Parallelism in Sequential Functional Languages

Guy Blelloch (blelloch@cs.cmu.edu)
John Greiner (jdg@cs.cmu.edu)
School of Computer Science
Carnegie Mellon University

Abstract

This paper formally studies the question of how much parallelism is available in call-by-value functional languages with no parallel extensions (*i.e.*, the functional subsets of ML or Scheme). In particular we are interested in placing bounds on how much parallelism is available for various problems. To do this we introduce a complexity model, the PAL, based on the call-by-value λ-calculus. The model is defined in terms of a profiling semantics and measures complexity in terms of the total *work* and the parallel *depth* of a computation. We describe a simulation of the A-PAL (the PAL extended with arithmetic operations) on various parallel machine models, including the butterfly, hypercube, and PRAM models and prove simulation bounds. In particular the simulations are *work-efficient* (the processor-time product on the machines is within a constant factor of the work on the A-PAL), and for p processors the slowdown (time on the machines divided by depth on the A-PAL) is proportional to at most $O(\log p)$. We also prove bounds for simulating the PRAM on the A-PAL.

Based on the model, we describe and analyze tree-based versions of quicksort and mergesort. We show that for an input of size n these algorithms run on the A-PAL model with $O(n \log n)$ work and $O(\log^2 n)$ depth (expected case for quicksort).

1 Introduction

Many researchers have argued that an important aspect of purely functional languages is their inherent parallelism—since the languages lack side effects, subexpressions may safely be evaluated in parallel. Furthermore, researchers have presented many implementation techniques to take advantage of this parallelism, including data-flow [28], parallel graph reduction [20, 30], and various compiler techniques [14]. Such work has suggested that it might not be necessary to add explicit parallel constructs to functional languages to get adequate parallelism from functional languages.

There has been little study, however, of how much parallelism can be achieved for various problems, or how the inherent parallelism in functional languages relates to more standard models used for analyzing parallel algorithms, such as the PRAM. For example, what are asymptotic bounds for sorting using a parallel implementation of a functional language such as ML or Haskell? What kind of sort would we use? How would the bounds compare with parallel sorting algorithms designed for various machine models? Does it matter whether the language is strict or lazy? Before these can be answered, we first need to augment functional languages with a formal model of complexity. Furthermore, if we want to compare results to previous research on parallel algorithms, we also need to relate this complexity to run time on various machine models. This relation needs to capture some aspects of the parallel implementation of the language. To address these issues this paper makes the following contributions:

1. We introduce a parallel model based on the pure λ-calculus using applicative order (call-by-value) evaluation and specified in terms of a profiling semantics [38, 39]. This semantics defines two measures of complexity. The *work* is the total amount of computation executed by a program. The *computational depth* (or simply depth) is the depth of the computation tree, assuming that the two subexpressions of an application $e_1\ e_2$ are evaluated in parallel. The language is basically equivalent within constant factors of complexity to the functional subsets of eager languages such as ML or Scheme when the parallelism in those languages comes from evaluating arguments in parallel [7]. This correspondence allows us to use the simpler λ-calculus to prove results about the complexity model while using an ML-like language to prove results about algorithms.

2. We prove results on how the complexities in our model relate to complexities of various machine-based models, including the PRAM [15], hypercube, and butterfly models. For the PRAM, we examine both the concurrent read, concurrent write (CRCW) and concurrent read, exclusive write (CREW) variants. The results are summarized in Figure 1. The proofs introduce a parallel version of the SECD machine [25], the P-ECD machine. A state of the P-ECD machine consists of a set of substates, and each state transition of the machine transforms this set into a new set of substates. On each step the substates are scheduled across the processors of the host machine. We also prove results for simulating the PRAM model on our model.

Machine Model	Time
CREW PRAM	$O(w/p + d\log p)$
CRCW PRAM	$O(w/p + d\log\log p)$
CRCW PRAM (randomized)	$O(w/p + d\log^* p)$
Butterfly (randomized)	$O(w/p + d\log p)$
Hypercube (randomized)	$O(w/p + d\log p)$

Figure 1: The mapping of work (w) and depth (d) in the proposed model (the A-PAL) to running time on various machine models. The number of processors on the machine is p. For the randomized algorithms the running times are high-probability bounds (*i.e.*, they will run within the specified time with very high probability). All the results assume that the number of independent variable names in a program is constant, as will be discussed in Section 3. For the butterfly we assume it has $p\lg p$ switches, and for the hypercube, we assume the multiport version (can communicate over all wires simultaneously).

3. We provide examples of analyzing algorithms, specifically parallel versions of quicksort and mergesort. Sequences of size n are stored as balanced trees, since for sequences stored as a list, any algorithm would require $\Omega(n)$ depth just to traverse the list. This accentuates the importance of storing data as trees rather than lists to take advantage of parallel implementations of functional languages. The merging in mergesort borrows ideas from algorithms designed for the PRAM [41], but has some substantial changes to make up for the lack of random access. Both sorting algorithms require $O(n\log n)$ work and $O(\log^2 n)$ depth, and our work bounds are optimal for both merging and sorting, and our depth bounds are optimal for merging.

Applicative-order evaluation is used instead of normal-order evaluation because of ambiguities in defining a formal model based on normal-order evaluation. The problem is that normal-order evaluation can have wide range of implementations, such as call-by-name, call-by-need (lazy), and call-by-speculation (lenient)[1], and these implementations would have very different complexity models. The first two, call-by-name and call-by-need, actually offer no significant parallelism [23]. Call-by-speculation offers plenty of parallelism but does the same amount of work as applicative-order semantics. In particular, a model that uses call-by-speculation would give the same asymptotic work bounds as our model, although it might be possible to improve some depth bounds. Most implementations of lazy languages suggested in the literature sit somewhere between call-by-need and call-by-speculation. Typically some heuristic or strictness analysis is used to decide when to use call-by-speculation instead of call-by-need, and there is some way to garbage collect speculative computations that are never needed. In these implementations a complexity model would depend critically on what heuristics are used or how good the strictness analysis is. An interesting line of future work would be to formally compare implementations using their complexity models.

One inconvenience with our model is the need to keep track of how many variable names are needed. In particular, our simulation bounds need to include the logarithm of the number of independent variables (v_e) in order to account for variable lookup. Fortunately it is straightforward to show that the number of variables for algorithms, such as sorting, is independent of the size of the input, so that v_e does not effect the asymptotic bounds. Another choice would be to restrict the λ-calculus to only allow a constant number of variables. This, however, would require that we chose a particular constant and then show how to convert programs with more variables into this fixed constant number.

The paper is organized as follows. Section 2 describes the model and Sections 3 and 4 relate the model to various machine models. Section 5 gives algorithms for sorting and merging. Section 6 discusses related work.

