A Proposal for Parallel Self-Adjusting Computation

Matthew Hammer Umut A. Acar
Toyota Technological Institute,
Chicago, IL
{hammer,umut}@tti-c.org.

Mohan Rajagopalan Anwar Ghuloum
Programming Systems Lab, Intel,
Santa Clara, CA.
{mohan.rajagopalan,anwar.ghuloum}@intel.com.

ABSTRACT

We present an overview of our ongoing work on parallelizing self-adjusting-computation techniques. In self-adjusting computation, programs can respond to changes to their data (e.g., inputs, outcomes of comparisons) automatically by running a change-propagation algorithm. This ability is important in applications where inputs change slowly over time. All previously proposed self-adjusting computation techniques assume a sequential execution model. We describe techniques for writing parallel self-adjusting programs and a change propagation algorithm that can update computations in parallel. We describe a prototype implementation and present preliminary experimental results.

1. INTRODUCTION

Self-adjusting computations [1] describe programs that have the inherent ability to respond to changes in their computational environment. Typical examples of changes include changes to the input data of a program and changes in the intermediate data generated during a run. Existing work on self-adjusting computation has demonstrated that this paradigm can lead to significant improvements in the running time of applications that are based on a broad range of regular algorithms [2]. This paradigm seems especially attractive for multi-core platforms given the fact that many parallel programs, games and media-processing kernels use algorithms that tend to be relatively stable under small changes to their data (e.g., [4])—stable algorithms benefit from self-adjusting computations [1]. Since existing techniques implicitly assume sequential execution semantics they can not directly be applied to these fundamentally parallel applications. What makes this particularly interesting is the fact that it is not obvious if existing techniques can be parallelized efficiently. This paper presents the first attempt at developing techniques for parallel self-adjusting computation. Preliminary results demonstrate the potential for this approach for multi-core platforms.

There are two key aspects in making a program self-adjusting: first, to identify data that can change and second, to provide efficient mechanisms to track these changes at runtime. Generally, data that can change over time is abstracted out into a *modifiable reference*, or *modifiable* for short, and the underlying run-time system tracks operations on modifiables by recording the *dynamic dependence graph* of operations. Internally, a *change propagation algorithm* is used to update computations as well as outputs in response to changes in the modifiable state (inputs or new modifiables that created during execution). Fundamentally, this algorithm identifies all affected `read` operations, which track operations that depend on the value of a modifiable, and re-executes them. Note that each re-execution can change the contents of other modifiables and thus "wake up" other `read` operations. Previous work shows that when combined with a particular form of memoization, change propagation can be effective for a reasonably broad range of applications undergoing both discrete [2] and kinetic changes [5].

A key challenge for parallelizing self-adjusting computations is the fact that existing techniques have been crafted assuming sequential execution semantics. For example, consider change propagation: since any `read` operation can potentially change the modifiable state and, in the presence of control-flow branches, can even invalidate other `read` operations, for correctness it is critical that the `read` operations be executed in their sequential order. In a serial execution setting, change propagation can be done efficiently by using totally ordered timestamps to represent crucial invariants in the execution trace [3]. Note that while this mechanism enables optimality ($O(1)$-overheads) in a sequential setting, the serialization constraints prevent parallelization. Another challenge concerns the fact that existing approaches treat modifiables declaratively. [1] For efficient parallelization, it may be desirable to explore an imperative semantics for modifiables. Such a semantics can allow modifiables to be named before they are written and to be written directly. For example, this approach enables writing a parallel version of quick-sort without an append operation.

This paper describes a proposal for parallelizing self-adjusting computation techniques and presents some preliminary results. For expressing parallelism, we rely on a `letpar` primitive whose bindings evaluate in parallel. This primitive suffices to express series-parallel computations that are seen in many divide-and-conquer algorithms. Section 2 formalizes a tiny language, called PAL that extends an the untyped

[1]In previous work modifiables are essentially purely functional language constructs: observationally they are pure (their implementation, however, is imperative).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2002 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

$$
\begin{array}{llll}
\textit{Values} & v & ::= & \star \mid x \mid l \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\
\textit{Expressions} & e & ::= & v \mid v_1\ v_2 \\
 & & \mid & \texttt{letpar}\ x = e\ \texttt{and}\ y = e\ \texttt{in}\ e\ \texttt{end} \\
 & & \mid & \texttt{mod}\ x\ \texttt{in}\ e\ \texttt{end} \\
 & & \mid & \texttt{write}\ v\ \texttt{into}\ l \\
 & & \mid & \texttt{read}\ v\ \texttt{as}\ x\ \texttt{in}\ e\ \texttt{end} \\
\textit{Traces} & \texttt{T} & ::= & \texttt{write}\ l \leftarrow v \mid \texttt{mod}\ l.\texttt{T} \\
 & & \mid & \texttt{read}_{l \to x = v.e}\ .\texttt{T} \\
 & & \mid & \texttt{letpar}\ \texttt{T}_1\ \texttt{and}\ \texttt{T}_2.\texttt{T}_3
\end{array}
$$

