagan and Mattias Felleisen. The semantics of future and its use in program optimization. In *Proceedings 22nd ACM Symposium on Principles of Programming Languages*, pages 209–220, 1995.

[15] Joseph Gil and Yossi Matias. Fast and efficient simulations among CRCW PRAMs. *Journal of Parallel and Distributed Computing*, 23(2):135–148, November 1994.

[16] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2), April 1983.

[17] John Greiner. A comparison of parallel algorithms for connected components. In *Proceedings 6th ACM Symposium on Parallel Algorithms and Architectures*, pages 16–25, June 1994.

[18] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. In *Proceedings 23rd ACM Symposium on Principles of Programming Languages*, pages 309–321, January 1996.

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

[20] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings ACM Conference on LISP and Functional Programming*, pages 351–363, August 1986.

[21] Paul Hudak and Steve Anderson. Pomset interpretations of parallel functional programs. In *Proceedings 3rd International Conference on Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 234–256. Springer-Verlag, September 1987.

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

[23] 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.

[24] Yossi Matias and Uzi Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, 1991.

[25] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the Symposium on Functional Programming and Computer Architecture*, pages 66–77, June 1995.

[26] Abhiram G. Ranade. *Fluent Parallel Computation.* PhD thesis, Yale University, New Haven, CT, 1989.

[27] Paul Roe. *Parallel Programming using Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, February 1991.

[28] 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.

[29] Mads Rosendahl. Automatic complexity analysis. In *Proceedings 4th International Conference on Functional Programming Languages and Computer Architecture.* Springer-Verlag, September 1989.

[30] David Sands. *Calculi for Time Analysis of Functional Programs.* PhD thesis, University of London, Imperial College, September 1990.

[31] David B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28(1):65–83, July 1995.

[32] Dan Suciu and Val Tannen. Efficient compilation of high-level data parallel algorithms. In *Proceedings 6th ACM Symposium on Parallel Algorithms and Architectures*, pages 57–66, June 1994.

[33] Wolf Zimmermann. Complexity issues in the design of functional languages with explicit parallelism. In *Proceedings International Conference on Computer Languages*, pages 34–43, 1992.

A DAG Definitions and Theorems

The following definitions and theorems are from Blelloch, Gibbons, and Matias [2].

A *p-traversal* of a DAG g, for $p \geq 1$, is a sequence of $k \geq 1$ steps, where each step i, for $i = 1, \ldots, k$, defines a set of nodes, V_i (that are *visited*, or scheduled, at this step), such that the following three properties hold. First, each node appears exactly once in the schedule, *i.e.*, the sets $V_1, \ldots, V_k$ partition the nodes of g. Second, a node is scheduled only after all its ancestors have been, *i.e.*, if $n' \in V_i$ and n is an ancestor of n', then $n \in V_j$ for some $j < i$. Third, each step consists of at most p nodes, *i.e.*, for all $i \in \{1, \ldots, k\}$, $|V_i| \leq p$.

Consider a traversal $T = V_1, \ldots, V_k$ of g. A node $n \in g$ is *scheduled* prior to a step i in T if n appears in $V_1 \cup \cdots \cup V_{i-1}$. An unscheduled node n is *ready* at step i in T if all its ancestors (equivalently, all its parents) are scheduled prior to step i. The *greedy p-traversal*, T_p of a DAG g, *based on* a 1-traversal of g, T_1, is the traversal that on each step i, schedules the p earliest nodes in T_1 that are ready. In other words, for all ready nodes n and n', if n precedes n' in T_1, then either both are scheduled, neither are scheduled, or only n is scheduled.

Let T_p be the greedy p-traversal based on a 1-traversal T_1. For each prefix, σ_p, of T_p, consider the longest prefix, σ_1, of T_1 that includes only nodes in σ_p. We say a node is *premature with respect to* σ_p if it is in σ_p but not in σ_1.

Theorem 5 *For any DAG of depth d and any 1-traversal T, the maximum number of premature nodes in the greedy p-traversal based on T is at most $(p-1)(d-1)$.*

Consider a series-parallel DAG g. Let A be an array initially containing the root node of g. Repeat the following two steps until all nodes in g have been scheduled:

1. Schedule the first $\min(p, |A|)$ nodes from A.

2. Replace each newly scheduled node by its ready children, in left-to-right order, in place in the array A.

Theorem 6 *The algorithm above makes the greedy p-DFT based on the 1-DFT of a series-parallel DAG g.*