2 The PAL Model

Our model is based on the untyped λ-calculus using an applicative-order (call-by-value) operational semantics that is augmented with complexity measures. We chose the λ-calculus rather than a specific language since its simplicity makes the simulation results in Section 3 much cleaner, and many features of modern languages (*e.g.*, data-types, conditionals, recursion, and local variables) can be simulated with constant overhead [7], therefore not affecting asymptotic performance. The abstract syntax of the model is

$$e \in \mathit{Expressions} \quad ::= \quad c \mid x \mid \lambda x.e \mid e_1\ e_2$$

where the meta-variable c ranges over a set of constants. We refer to the pure version with no constants as the *parallel applicative λ-calculus* (PAL) model. For the sake of practicality, we also consider a model that includes a set of arithmetic constants (the integers along with some integer operators). We refer to this extended version as the Arithmetic-PAL (A-PAL) model. The A-PAL model can be simulated on the PAL with costs polylogarithmic in the integer range.

In the applicative-order λ-calculus the function and argument can always be evaluated in parallel, and this is the only form of parallelism we consider in this paper. To account for this parallelism our model tracks two complexity measures, the total work executed by a computation and the parallel depth of the computation. When evaluating an expression $e_1\ e_2$ the work of the computation is the sum of the work required to evaluate e_1 and e_2 plus the work needed to apply the result of e_1 to the result of e_2. The depth of the computation is the maximum of the depths of evaluating e_1 and e_2, plus the depth of applying the result of e_1 to the result of e_2. We keep track of the work in addition to the depth for the purpose of proving useful simulation bounds on parallel machines that have a fixed number of processors.

We formalize the work and depth complexities in terms of a *profiling semantics* [38, 39], which extends the standard operational semantics with cost measures. The judgment $E \vdash e \xrightarrow{\lambda} v; w, d$ reads as "In the environment E, the expression e evaluates to value v in work w and depth d." This relation is defined by the rules in Figure 2.

When evaluating a program, we start with an empty environment $[]$. The extension of an environment with a variable and associated value is denoted by $E[x \mapsto v]$, where x may be in E. If E has a binding for x, the associated value is denoted by $E(x)$.

The APP and APPC rules show how work is combined with addition and depth with maximum. The uses of the

[1] We use the term to mean a fully speculative implementation [19].

$$E \vdash c \xrightarrow{\lambda} c; 1, 1 \qquad \text{(CONST)}$$

$$E \vdash \lambda x.e \xrightarrow{\lambda} cl(E, x, e); 1, 1 \qquad \text{(LAM)}$$

$$\frac{E(x) = v}{E \vdash x \xrightarrow{\lambda} v; 1, 1} \qquad \text{(VAR)}$$

$$\frac{E \vdash e_1 \xrightarrow{\lambda} cl(E', x, e'); w_1, d_1 \qquad E \vdash e_2 \xrightarrow{\lambda} v_2; w_2, d_2 \qquad E'[x \mapsto v_2] \vdash e' \xrightarrow{\lambda} v; w_3, d_3}{E \vdash e_1\ e_2 \xrightarrow{\lambda} v; w_1 + w_2 + w_3 + 2, \max(d_1, d_2) + d_3 + 2} \qquad \text{(APP)}$$

$$\frac{E \vdash e_1 \xrightarrow{\lambda} c; w_1, d_1 \qquad E \vdash e_2 \xrightarrow{\lambda} v_2; w_2, d_2 \qquad \delta(c, v_2) = v}{E \vdash e_1\ e_2 \xrightarrow{\lambda} v; w_1 + w_2 + 2, \max(d_1, d_2) + 2} \qquad \text{(APPC)}$$

Figure 2: The profiling semantics of the PAL model.

constant 2 in the these rules is to make an exact correspondence between work and depth and states processed in our simulations (see Section 3). Otherwise the constants do not matter since we are interested in asymptotic analysis. Program constants, λ-expressions, and variables are assumed to evaluate with constant work and depth. As usual, program constants evaluate to themselves, λ-expressions evaluate to closures, and the value of variables is determined by the current environment. Applying a constant function is also assumed to evaluate with constant work and depth. This is a reasonable assumption for most constant functions, including those used here. It is straightforward, however, to augment the model with constant functions whose work and depth is a function of their argument [7].

Definition 1 *The* PAL model *is the λ-calculus with no constants and with the semantics defined by* $E \vdash e \xrightarrow{\lambda} v; w, d$.

Adding Constants to the PAL Model

We now extend the basic PAL model with arithmetic constants to obtain the Arithmetic-PAL model. These constants can be simulated in the base model, but this would require polylogarithmic overheads in both work and depth. The constants are

$$c \in \mathit{Constants} \ ::= \ \ldots \mid i \mid \mathbf{add} \mid \mathbf{add}_i \mid \mathbf{mul} \mid \mathbf{mul}_i \mid \mathbf{neg} \mid \mathbf{div2} \mid \mathbf{pos?}$$

where i ranges over the integers. The primitive functions are addition, multiplication, negation, division by two, and the test for positive integers. For syntactic simplicity, all primitive functions are curried. The choice of primitives is not important, but for the purpose of lower bounds proofs they should be incompressible [2], which ensures that certain kinds of data encoding schemes cannot asymptotically improve complexity bounds, *e.g.*, encoding arrays as integers. This is why general division has been omitted.

The δ functions for these constants are given in Figure 3. The two closures in the δ-rule for **pos?** are standard encodings for the booleans and can be used to encode conditionals [7]. Applying each of these constants requires constant work.

$$\begin{array}{rclrcl}
\delta(\mathbf{add}, i) & = & \mathbf{add}_i & \delta(\mathbf{mul}, i) & = & \mathbf{mul}_i \\
\delta(\mathbf{add}_i, i') & = & i + i' & \delta(\mathbf{mul}_i, i') & = & i \times i' \\
\delta(\mathbf{neg}, i) & = & -i & \delta(\mathbf{div2}, i) & = & \lfloor i/2 \rfloor
\end{array}$$

$$\delta(\mathbf{pos?}, i) = \mathit{if}\ i > 0\ \mathit{then}\ cl([], x, \lambda y.x)\ \mathit{else}\ cl([], x, \lambda y.y)$$

Figure 3: The δ functions for the A-PAL model.

Definition 2 *The* A-PAL model *is the λ-calculus with the constants i,* **add***,* $\mathbf{add}_i$*,* **mul***,* $\mathbf{mul}_i$*,* **neg***,* **div2***, and* **pos?** *and with the semantics defined by* $E \vdash e \xrightarrow{\lambda} v; w, d$.

3 Simulating the A-PAL on Various Machines

In this section we give simulation bounds for simulating the A-PAL model (or PAL) on various machine models.

We first describe the simulation on a serial RAM and then extend this for the simulation on a PRAM, butterfly network, and hypercube. To simulate the A-PAL on the RAM, we use a variant of the SECD machine [25, 31] as an intermediate step. We first show how the work complexity of an A-PAL program is related to the number of state transitions of the SECD machine and then show that each transition can be implemented within given bounds. For the parallel simulations of the A-PAL, we introduce a parallel variant of the SECD machine, the Parallel ECD (P-ECD) machine. The basic idea of the P-ECD machine is that it keeps a set of substates that can be evaluated in parallel. A state transition causes each substate to convert into either 0, 1, or 2 new substates, so the number of substates will vary over the computation. We show that the work complexity of a program is equal to the total number of substates processed and that the depth complexity is exactly equal to the number of steps taken by the P-ECD machine. We then show using an appropriate scheduling how this can be mapped onto various machines with a fixed number of processors.