Figure 1: The abstract syntax of PAL.

lambda calculus with the letpar primitive and with support for imperative modifiables and parallel change propagation. Section 3 outlines algorithmic techniques for constructing dependence graphs and for parallel change propagation. The change-propagation algorithm takes advantage of the parallelism expressed by the program (which is recorded in the dependence graph) by executing parallel reads in parallel. In Section 4, we describe a prototype implementation and present some preliminary experimental results.

2. LANGUAGE

Parallel Adaptive Language (PAL) is a small language, analogous to AFL [3] that was developed to describe sequential self-adjusting computations. Figure 1 shows the abstract syntax. Apart from the standard functional constructs, the language includes three constructs `mod`, `read`, and `write` for creating, reading and writing modifiables repsectively, and a `letpar` construct for writing parallel expressions. The PAL language treats modifiables imperatively—they are named before being written—rather than purely functional. We note, however, that for correctness of change propagation, it is required that modifiables be written at most once, and never be read before they are written. These properties can be ensured by constructing a substructural type system with capabilities [12][9]. Due to space constraints we do not present the type system in this paper.

Figure 2 shows the big-step dynamic semantics of PAL. Evaluation of an expression yields a trace, written by T and its variants. A trace records the operations on modifiables and the `letpar` operations. Syntax for traces is given in Figure 1.

The interesting evaluation judgements concern `letpar` and the operations on modifiables. The evaluation of `mod,read` and `write` constructs are similar to the previous work on the AFL language [3] except that PAL treats modifiables imperatively. To evaluate a `letpar` construct, we first evaluate the bindings in parallel, and then evaluate the body after substituting the values of parallel expressions. Before evaluating the body, the modifiables created by the parallel evaluations must be combined in the same store by a *store-combine* operation, written as $\Sigma(\sigma, \sigma_1, \sigma_2)$. Although the evaluations are in parallel, it is critical that the freshness of allocated modifiables be preserved across parallel evaluations. [2] We assume that $\Sigma(\sigma, \sigma_1, \sigma_2)$ "alpha-varies" the stores to ensure freshness. Due to space constraints, we do not formalize the $\Sigma(\cdot, \cdot, \cdot)$ operation here.

[2]Since memory allocation is treated sequentially even in the parallel setting, this property is guaranteed trivially in a real implementation.

(values)

$$\sigma, v \Downarrow v, \sigma, \varepsilon$$

(apply)

$$\frac{(v_1 = \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}) \qquad \sigma, [v_1/f, v_2/x]\ e \Downarrow v, \sigma', \texttt{T}}{\sigma, v_1\ v_2 \Downarrow v, \sigma', \texttt{T}}$$

(letpar)

$$\frac{\begin{array}{c}(\sigma, e_1 \Downarrow \sigma_1, v_1, \texttt{T}_1) \mid\mid (\sigma, e_2 \Downarrow \sigma_2, v_2, \texttt{T}_2) \\ \Sigma(\sigma, \sigma_1, \sigma_2), [v_1/x, v_2/y]\ e_2 \Downarrow \sigma_3, v_3, \texttt{T}_3\end{array}}{\begin{array}{c}\sigma, \texttt{letpar}\ x = e_1\ \texttt{and}\ y = e_2\ \texttt{in}\ e_3\ \texttt{end} \Downarrow \\ \sigma_3, v_3, \texttt{letpar}\ \texttt{T}_1\ \texttt{and}\ \texttt{T}_2.\texttt{T}_3\end{array}}$$

(mod)

$$\frac{\sigma[l \leftarrow \square], [l/x]\ e \Downarrow \sigma', \star, \texttt{T}}{\sigma, \texttt{mod}\ x\ \texttt{in}\ e\ \texttt{end} \Downarrow \sigma', l, \texttt{mod}\ l.\texttt{T}}$$

(read)

$$\frac{\sigma, [\sigma[l]/x]\ e \Downarrow \sigma', \star, \texttt{T}}{\sigma, \texttt{read}\ l\ \texttt{as}\ x\ \texttt{in}\ e\ \texttt{end} \Downarrow \sigma', \star, \texttt{read}_{l \to x = v.e}\ .\texttt{T}}$$

(write)

$$\sigma, \texttt{write}\ v\ \texttt{into}\ l \Downarrow \sigma[l \leftarrow v], \star, \texttt{write}\ l \leftarrow v$$

Figure 2: Dynamic semantics for PAL.

(empty)

$$\sigma, \varepsilon \curvearrowright \sigma, \varepsilon$$

(letpar)

$$\frac{\begin{array}{c}(\sigma, \texttt{T}_1 \curvearrowright \sigma_1, \texttt{T}'_1) \mid\mid (\sigma, \texttt{T}_2 \curvearrowright \sigma_2, \texttt{T}'_2) \\ \Sigma(\sigma, \sigma_1, \sigma_2), \texttt{T}_3 \curvearrowright \sigma_3, \texttt{T}'_3\end{array}}{\sigma, \texttt{letpar}\ \texttt{T}_1\ \texttt{and}\ \texttt{T}_2.\texttt{T}_3 \curvearrowright \sigma_3, \texttt{letpar}\ \texttt{T}'_1\ \texttt{and}\ \texttt{T}'_2.\texttt{T}'_3}$$

(mod)

$$\frac{\sigma, \texttt{T} \curvearrowright \sigma', \texttt{T}'}{\sigma, \texttt{mod}\ l.\texttt{T} \curvearrowright \sigma', \texttt{mod}\ l.\texttt{T}'}$$

(read/no ch.)

$$\frac{\sigma[l] = v \qquad \sigma, \texttt{T} \curvearrowright \sigma', \texttt{T}'}{\sigma, \texttt{read}_{l \to x = v.e}\ .\texttt{T} \curvearrowright \sigma', \texttt{read}_{l \to x = v.e}\ .\texttt{T}'}$$

(read/ch.)

$$\frac{\sigma[l] \neq v \qquad \sigma, [\sigma[l]/x]\ e \curvearrowright \sigma', \texttt{T}'}{\sigma, \texttt{read}_{l \to x = v.e}\ .\texttt{T} \curvearrowright \sigma', \texttt{read}_{l \to x = \sigma[l].e}\ .\texttt{T}'}$$

(write)

$$\sigma, \texttt{write}\ l \leftarrow v \curvearrowright \sigma, \texttt{write}\ l \leftarrow v$$

Figure 3: Parallel change propagation.

Figure 3 shows the judgements for parallel change-propagation. Change propagation mimics the dynamic semantics. It traverses the trace and seeks for affected reads that read locations whose contents have changed. Affected reads are re-evaluated and the resulting trace is substituted in place of the original trace. As in the dynamic semantics, the parallel parts of `letpar` expression are change propagated in parallel.

3. PARALLELIZATION

Since the change-propagation judgements given in Figure 3 scan the whole trace, they do not translate to an efficient algorithm—not even a serial one. In the case of sequential computation, change-propagation can be performed efficiently by representing traces via *dynamic dependence graphs* [3]. This representation hinges on a total ordering of the read operations and thus crucially relies on sequential execution. We outline a representation for traces that is suitable for parallel computation and give a parallel change-propagation algorithm.

Representing traces. We represent a trace as a dependence graph consisting of modifiables, a computation-tree, and read and write edges between modifiables and the nodes of the computation tree. A trace of the form `mod` l.`T` is represented by a modifiable labeled with l, and a computation-tree representing `T`. A trace of the form $\texttt{read}_{l \rightarrow x = v.e}$.`T` is represented by the computation-tree of `T` and an edge from l to the root of the tree of `T`. A trace of the form `write` $l \leftarrow v$ is represented by a computation-tree consisting of a single node and an edge from that node to the modifiable l.

The *computation-tree* consists of *par* ($\diamond$), *seq* ($\triangle$), and *read* ($\circ$), and *write* ($\bullet$) nodes. A trace of the form `letpar` T_1 `and` $T_2.T_3$ is represented as a computation-tree with a `seq` node whose left child is a `par` node and whose right child is the representation for T_3. The children of the `par` node are the representations for T_1 and T_2. We represent the computation-tree for the trace of the form $\texttt{read}_{l \rightarrow x = v.e}$.`T` with a tree with root s (a read node), whose only child is the root of the subtree that represents `T`. We tag s with x, v, e. The computation-tree for a trace of the form `write` $l \leftarrow v$ is a leaf.