The SECD machine is a state machine with transition function $\stackrel{\mathrm{S}}{\Longrightarrow}$, where a state (S, E, C, D) consists of a data stack S of values, an environment E, a control list C of expressions or the symbol @ (*apply*), and a "dump" D which is a list of (S, E, C) triples used as a control stack to return from function calls. To evaluate an expression e, the machine starts in the state $(nil, nil, [e], nil)$. It halts when S is a singleton and both C and D are *nil*, with the result being the singleton value in S.

Using the SECD machine, the mapping between work in the A-PAL model and time on a RAM can be split into two simpler parts: the mapping of work in the A-PAL to the number of states in a SECD machine transition sequence, and the mapping of this to time on a RAM.

Lemma 1 *If* $[] \vdash e \stackrel{\lambda}{\longrightarrow} v; w, d$, *then the SECD machine evaluates* e *to* v *in a transition sequence of* w *states.*

Proof outline: Generalizing the lemma to all environments, we can show by structural induction on the A-PAL evaluation derivation that if $E \vdash e \stackrel{\lambda}{\longrightarrow} v; w, d$, then the transition sequence $(S, E, e :: C, D) \stackrel{\mathrm{S}^*}{\Longrightarrow} (v :: S, E, C, D)$ involves w states. The full proof can be found in [7]. □

In our variant of the SECD machine, environments are represented as balanced trees, such as AVL trees. Extending an environment creates a new environment sharing as much structure with the old environment as possible. In particular, extending an environment with a variable name already in the environment creates a new environment with only that binding changed, so that the new environment is no larger than the old. As a result, no environment created during the evaluation of expression e, contains more than the number of variables in e. In the worst case this is equal to the number of λ-expressions since each could have its own variable name, but we assume without loss of generality that names are shared among λ's where it does not cause a conflict. In practice v_e, the logarithm of the number of variables in e, is a small constant that is independent of the data size—it is easy to share names in all common data representations.

Lemma 2 *The SECD transition on state* (S, E, C, D) *can be simulated on a RAM in no more than* $k \lg |E|$ *time, for some constant* k, *where* $|E|$ *is the number of variables in* E.

Proof outline: All transitions except for environment lookup and environment extension can be implemented with simple list manipulations or primitive arithmetic operations and take constant time (this assumes the RAM supports the same arithmetic operations as the A-PAL). The balanced trees representing environments allow lookup and extension in logarithmic time. □

Corollary 1 *Each SECD transition in the evaluation of* e *from the empty environment can be simulated on a RAM in no more than* kv_e *time, for some constant* k.

Proof outline: Follows from v_e bounding the depth of each environment in the evaluation. □

We note that Lemma 2 holds for a pointer machine [24, 44, 2] as well as a RAM since the simulation does not require random access to memory.

Theorem 1 *If* $[] \vdash e \stackrel{\lambda}{\longrightarrow} v; w, d$, *then a RAM can calculate* v *from* e *in no more than* $kv_e w$ *time, for some constant* k.

Proof: Follows from Lemma 1 and Corollary 1. □

Simulating the A-PAL on the P-ECD

For the parallel simulation we introduce the P-ECD machine. Again the simulation can be split into relating the complexity of the A-PAL to the number of state transitions of the P-ECD, and then we can bound the time to execute each transition on various parallel machines.

Each step of the P-ECD machine transforms the current state (Q, M) into a new state. The array Q of substates describes the subexpressions being evaluated, and the array M describes the partial results obtained so far, taking the place of the data stack in the SECD machine. Each element of M contains zero (*noval*) or one ($val(v)$) partial result.

To execute a step we process all the substates $\langle E, C, D\rangle$ in parallel. Processing of each substate consists of executing three transitions. At the beginning of the step, each substate consists of environment E, a balanced tree as in our SECD machine; control C, a single expression to be evaluated; and dump D, a description of how this computation is to communicate its results. After the three transitions each substate results in zero, one, or two new $\langle E, C, D\rangle$ substates. The combination of all these new substates makes up the new Q, thus the size of Q can vary over time. The P-ECD machine starts with one substate $\langle nil, e, nil\rangle$, where e is the program to be evaluated, and exits when a substate reaches a special $\mathbf{Exit}(v)$ substate, where v is the program result (this can only happen when it is the only substate left). We are also guaranteed that a step results in no new substates if and only if the computation is finishing.

The three transitions of a step, *eval*, *valf*, and *vala*, are defined in Figure 4. The *eval* substep may create an intermediate substate $\mathbf{res}(v, D)$ containing the value of this subcomputation which is then communicated by one of *valf* and *vala*. These two substeps coordinate the intermediate results obtained from evaluating functions and arguments, so the processors must synchronize between these latter substeps. Array M can be side-effected by the substeps: *eval* can extend the array, and *valf* and *vala* can update its contents.

We now argue informally why the machine works. The interesting transitions are *eval* on applications and the non-identity *valf* and *vala* transitions. This *eval* transition creates two new substates, one each to evaluate the function and argument. The index i added to the dump D is guaranteed to be independent for each substate processed (*e.g.*, the processor ID plus the number of substates processed in previous steps) and is used as an index into M. Whichever calculation completes first writes its result into M_i and returns no substates. Whenever the second calculation completes, it reads the result from M_i and initiates the application of v_1 to v_2. In the case that the two branches complete on the

$$
\begin{array}{llcll}
E,\ c, & D & \overset{eval}{\Longrightarrow} & \mathbf{res}(c, D) & \text{constant} \\
E,\ \lambda x.e, & D & \overset{eval}{\Longrightarrow} & \mathbf{res}(\mathit{cl}(E, x, e), D) & \text{lambda} \\
E,\ x, & D & \overset{eval}{\Longrightarrow} & \mathbf{res}(E(x), D) & \text{variable} \\
E,\ e_1\ e_2, & D & \overset{eval}{\Longrightarrow} & M_i := \mathit{noval}; & \text{apply} \\
 & & & \mathbf{2S}(\langle E, e_1, \mathbf{fn}(E, i) :: D\rangle, \langle E, e_2, \mathbf{arg}(E, i) :: D\rangle) & \\
 & & & \text{where } i \text{ is new} & \\
E,\ @(\mathit{cl}(E, x, e), v), & D & \overset{eval}{\Longrightarrow} & \mathbf{1S}(\langle E[x \mapsto v], e, D\rangle) & \text{func-call} \\
E,\ @(c, v), & D & \overset{eval}{\Longrightarrow} & \mathbf{res}(\delta(c, v), D) & \text{prim-call} \\
 & & & & \\
\mathbf{res}(v, \mathit{nil}) & & \overset{valf}{\Longrightarrow} & \mathbf{Exit}(v) & \text{exit} \\
\mathbf{res}(v, \mathbf{fn}(E, i) :: D) & & \overset{valf}{\Longrightarrow} & \mathit{case}\ M_i\ \mathit{of} & \text{left-return} \\
 & & & \quad \mathit{val}(v') \Rightarrow \mathbf{1S}(\langle E, @(v, v'), D\rangle) & \\
 & & & \quad \mathit{noval} \Rightarrow M_i := \mathit{val}(v); \mathbf{0S} & \\
\mathbf{res}(v, \mathbf{arg}(E, i) :: D) & & \overset{vala}{\Longrightarrow} & \mathit{case}\ M_i\ \mathit{of} & \text{right-return} \\
 & & & \quad \mathit{val}(v') \Rightarrow \mathbf{1S}(\langle E, @(v', v), D\rangle) & \\
 & & & \quad \mathit{noval} \Rightarrow M_i := \mathit{val}(v); \mathbf{0S} & \\
\end{array}
$$