We summarize the mapping from a trace `T` into a DDG, written as $[\![\texttt{T}]\!]$, as follows.

$$[\![\texttt{letpar T}_1\ \texttt{and T}_2.\texttt{T}_3]\!] = \begin{cases} \triangle \text{ with left child } \diamond \text{ (children } [\![\texttt{T}_1]\!], [\![\texttt{T}_2]\!]) \text{ and right child } [\![\texttt{T}_3]\!] \end{cases}$$

$$[\![\texttt{read}_{l \rightarrow x = v.e}\ .\texttt{T}]\!] = \begin{cases} \circ \leftarrow \Box_l \\ [\![\texttt{T}]\!] \end{cases}$$

$$[\![\texttt{mod}\ l.\texttt{T}]\!] = \begin{cases} \Box_l \\ \quad [\![\texttt{T}]\!] \end{cases}$$

$$[\![\texttt{write}\ l \leftarrow v]\!] = \bullet \rightarrow \Box_l$$

Figure 4 shows the computation tree of a code snippet with inputs x and y and result r. The code computes $2 * x' - 3 * y'$ where x' and y' are the contents of the modifiables x and y.

Parallel change propagation. After a self-adjusting program executes, the contents of any of the modifiables can be changed and the computation can be updated by running a parallel change-propagation algorithm. As described in the formal semantics (Figure 3), this process involves systematically finding affected reads (those which read a modifiable whose contents have changed) and re-executing these in a safe order. The process finishes when every affected read has been handled. We first give an algorithm for performing parallel change propagation. We then describe how the algorithm can be implemented efficiently.

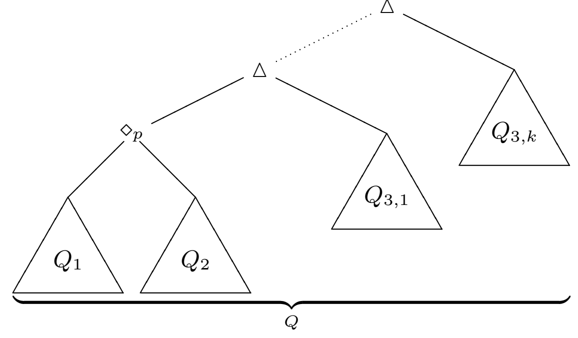

Figure 5: split(Q, p) yields disjoint sub-queues Q_1, Q_2 and $Q_3 = Q_{3,1} \cup \cdots \cup Q_{3,k}$ which are constrained to only hold reads in the respective sub-trees.

```
propagate(Q, s) =
  x ← next(Q, s) in
  case x of
    NONE ⇒ (Q, s)
  | SERIAL(r) ⇒
      Q'  ← Q \ {r}
      Q'' ← exec r under Q'
      propagate(Q'', s)
  | PARALLEL(r1, r2) ⇒
      p ← LCA(r1, r2)
      s1, s2 ← children(p)
      Q1, Q2, Q3 ← split(Q, p)
      in parallel do {
        Q1' ← propagate(Q1, s1)
        Q2' ← propagate(Q2, s2)
      }
      Q' ← Q1' ∪ Q2' ∪ Q3
      propagate(Q', s)
```

Figure 6: Abstracted change propagation algorithm.

Specifically, we are interested in the following operations on sub-trees and queues:

- next(Q, s) indicates the kind of work that remains to be performed in the computation-tree rooted at s. The operation returns `NONE` if there are no affected reads in the subtree. It returns `SERIAL`(r) if r is an affected read that must be executed before all the other affected reads. It returns `PARALLEL`(r_1, r_2) if two independent/parallel reads, r_1 and r_2, are the earliest affected reads.

- split(Q, p), returns three sub-queues, Q_1, Q_2 and Q_3. The Q_1 and Q_2 are queues consisting of the reads in Q that are in the left and in the right subtree of p respectively. The queue Q_3 holds remaining reads of Q. Figure 5 illustrates the operation.

Once we assume these operations are available the algorithm itself is a straightforward (Figure 6): each invocation of the algorithm is given a queue Q, which is initialized to contain the reads that are first affected by any changed modifiables, as well as a sub-tree s which constrains the invocation to re-execute reads only under s. The initial invocation has s as

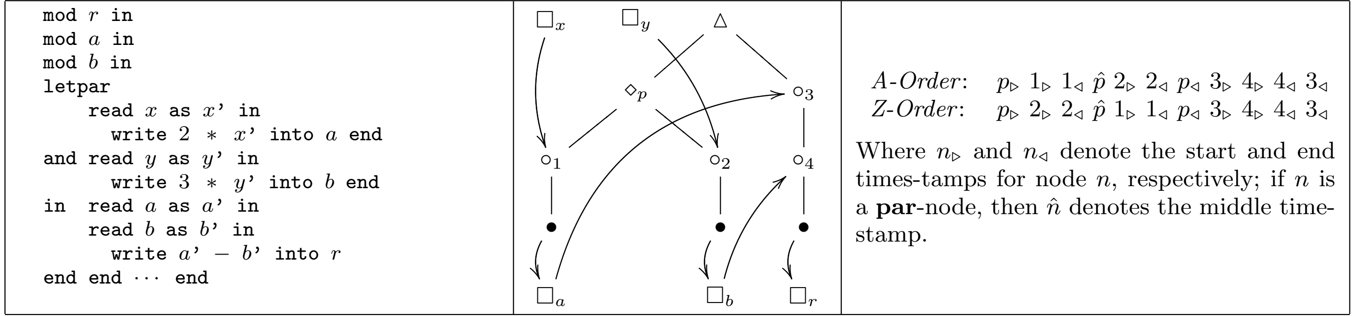

Figure 4: A code snippet and its computation tree and total orderings, which make up its DDG.