Otherwise, *valf* and *vala* are identities.

$$Q_j \overset{eval}{\Longrightarrow} Q_j' \overset{valf}{\Longrightarrow} Q_j'' \overset{vala}{\Longrightarrow} Q_j''', \ \textit{for each } j \in \{1, \ldots, q\}$$

Figure 4: Transitions on the substates of the P-ECD. On each step, each substate leads to zero, one, or two new substates (**0S**, **1S**, or **2S**) for the following step. Semicolons are used to sequence a group of statements.

Step	expressions in Q	$\lvert Q\rvert$
1	**add** (**add** 1 2) (**add** 3 4)	1
2	**add** (**add** 1 2), **add** 3 4	2
3	**add**, **add** 1 2, **add** 3, 4	4
4	**add** 1, 2, **add**, 3	4
5	**add**, 1, @(**add**,3)	3
6	@(**add**,1), @($\mathbf{add}_3$,4)	2
7	@($\mathbf{add}_1$,2)	1
8	@(**add**,3)	1
9	@($\mathbf{add}_3$,7)	1
	Work:	19

Figure 5: P-ECD example evaluation using the expression **add** (**add** 1 2) (**add** 3 4). The total work is the sum over all steps of the lengths of Q.

same step, we guarantee that they both do not believe that the other is still running by synchronizing between the *valf* and *vala* phases. (With an atomic test-and-set, synchronizing could be avoided.)

As an example of the execution of the P-ECD, Figure 5 shows Q at the beginning of each step of evaluating the expression (**add** (**add** 1 2) (**add** 3 4)).

Lemma 3 *For all expressions e, if there exists a value v such that $[] \vdash e \xrightarrow{\lambda} v; w, d$, then v is calculated from e using d steps of a P-ECD machine. Furthermore, the P-ECD calculation processes a total of w substates.*

Proof: We prove that the number of steps taken by the P-ECD machine is d by induction on the structure of the A-PAL evaluation derivation. The induction hypothesis is that if $E \vdash e \xrightarrow{\lambda} v; w, d$ and the P-ECD machine at step s is in a state (Q, M) such that substate $\langle E, e, D\rangle$ is in Q, then an instance of the *eval* substep of step $s + d - 1$ results in $\mathbf{res}(v, D)$.

CONST, LAM, or VAR: The current *eval* substep results in $\mathbf{res}(v, D)$. By the profiling semantics, $d = 1$, so the hypothesis is true.

APP: By *eval*, two substates $\langle E, e_1, D_1\rangle$ and $\langle E, e_2, D_2\rangle$ are created after one step. By the induction hypothesis, e_1 completes after d_1 steps, and e_2 completes after d_2 steps. If the calculation for e_1 completes before the calculation for e_2 (*i.e.*, $d_1 < d_2$), then when e_2 completes, $\langle E, @(v_1, v_2), D\rangle$ is in Q at step $s + d_2 + 1$. Otherwise, when e_1 completes, $\langle E, @(v_1, v_2), D\rangle$ is in Q at step $s + d_1 + 1$. Therefore, $\langle E', @(\mathit{cl}(E, x, e), v_2), D\rangle$ is in Q at step $s + \max(d_1, d_2) + 1$. At the beginning of the next step, $s + \max(d_1, d_2) + 2$, the substate $\langle E[x \mapsto v], e, D\rangle$ is in Q. By the induction hypothesis, an instance of the *eval* substep of step $(s + \max(d_1, d_2) + 2) + d_3 - 1$ results in $\mathbf{res}(v, D)$. Since the profiling semantics shows that $d = \max(d_1, d_2) + d_3 + 2$, this gives the desired results.

APPC: The argument is the similar to the previous rule, except that at the beginning of step $s + \max(d_1, d_2) + 1$ the substate $\langle E, @(c, v_2), D\rangle$ is in Q, and an instance of the *eval* substep results in $\mathbf{res}(v, D)$.

Now we show that the calculation processes w substates, using induction on the A-PAL derivation.

CONST, LAM, or VAR: Exactly one P-ECD substate is processed for each of these A-PAL rules.

APP: By induction, computing e_1, e_2, and e' processes w_1, w_2, and w_3 substates, respectively. In addition, one substate with expression $e_1\ e_2$ and one with expression $@(cl(E, x, e), v)$ are processed, so the total processed is $w = w_1 + w_2 + w_3 + 2$.

APPC: By induction, computing e_1 and e_2 processes w_1 and w_2 substates, respectively. In addition, one substate with expression $e_1\ e_2$ and one with expression $@(c, v)$ are processed, so the total processed is $w = w_1 + w_2 + 2$.

□

Simulating the A-PAL on other Parallel Machines

We now need to show how to simulate the P-ECD machine on a PRAM, butterfly network, and hypercube. For the butterfly we assume that for p processors we have $p \lg p$ switches and p memory banks, and that memory references can be pipelined through the switches. On such a machine each of the p processors can access (read or write) n elements in $O(n + \log p)$ time with high probability [27, 33]. The $O(\log p)$ time is due to latency through the network. We also assume the butterfly network has simple integer adders in the switches, such that a prefix-sum computation can execute in $O(\log p)$ time. A separate prefix tree, such as on the CM-5, would also be adequate. For the hypercube we assume a multiport hypercube in which messages can cross all wires on each time step, and for which there are separate queues for each wire. This model is quite similar to butterfly and has the same bounds for simulating shared memory. However, we do not need to assume that the switches have integer adders. As in the previous models, we assume that primitive function calls can be implemented in constant time on a single processor.

Lemma 4 *Each step of the P-ECD machine with q substates can be processed on a p processor machine within the following time bounds:*

Machine Model	*Time*
CREW PRAM	$kv_e(\lceil q/p \rceil + \log p)$
CRCW PRAM	$kv_e(\lceil q/p \rceil + \log\log p)$
CRCW PRAM (rand.)	$kv_e(\lceil q/p \rceil + \log^* p)$
Butterfly (rand.)	$kv_e(\lceil q/p \rceil + \log p)$
Hypercube (rand.)	$kv_e(\lceil q/p \rceil + \log p)$

for some constant k, where the bounds on randomized machines hold with high probability.

Proof: For the simulation we keep the substates returned by each step in an array. If this substate array is of size q, each processor is responsible for q/p elements of the array (*i.e.*, processor i is responsible for the elements $[iq/p, \ldots, (i+1)q/p - 1]$). We assume each processor knows its own processor number, so it can calculate a pointer to its section of the array. For the CREW and butterfly simulations the size of the array is exactly q. For the CRCW PRAM simulations the array can have holes in it that don't contain substates, as explained below. These holes are marked, and we guarantee that the total length of the array is at most kq for some constant k. This means that each processor is responsible for at most kq/p elements.

The simulation of a step consists of the following substeps:

1. Locally evaluating the substates using the *eval* transition in Figure 4. This requires accessing shared memory for reading but requires no communication among the substates.

2. Evaluating the *valf* and *vala* transitions. This requires a synchronization between the two transitions. Each processor first applies the *valf* transitions for all the substates for which it is responsible. The processors then synchronize, and then each processor applies the *vala* transitions.

3. Creating a new substate array for the next step. After the substep transitions, each array element contains zero, one, or two substates (**0S**, **1S**, or **2S**), and these must be distributed into the new array.

We need to show that each of these steps can be executed in the given bounds. The first step requires the time it takes to process q/p substates. The *eval* transition is similar to the *eval* for the serial SECD machine. The only real difference is the apply transition. Each of the other substate transitions require the v_e time that was required in the serial machine and can have at most v_e memory references. The apply transition can also be executed in these bounds since it just requires an additional memory write. We can generate the independent i's simply by using the array index for the substate added to an offset which gets reset on each round. None of the memory references require concurrent writes. The time for the first substep on the CREW and CRCW PRAM is therefore q/p. The time on the butterfly and hypercube is $q/p + \lg p$ since the memory references require a $\lg p$ latency through the network. The second step can also be executed in the same bounds.

The third step requires generating a new substate array. Each transitioned substate of the old array contains zero, one, or two substates, which need to be distributed into a new array for the next step. For the CREW PRAM and butterfly this can be done by executing a prefix-sum on the number of new substates and using the result as an offset into the new array. In both cases for p processors the prefix sum and writing into the new array can run in $O(q/p + \log p)$ time. This will give a new array that is exactly the length of the number of new substates. On the CRCW PRAM the distribution into the new array can be done more efficiently using a solution to the *linear approximate compaction* problem [26]: given an array of n cells, m of which contain an object, place the m objects in distinct cells of an array of size km for some constant $k > 1$. The idea is to first allocate two new positions for each substate, mark the substates that will remain (neither for **0S**, one for **1S**, and both for **2S**) and then do an approximate compaction. Since the result array is a constant times larger than the total number of remaining substates, we will maintain the invariant mentioned earlier. Gil, Matias, and Vishkin [16] have shown that the linear approximate compaction problem can be solved on a p processor CRCW PRAM (ARBITRARY) in $O(n/p + \log^* p)$

expected time (using a randomized solution). Goldberg and Zwick [17] have recently shown that the problem can be solved deterministically in $O(n/p + \log\log p)$ time.

When we add the times for the three substeps, we get the stated bounds for each of the machines. □

Theorem 2 *If* $[] \vdash e \xrightarrow{\lambda} v; w, d$, *then v can be calculated from e on a CREW PRAM with p processors in $kv_e(w/p + d\log p)$ time, for some constant k. Analogous results are true for the other models.*

Proof: The proof uses Brent's scheduling principle [9]. We prove it for the CREW PRAM, but the other proofs are almost identical. We assume that step i of the P-ECD processes q_i substates. We know from Lemma 3 that $\sum_{i=0}^{i<d} q_i = w$. We also know from Lemma 4 that it takes $k'v_e(\lceil q_i/p \rceil + \log p)$ time to process step i. The total time to process all substates is then

$$\begin{aligned} T &= \sum_{i=0}^{i<d} k'v_e(\lceil q_i/p\rceil + \log p) \\ &< k'v_e \sum_{i=0}^{i<d} (q_i/p + 1 + \log p) \\ &= k'v_e((\sum_{i=0}^{i<d} q_i/p) + d(1 + \log p)) \\ &= k'v_e(w/p + d(1 + \log p)) \\ &\le 2k'v_e(w/p + d\log p) \\ &= kv_e(w/p + d\log p) \end{aligned}$$

where we have set $k = 2k'$. □

4 Simulating a PRAM on an A-PAL

In this section we consider simulating a PRAM on an A-PAL. The simulation we use gives the same results for the EREW, CREW, and CRCW PRAM as well as for the multiprefix [32] and scan models [4]. The simulation is optimal in terms of work for all the PRAM variants. This is because it takes logarithmic work to simulate each random access into memory (this is the same as for pointer machines [2]). Since we don't know how to do better for the weaker models, we will base our results on the most powerful model, the CRCW PRAM with unit-time multiprefix sums (MP PRAM).

Theorem 3 *A program that runs in time t on a p processor MP PRAM using m memory can be simulated on the A-PAL model with $k_w p \log m$ work and $k_d t \log m \log p$ depth, for some constants k_w and k_d.*

Proof: We will simulate a PRAM based on state transitions on the state (C, M, P) where C is the code, M is the memory, and P is state for all the processors (*i.e.*, registers and program counter). Let $c = |C|$, $m = |M|$, and $p = |P|$. We assume C, M, and P are stored as balanced binary trees and that $p \le m$, and $c \le m$. Each state transition corresponds to a step of the PRAM, and the processors will be strictly synchronous. Register-to-register instructions can be implemented with $O(p)$ work and $O(\log p)$ depth, and concurrent reads with $O(p \log m)$ work and $O(\log m)$ depth. This just requires traversing the appropriate trees. The writes are the only interesting instruction to implement, and can be implemented by sorting the write requests from the processors by address and then recursively splitting the requests at each node of M as we insert them. We can sort the p requests in $O(p\log p)$ work and $O(\log^2 p)$ depth as discussed in the next section. We assume the sorted requests, which we call the write-tree, start out balanced and are sorted from left to right in the tree. To implement a concurrent write or multiprefix, we combine nodes in the write-tree that have the same address. Since the addresses are sorted this can be done in $O(p)$ work and $O(\log p)$ depth.

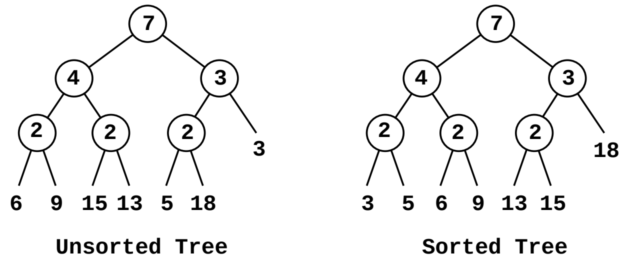
We now consider the insertion of the sorted requests of a write-tree W into memory M (`modify(`M`,`W`)`). We assume that M stores the addresses and associated values at the leaves, ordering the addresses from left-to-right, and that the internal nodes contain the value of the greatest address in the left branch. We assume all addresses in W are also in M, and that each node of W stores the minimum and maximum address of its descendants, so that we can access these in constant work and depth. To insert W into M, we first check if M is a single node, in which case W must also be a single node, and we simply modify the value and return. Otherwise, we check if all the addresses in W belong to just one of the branches of the M tree. If so, we call `modify` recursively on that branch of M with the same W and put the result back together with the other branch of M when the call returns. If not, we split W based on the address stored at the root of M and call `modify` in parallel on the two children of M and the two split parts of W. This algorithm works since all addresses in the original write-tree will eventually find their way to the appropriate leaf of the M tree and modify that leaf.