the root of the entire program's computation-tree. For each invocation, we query Q by using next and case on its result. In the serial case we re-execute the given read and continue propagating changes. In the parallel case we find a suitable split node as the least-common-ancestor of the parallel reads r_1 and r_2 and continue, in parallel, to propagate changes in each sub-queue after using split. Once these sub-queues are exhausted we merge the results and again continue. The recursive/parallel process finishes when every queue has been exhausted.

We describe data-structures for implementing the parallel change propagation algorithm efficiently. In this paper, we only consider non-parallel (sequential) data structures. Although these data structures may suffice for a multi-core implementation with small (< 100) number of cores, a scalable implementation may require concurrent versions of these data structures.

Two total orders. To enable change propagation, we maintain two separate total orderings, *A-order* and *Z-order*, of the computation-tree. Each total order is consistent with an in-order traversal of seq-nodes. In the A-order, the left subtree of par-nodes come before the right subtree. In the Z-order, the right subtree of the par-nodes come before the left subtree. Using these orderings, we assign an A- and a Z- *interval* to each read. The A-interval (Z-interval) consists of two time stamps corresponding to the start and end time of that read in the A-order (Z-order). We maintain each ordering by using a constant-time order-maintenance data structure [10][6]. Using these orderings, for all pairs of reads r_1, r_2 we can check in constant time if re-execution of r_1 must precede r_2 or not: r_1 precedes r_2 in both orderings if this is the case, and if the orderings disagree then r_1 and r_2 are independent. Again, in Figure 4 we give the corresponding orderings.

Although we define the orderings by a transversal of a completed computation tree it should be noted that in practice the orderings are created and updated as the tree is grown (*i.e.*, the tree and total orderings are created and altered together using mutable data structures for an efficient implementation).

Priority queues. We require a priority queue that supports findMin, deleteMin, split, and merge operations. These operations can be performed in logarithmic time in the size of the queue by using binary search trees (e.g., treaps [11]).

```
propagate (Q_a, Q_z, t_a, t_z) =
1  if Q_a = ∅ then
2     return ∅

3  r_a ← findMin (Q_a)
4  r_z ← findMin (Q_z)
5  if r_a = r_z then
6     Q'_a ← deleteMin (Q_a)
7     Q'_z ← deleteMin (Q_z)
8     ((t^0_a, t^1_a), (t^0_z, t^1_z)) ← interval r_a
9     deleteInterval (t^0_a, t^1_a)
10    deleteInterval (t^0_z, t^1_z)
11    Q''_a ← Q'_a \ {r | (interval(r)) ⊑ (t^0_a, t^1_a) }
12    Q''_z ← Q'_z \ {r | (interval(r)) ⊑ (t^0_z, t^1_z) }
13    A ← execute r_a with t_a, t_z) in ((t^0_a, t^1_a), (t^0_z, t^1_z))
14    (Q'''_a, A') ← merge (A, Q''_a, t_a)
15    (Q'''_z, A'') ← merge (A', Q''_z, t_z)
16    A_p ← propagate (Q'''_a, Q'''_z, t_a, t_z)
17    return A'' ∪ A_p
18 else
19    s ← LCA(r_a, r_z)
20    (t^{s0}_a, t^{s1}_a, t^{s2}_a) ← timeStampsA (s)
21    (t^{s0}_z, t^{s1}_z, t^{s2}_z) ← timeStampsZ (s)

22    (Q^L_a, Q^R_a, Q^O_a) ← split (Q_a, t^{s1}_a, t^{s2}_a)
23    (Q^L_z, Q^R_z, Q^O_z) ← split (Q_z, t^{s1}_z, t^{s2}_z)
24
25    in parallel do
26       A^0_p ← propagate (Q^L_a, Q^L_z, t^{s1}_a, t^{s1}_z)
27       A^1_p ← propagate (Q^R_a, Q^R_z, t^{s2}_a, t^{s2}_z)

28    A ← A^0_p ∪ A^1_p
29    (Q'_a, A') ← merge (A, Q^O_a, t_a)
30    (Q'_z, A'') ← merge (A', Q^O_z, t_z)

31    A^2_p ← propagate (Q'_a, Q'_z, t_a, t_z)
32    return A'' ∪ A^2_p
```

Figure 7: The pseudo-code for change propagation.