We now consider the total work and depth required. Splitting W into two trees based on a key can be implemented in $O(\log p)$ work and depth by following down to the appropriate leaf, splitting along the way. Since M is of depth $\lg m$, the total depth complexity is therefore bound by $O(\log p \log m)$. To prove the bounds on the work, we observe that it cannot take more than $O(p\log p)$ work to split the tree into p pieces of size 1 since each split takes $O(\log p)$ work and there are $p-1$ of them. This means the total work needed to split the original write-tree is bound by $O(p \log p)$. The only other work is the check at each node of the M tree of whether we have to split or send all values down to one or the other branches. The maximum work done for these checks is $O(p\log m)$ since there can be at most p separate chains (one per leaf of the write-tree) each which is at most as deep as the M tree ($O(\log m)$). The total work is therefore $O(p(\log p + \log m)) = O(p \log m)$. □

5 Analyzing Algorithms

In this section we examine how the model can be used to analyze algorithms. As examples, we describe parallel versions of quicksort and mergesort. These two algorithms illustrate some of the techniques necessary for programming efficient algorithms in the model.

We first note that any sorting algorithm that represents its input as a list requires depth at least proportional to




```
datatype 'a Tree =
    Empty | Leaf of 'a
  | Node of int * 'a Tree * 'a Tree
```

Figure 6: Representing sequences as trees. The values are stored at the leaves and each internal node stores the size of its subtree (the number of leaves below it).

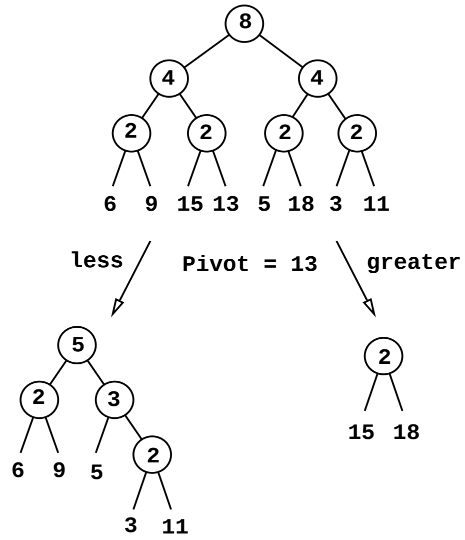
its input size—this is the time required just to look at all the elements. In fact a simple mergesort that makes its two recursive calls in parallel will match this lower bound for depth. To derive parallel algorithms that are sublinear in the input size requires that the input and output are represented as trees. This section shows how trees can be used to derive effective parallel versions of quicksort and mergesort and analyzes these versions in the PAL model. The tree representation we will use is given in Figure 6. We assume that the ordering for sorted sequences is specified by a left-to-right traversal of the tree.

Parallel Quicksort: The code for our quicksort algorithm is given in Figure 7. The function `qsort_rec` returns a sorted tree, but in general it will not be perfectly balanced, so the function `rebalance` rebalances it. The function `qsort_rec` is similar to the sequential version of quicksort on lists, except that `elt`, `select`, and `append` are implemented on trees. The function `elt` can be implemented by traversing the tree down to the appropriate leaf, and for a tree of depth d, this requires $O(d)$ work and depth (there is no parallelism). The function `select` is implemented by calling itself recursively in parallel on both branches and putting the results back together. Assuming the function `f` has constant work and depth, `select` on a tree of size n and depth d requires $O(n)$ work and $O(d)$ depth. We note that the tree returned by `select` is generally not going to be balanced, which is why we do not assume that $d = \lg n$. The `append` function simply puts its two arguments together in a tree node and therefore has constant work and depth.

We first present a general theorem that bounds work and depth for our quicksort in the expected case for any input tree, even if not balanced, and as a corollary give the bounds for balanced input.

Theorem 4 *The quicksort algorithm specified in Figure 7 when applied to a tree with n leaves and depth d will execute in $O(n \log n)$ work and $O(d \log n)$ depth on the A-PAL model, both expected case* (i.e., *average over all possible inputs of that depth and size).*

Proof: We first consider `qsort_rec`. We note that since the pivots in quicksort will not perfectly split the data, some recursive paths will be longer than others. We call the longest path of recursive calls for `qsort_rec` on a particular input




```
fun append Empty b = b
  | append a Empty = a
  | append a b = Tree (size(a)+size(b),a,b)

fun select f Empty = empty
  | select f (Leaf x) =
    if f x then Leaf x else Empty
  | select f (Tree (_,left,right))
    append (select f left) (select f right)

fun qsort_rec x =
    if (size x) < 2 then x
    else
      let val pivot = elt a ((size x)/2)
          val less = select (fn x => x < pivot) a
          val equal = select (fn x => x = pivot) a
          val greater = select (fn x => x > pivot) a
      in
          append (qsort_rec less)
                 (append equal (qsort_rec greater))
      end

fun rebalance Empty = Empty
  | rebalance (Leaf x) = Leaf x
  | rebalance a =
    let val half = (size x)/2
    in
        append (rebalance (take x half)
               (rebalance (drop x half))
    end

fun quicksort a = rebalance (qsort_rec a)
```

Figure 7: An example diagram of the `select` function and the code for the parallel quicksort algorithm.

the *recursion depth* for that input. We note that the worst case recursion depth is $O(n)$ and that fewer than 1 out of n of the possible inputs will lead to a recursion depth greater than $k \log n$ [34]. To determine the total computational depth of qsort_rec, we need to consider the computational depth along the longest path. We claim that this computational depth is at most $O(d)$ times the recursion depth since each node along the recursion tree will require at most $O(d)$ depth. This is because elt and select will run in $O(d)$ depth.[2]. Since a fraction of only $1/n$ of the inputs will have a recursion depth greater than $O(\log n)$, and these cases will have recursion depth at most $O(n)$, the average (expected case) computation depth of qsort_rec is

$$D(n) = O(d(\log n + \frac{1}{n}n)) = O(d \log n).$$

To see that the work is expected to be $O(n \log n)$, we simply note that all steps do no more than a constant fraction more work than a list-based sequential implementation.