Least-common ancestors. The parallel change-propagation algorithm requires a dynamic data structure for finding least common ancestors of read-nodes of the computation-tree. Such a data structure requires $O(1)$ time [8].

The algorithm revisited. Now we give a more detailed account of the parallel change propagation algorithm. Figure 7 shows the pseudo-code for the algorithm. The algorithm takes as arguments two priority queues Q_a and Q_z and the time stamps t_a and t_z. The queue Q_a (Q_z) prioritizes the reads with respect to their start time in the A-ordering (Z-

ordering). The time stamps t_a and t_z are the last time stamps that the reads in each queue can have in both orderings. These queues are initialized to the reads of the modifiables whose values are changed externally and t_a and t_z are set to the greatest time stamps in each ordering.

The algorithm completes when there are no more reads to execute and returns the set of reads that become affected during change propagation (due to executed writes) and that have timestamps greater than t_a (and symmetrically t_z). The algorithm starts by accessing the minimum reads, r_a and r_z, in each queue. If the reads (r_a and r_z) are equal, then the read is re-executed after the time stamps within its interval are deleted and the reads contained in the interval are removed from the queues. Re-execution returns the reads that become affected as a set (A). After re-execution, the affected reads are inserted into each queue if they start before t_a and t_z respectively. Change propagation then continues on the updated queues and the algorithm finishes by returning all affected reads that start after t_a (or symmetrically t_z). This part of change propagation is similar to the sequential case, except that the affected reads have to be treated somewhat differently because modifiables may be allocated and written imperatively.

If the reads (r_a and r_z) are not equal, then the algorithm finds their least common ancestor s (a `letpar` node) in the DDG. The algorithm then finds the time stamps of s (since s is a `letpar` node it has three time stamps) and splits the queues into three parts, left (L), right (R), and other (O). The left queues Q_a^L, Q_z^L contain the reads in the intervals (t_a^{s0}, t_a^{s1}) and (t_z^{s1}, t_z^{s2}); the right queues Q_a^R, Q_z^R contain the reads in the intervals (t_a^{s1}, t_a^{s2}) and (t_z^{s0}, t_z^{s1}); and the rest of the reads are placed into the queues Q_a^O, Q_z^O. The three parts L, R and O correspond to the three subtrees of the `letpar` node s (this roughly corresponds to the split operation discussed earlier and illustrated in Figure 5). The algorithm then performs change propagation on the left and the right subtrees in parallel, and updates the queues for the rest of the reads by inserting the reads that become affected. The algorithm then continues change propagation on the rest of the reads and returns the set of affected reads greater than t_a (and symmetrically t_z).

4. IMPLEMENTATION

Our prototype implementation is still being developed and thus uses mostly sub-optimal versions of the data-structures discussed in Section 3. In order to give a preliminary evaluation of the algorithm we thus substitute wall-clock times with results obtained by using counters for *work* (denoted as W) and *depth* (denoted as D). We measure work of a computation as the number of reads in its trace; and intuitively, the depth of a computation measures the work of the longest sequential execution path, ignoring parallel tasks with smaller work. Formally work and depth can be found recursively given a trace as follows:

$$\begin{aligned}
\mathsf{W}(\texttt{write}\ l \leftarrow v) &= 0 \\
\mathsf{W}(\texttt{mod}\ l.\texttt{T}) &= \mathsf{W}(\texttt{T}) \\
\mathsf{W}(\texttt{read}_{l \rightarrow x = v.e}\ .\texttt{T}) &= \mathsf{W}(\texttt{T}) + 1 \\
\mathsf{W}(\texttt{letpar T}_1\ \texttt{and T}_2.\texttt{T}_3) &= \mathsf{W}(\texttt{T}_1) + \mathsf{W}(\texttt{T}_2) + \mathsf{W}(\texttt{T}_3)
\end{aligned}$$

$$\begin{aligned}
\mathsf{D}(\texttt{write}\ l \leftarrow v) &= 0 \\
\mathsf{D}(\texttt{mod}\ l.\texttt{T}) &= \mathsf{D}(\texttt{T}) \\
\mathsf{D}(\texttt{read}_{l \rightarrow x = v.e}\ .\texttt{T}) &= \mathsf{D}(\texttt{T}) + 1 \\
\mathsf{D}(\texttt{letpar T}_1\ \texttt{and T}_2.\texttt{T}_3) &= \max\{\mathsf{D}(\texttt{T}_1), \mathsf{D}(\texttt{T}_2)\} + \mathsf{D}(\texttt{T}_3)
\end{aligned}$$

The crucial distinction between these definitions is how they treat parallel sub-traces: their work is summed whereas their depth is taken as a max. During change-propagation, we again measure work and depth, but this time we limit the trace nodes that are counted to only those that correspond to nodes in the DDG which are re-executed are newly created during re-execution.

Evaluation. By using work and depth as metrics for evaluating the algorithm we can estimate the theoretical running-time given an implementation that actually uses the data-structures we described earlier. Moreover, by examining the ratio of W/D for a given program, we can estimate the expected amount of parallelism in both initial runs of programs written in PAL as well as the amount of parallelism to expect from the fully-implemented change-propagation algorithm.

Previous work [2] considers varying *batch-sizes* for running change-propagation, where a *batch* is simply a set of changes that are fed to the change-propagation algorithm simultaneously. This work showed empirically that larger batch sizes hide the overhead of adaptive runtime bookkeeping and change-propagation by amortizing this cost across many changes that would otherwise require re-execution "from-scratch" outside of the adaptive framework. Here we consider varying batch-sizes to estimate how parallelism of the change-propagation algorithm increases as batch-sizes increase. We are also interested in showing, again using our metrics, that the parallel change-propagation algorithm as well as the initial runs of a program aren't affected aversely by the introducing parallelism to each. As we will show, our parallel approach accomplishes these goals: the work of initial runs and change-propagation are asymptotically equivalent to serial versions of previous work and still yield varying levels of parallelism for each.

Figure 8 shows measurements of the quick-sort algorithm implemented in our C library. In this implementation the inputs, outputs and intermediate results are represented as linked-lists interlaced with modifiables, as in previous work. [3] In our experiments we use randomly generated inputs. For all six graphs, we consider input sizes from 1 to 2^{16} (64k). The first three graphs concern the initial runs of the quick-sort algorithm on these input sizes, whereas the later three graphs concern change-propagation on these initial runs with batch-sizes plotted as separate lines and ranging from 1 to 2^{12} (4k). The change-propagation plots are given in log-scale in the y-axis. Since the run-time of quick-sort varies with the quality of the pivot choices, there is some variation in our results. Each data-point thus results from an average of 64 runs.

In the first graph we see that, as we would expect from a quick-sort implementation, the work done for the initial run is $O(n \log n)$. In the second graph we consider the depth

[3] This version of quick-sort exploits the fact that modifiables in PAL are treated imperatively by building its sorted output list in-place and in parallel rather than using an additional append operation to join sorted lists found recursively.

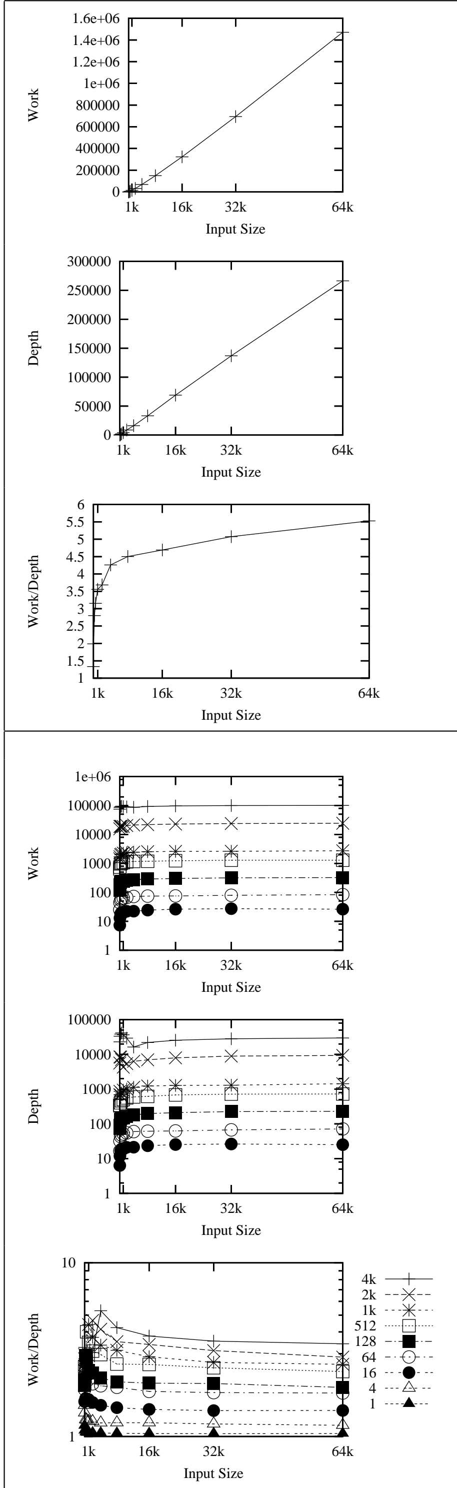

Figure 8: W, D and W/D for initial runs (top 3) and change-propagation runs of various batch sizes (bottom 3) for Quick-Sort

as a function of input-size and see that this grows linearly. This is what we would expect from a list-based implementation of quick-sort since each level of recursion does half the work of the previous level, in expectation, and the first level does linear work. Finally, in the third graph we take the ratio of work and depth to plot the theoretical amount of parallelism that this version of quick-sort exhibits. As we can see, this grows logarithmically, as we would expect from $\mathsf{W}/\mathsf{D} = (n \log n)/n = \log n$.