We now briefly consider the routine rebalance. We note that the depth of the tree returned by qsort_rec is at most a constant times the recursion depth. The function rebalance is implemented by splitting the tree along the path that separates the tree into two equal size pieces (or off by 1), recursively calls itself on the two parts, and appends the results. We claim that for a tree of size n and depth d it will run with $O(n \log n)$ work and $O(d \log n)$ depth (worst case). Given the above bounds on the recursion depth, this gives an expected depth of $O(\log^2 n)$. □

Corollary 2 *The quicksort algorithm specified in Figure 7 when applied to a balanced tree with n leaves will execute in $O(n \log n)$ work and $O(\log^2 n)$ depth on the A-PAL model, both expected case.*

Parallel Mergesort: We first consider the problem of merging two sorted trees. We use n to refer to sum of the sizes of the two trees. We assume that each internal node of the input trees contains the maximum value of its descendants, as well as its size. This is clearly easy to generate in $O(n)$ work and $O(\log n)$ depth. The main component of the parallel algorithm is a routine select_kth which given two ordered trees a and b, returns the k^{th} smallest value from the combination of the two sequences (see Figure 8). It is implemented using a dual binary search in which we go down a branch from one of the two sequences on each step, using the maximal element at each node for navigation. Assuming the depths of the two trees are d_a and d_b, the work and depth complexity of this routine is $O(d_a + d_b)$.

To merge two trees, we use select_kth to find their combined median element. We then select the elements less and greater, respectively, than the median for each tree with the functions take_less and drop_less. These can be implemented with $O(\log n)$ work and depth since the trees are sorted and balanced (it just requires going down a tree splitting along the way). Recursively merging the two trees of lesser elements and the two trees of greater elements gives us two sorted trees which are guaranteed to be the same size (or off by one) by construction. So, joining them under a new node produces a balanced sorted tree. As a whole, merging in this manner takes $O(n)$ work and $O(\log^2 n)$ depth since we recurse for the $\lg n$ depth of the trees.

[2] Note that although select does not return balanced trees, it will never return a tree with depth greater than the original tree, which has depth d.

```
datatype 'a Tree =
    Empty | Leaf of 'a
  | Node of int * 'a * 'a Tree * 'a Tree

fun select_kth k (Leaf v1) (Leaf v2) =
    if v2 > v1 then if k = 0 then v1 else v0
    else if k = 0 then v0 else v1
  | select_kth k (Leaf v1) (Node (n2,v2,l2,r2)) =
    if v2 > v1 then if k > n2
       then select_kth (k-n2) (Leaf v1) r2
       else select_kth k (Leaf v1) l2
    else if n2 > k
       then select_kth k (Leaf v1) l2
       else select_kth (k-n2) (Leaf v1) r2
  | select_kth k (Node (n1,v1,l1,r1)) (Leaf v2) =
    select_kth k (Leaf v2) (Node (n1,v1,l1,r1)) =
  | select_kth k (Node (n1,v1,l1,r1))
                 (Node (n2,v2,l2,r2)) =
    if v2 > v1 then if k > (n1+n2)
       then select_kth k (Node (n1,v1,l1,r1)) l2
       else select_kth (k-n1) r1 (Node (n2,v2,l2,r2))
    else if k > (n1+n2)
       then select_kth k l1 (Node (n2,v2,l2,r2))
       else select_kth (k-n1) (Node (n1,v1,l1,r1)) r2

fun merge (Leaf x) b = insert x b
  | merge a (Leaf y) = insert y a
  | merge a b =
    let val k = ((size a) + (size b)) / 2
        val median = select_kth k a b
    in
        append (merge (take_less a median)
                      (take_less b median))
               (merge (drop_less a median)
                      (drop_less b median))
    end

fun merge_sort a =
  if (size a) < 2 then a
  else let val half = (size a)/2
       in merge (merge_sort (take a half))
                (merge_sort (drop a half))
```

Figure 8: Code of the parallel mergesort algorithm.

Theorem 5 *The mergesort algorithm specified in Figure 8 when applied to a balanced tree with n leaves will execute in $O(n \log n)$ work and $O(\log^3 n)$ depth on the A-PAL model.*

Proof: We can write the following recurrences for work and depth:

$$\begin{aligned} W(n) &= 2W(n/2) + W_{\text{merge}}(n) \\ &= 2W(n/2) + O(n) \\ &= O(n \log n) \\ D(n) &= D(n/2) + D_{\text{merge}}(n) \\ &= D(n/2) + O(\log^2 n) \\ &= O(\log^3 n) \end{aligned}$$

□

This version of mergesort is not as efficient as the quicksort previously described. However, if merging uses $n/\lg n$ splitters, rather than just the median, the depth complexities of merging and mergesort can each be improved by a factor of $\log n$ [7].

6 Related Work

Several researchers have used cost-augmented semantics for automatic time analysis of serial programs [3, 38, 39, 45]. This work was concerned with serial running time, and since they were primarily interested in automatically analyzing programs rather than defining complexity, they each altered the semantics of functions to simplify such analysis. Furthermore, none related their complexity models to more traditional machine models, although since the languages are serial this should not be hard.

Roe [36, 37] and Zimmermann [46, 47] both studied profiling semantics for parallel languages. Roe formally defined a profiling semantics for an extended λ-calculus with *lenient* evaluation. In his semantics, the two subexpressions of a special *let* expression *plet* $x = e_1$ *in* e_2 evaluate in parallel such that the evaluation of an occurrence of x in e_2 is delayed until its value is available. To define when this is the case, he augmented the standard denotational semantics with the time that each expression begins and ends evaluation. He did not show any complexity bounds resulting from his definition or relate this model to any other. Zimmerman introduced a profiling semantics for a data-parallel language for the purpose of automatically analyzing PRAM algorithms. The language therefore almost directly modeled the PRAM by adding a set of PRAM-like primitive operations. Complexity was measured in terms of time and number of processors, as it is measured for the PRAM. It was not shown, however, whether the model exactly modeled the PRAM. In particular since it is not known until execution how many processors are needed, it is not clear whether the scheduling could be done on the fly.

Hudak and Anderson [19] suggest modeling parallelism in functional languages using an extended operational semantics based on partially ordered multisets (pomsets). The semantics can be though of as keeping a trace of the computation as a partial order specifying what had to be computed before what else. Although significantly more complicated, their call-by-value semantics are related to the A-PAL model in the following way. The work in the A-PAL model is within a constant factor of the number of elements in the pomset, and the steps is within a constant factor of the longest chain in the pomset. They did not relate their model to other models of parallelism or describe how it would effect algorithms.

Previous work on formally relating language-based models (languages with cost-augmented semantics) to machine models is sparse. Jones [21] related the time-augmented semantics of simple while-loop language to that of an equivalent machine language in order to study the effect of constant factors in time complexity. Seidl and Wilhelm [40] provide complexity bounds for an implementation of graph reduction on the PRAM. However, their implementation only considers a single step and requires that you know which graph nodes to execute in parallel in that step and that the graph has constant in-degree. Under these conditions they show how to process n nodes in $O(n/p + p \log p)$ time (which is a factor of p worse than our bounds in the second term, see Lemma 4). There have also been several experimental studies of how much parallelism is available in sequential functional languages [11, 8, 10].