In the second group of graphs, we consider the same initial runs and plot the work and depth of change-propagation for various batch-sizes. In these experiments we setup change-propagation by creating a new list of the given batch-size and append it as a change to the original input list. Then we run change-propagation which eventually modifies the sorted output list, putting the new elements in their proper positions there. By propagating m new elements through these traces we would expect to execute $O(m \log n)$ reads in total since we have an expected $O(\log n)$ levels of quick-sort, and each of these has m new elements to handle. In the forth graph we see that our results confirm this.

It is not difficult to show that the depth of change-propagation for quick-sort with m changes is $O(m + \log n)$. As can be seen in the fifth graph, the experiments confirm this bound.

In the last graph we consider the ratio of the previous two graphs, as before for the initial run graphs. The ratio W/D gives us a measure of parallelism given an initial list size and a batch-size for changes. When the batch-size is very low (*e.g.*, 1), we see no parallelism at all, as $\mathsf{W}/\mathsf{D} = 1$. However, when the batch sizes increase we note that the parallelism of the change-propagation algorithm approaches that of the initial run itself. We expect, and confirm experimentally, that $\mathsf{W}/\mathsf{D} = (m \log n)/m = \log n$ when $m > \log n$, and otherwise find that $\mathsf{W}/\mathsf{D} = (m \log n)/\log n = m$ for smaller m.

Given a sufficiently large n and m, we see parallelism that is suitable for machines with several cores. The parallelism can be further increased beyond this by moving from list representations to trees [7] for storing input, output and intermediate results.

5. FUTURE WORK

Static semantics. We've given syntax and dynamic semantics for PAL as well as an informal description of "proper usage", which includes ensuring that modifiables are always written before being read, and that each modifiable is written at most once. Future work should include a static semantics which ensures that these conventions are followed and an associated soundness result. We suspect a substructural type system which gives unwritten modifiables linear types will suffice for this, which may resemble type systems used for ensuring safe but explicit memory management in other work [12][9].

Concurrent data structures. The key data structures required by the dependence tracking and the change-propagation algorithm can be implemented in constant time in the sequential case, except for the priority queue data structure that requires logarithmic time. In the parallel case, it would be possible to use the same data structures with proper locking techniques, but it is important to develop parallel data structures that scale as the number of processors increase.

Memoization. In the serial version of change-propagation from previous work [2], a special type of memoization is used to avoid re-executing nested reads when these reads

aren't affected by changes even if their ancestors in the DDG are affected. This technique requires special annotations in the code, additional runtime data-structures as well as additional details in the change-propagation algorithm itself. To provide a similar technique in the parallel case, we must extend our approach similarly.

Other features. Our PAL language offers limited support for writing parallel programs: they can essentially fork and later synchronize, but parallel threads of control can't otherwise communicate. It may be interesting to provide communication features such as message-passing and/or shared memory in future iterations of PAL while still giving useful semantics for adapting to changes.

6. CONCLUSION

This paper describes techniques for parallel self-adjusting computation. We describe a language PAL with a fork-based parallelism primitive for writing parallel self-adjusting programs. Programs written in PAL can be executed in parallel on a given input. In addition, the input can be changed, and the changes can be propagated to update the output by using a parallel change propagation algorithm. We describe the parallel change propagation algorithm and describe some data structures for implementing the algorithm. The parallel change-proapgation algorithm executes tasks in parallel whenever possible. We present preliminary experimental results based on simulations. The experiments show that the approach realizes the parallelism expressed in the program accurately both during execution and during change propagation.

7. REFERENCES

[1] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[2] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[4] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[5] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic algorithms via self-adjusting computation. Technical Report CMU-CS-06-115, Department of Computer Science, Carnegie Mellon University, March 2006.

[6] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 133–144, New York, NY, USA, 2004. ACM Press.

[7] Guy E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[8] Richard Cole and Ramesh Hariharan. Dynamic lca queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.

[9] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 262–275, New York, NY, 1999.

[10] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 365–372, 1987.

[11] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4–5):464–497, 1996.

[12] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071, 2001.