The work-step paradigm has been used for many years for informally describing parallel algorithms [42, 22]. It was first included in a formal model by Blelloch in the VRAM [5]. NESL [6], a data-parallel functional language, includes complexity measures based on work and steps and has been used for describing and teaching parallel algorithms. Skillicorn [43] also introduced cost measures specified in terms of work and steps for a data-parallel language based on the Bird-Meertens formalism. In both cases the languages were not based on the pure λ-calculus but instead included array primitives. Also neither formally showed relationship of their models to machine models. Part of the motivation of the work described in this paper was to formalize the mapping of complexity to machine models and to see how much parallelism is available without adding data-parallel primitives.

Dornic, *et al.* [13] and Reistad and Gifford [35] explore adding time information to a functional language type system. But for type inference to terminate, only special forms of recursion can be used, such as those of the Bird-Meertens formalism.

There has been much work on comparing machine models within traditional complexity theory. The most closely related is that of Ben-Amram and Galil [2], who show that a pointer machine incurs logarithmic overhead to simulate a RAM. The pointer machine [24, 44] is similar to the SECD machine in that it addresses memory only through pointers, but it lacks direct support for implementing higher-order functions. We borrow from them the parameterization of models over incompressible data types and operations. Paige [29] also compares models similar to those used by Ben-Amram and Galil.

Goodrich and Kosaraju [18] introduced a parallel pointer machine (PPM), but this is quite different from our model since it assumes a fixed number of processors and allows side effecting of pointers. Another parallel version of the SECD machine was introduced by Abramsky and Sykes [1], but their Secd-m machine was non-deterministic and based on the fair merge.

7 Conclusions

This paper has discussed a complexity model based on the λ-calculus and shown various simulation results. A goal of this work is to bring a closer tie between parallel algorithms and functional languages. We believe that language-based complexity models, such as the ones suggested in this paper, could be a useful way for describing and thinking about parallel algorithms directly, rather than always needing to translate to a machine model.

This paper leaves several open questions, including

- We mentioned that a call-by-speculation implementation of normal-order evaluation might allow for improved depth bounds for various problems. In particular it allows for pipelined execution. Does this help, and on what problems?

- Is it possible to sort within $d = o(\log^2 n)$, and $w = O(n \log n)$?

- Can the bounds for simulating the A-PAL on a PRAM be improved? The bounds for the butterfly network are tight.
- Our simulations are memory inefficient. Can good bounds be placed on the use of memory?
- Because it lacks random-access, can the A-PAL model be simulated more efficiently than the PRAM on machines that have less powerful communication (*e.g.*, fixed-topology networks, parallel I/O models, or the LOGP model [12]), and can the complexity model be augmented to capture the notion of locality for these machines?

8 Acknowledgments

This research was sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330 and contract number F19628-91-C-0168. It was also supported in part by an NSF Young Investigator Award and by Finmeccanica. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government.

References

[1] Samson Abramsky and R. Sykes. Secd-m: A virtual machine for applicative programming. In Jean-Pierre Jouannaud, editor, *Proceedings 2nd International Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 81–98, 1985.

[2] Amir M. Ben-Amram and Zvi Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617–648, July 1992.

[3] Bror Bjerner and Sören Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Proceedings 4th International Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, September 1989.

[4] Guy Blelloch. *An L1 User's Manual (Version 1.2: Draft)*, November 1989.

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

[6] 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.

[7] Guy E. Blelloch and John Greiner. A parallel complexity model for functional languages. Technical Report CMU-CS-94-196, Carnegie Mellon University, October 1994.

[8] A. P. Willem Bohm and R. E. Hiromoto. The dataflow time and space complexity of FFTs. *Journal of Parallel and Distributed Computing*, 18(3):301–313, July 1993.

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

[10] I. Checkland and C. Runciman. Perfect hash functions made parallel—lazy functional programming on a distributed multiprocessor. In T. N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proceedings 26th Hawaii International Conference on System Sciences*, volume 2, pages 397–406, January 1993.

[11] C. D. Clack and Simon Peyton Jones. Generating parallelism from strictness analysis. Technical Report Internal Note 1679, Dept. Comp. Sci., University College London, February 1985.

[12] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[13] Vincent Dornic, Pierre Jouvelot, and David K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, March 1992.

[14] 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.

[15] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[16] J. Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proceedings Symposium on Foundations of Computer Science*, pages 698–710, October 1991.

[17] T. Goldberg and U. Zwick. Optimal deterministic processor allocation. In *Proceedings 4th ACM-SIAM Symp. on Discrete Algorithms*, January 1995.

[18] Michael T. Goodrich and S. Rao Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. In *30th Annual Symposium on Foundations of Computer Science*, pages 190–195, November 1989.

[19] 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.

[20] Paul Hudak and Eric Mohr. Graphinators and the Duality of SIMD and MIMD. In *ACM Conference on Lisp and Functional Programming*, pages 224–234, July 1988.

[21] Neil D. Jones. Constant time factors *do* matter (extended abstract). In *Proceedings 25th ACM Symposium on Theory of Computing*, pages 602–611, 1993.

[22] 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.

[23] Richard Kennaway. A conflict between call-by-need computation and parallelism (extended abstract). In *Proceedings Conditional Term Rewriting Systems-94*, February 1994.

[24] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, MA, 1968.

[25] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[26] Yossi Matias and Uzi Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *Proceedings ACM Symposium on Theory of Computing*, pages 307–316, May 1991.

[27] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memory. *Acta Informatica*, 21:339–374, 1984.

[28] Rishiyur S. Nikhil. ID Version 90.0 Reference Manual. Computation Structures Group Memo 284-1, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1990.

[29] Robert Paige. Real-time simulation of a set machine on a RAM. In W. Koczkodaj, editor, *Proceedings International Conference on Computing and Information*, volume 2, pages 68–73, 1989.

[30] S. L. Peyton Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, 32(2):175–186, 1989.

[31] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[32] Abhiram G. Ranade. *Fluent Parallel Computation*. PhD thesis, Yale University, Department of Computer Science, New Haven, CT, 1989.

[33] Abhiram G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, June 1991.

[34] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal of Computing*, 14(2):396–409, 1985.

[35] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *Proceedings ACM Conference on LISP and Functional Programming*, pages 65–78, July 1994.

[36] Paul Roe. Calculating lenient programs' performance. In Simon L Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Proceedings Functional Programming, Glasgow 1990*, Workshops in computing. Springer-Verlag, August 1990.

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

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

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

[40] Helmut Seidl and Reinhard Wilhelm. Probabilistic load balancing for parallel graph reduction. In *Proceedings TENCON '89, 4th IEEE Region 10 International Conference*, pages 879–884, November 1989.

[41] Yossi Shiloach and Uzi Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, 1981.

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

[43] David B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. To appear in the Journal of Parallel and Distributed Computing.

[44] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. System Sci.*, 18:110–127, 1979.

[45] Philip Wadler. Strictness analysis aids time analysis. In *Proceedings 15th ACM Symposium on Principles of Programming Languages*, January 1988.

[46] Wolf Zimmermann. Automatic worst case complexity analysis of parallel programs. Technical Report TR-90-066, International Computer Science Institute, December 1990.

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