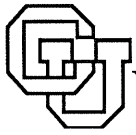
SEQUENCE COMPARISON FOR PARALLEL EXECUTIONS

Zulah K.F. Eckert and Gary J. Nutt

CU-CS-726-94



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Sequence Comparison For Parallel Executions

CU-CS-726-94 1994

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

Sequence Comparison for Parallel Executions

Zulah K. F. Eckert* and Gary J. Nutt
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO, 80309-0430
email:{eckert,nutt}@cs.colorado.edu

Abstract

Event traces are a fundamental tool for studying the performance of multiprocessors. While there has been considerable development of tools and techniques, there has been surprisingly little theoretical work on the feasibility and complexity of tools that manipulate and use traces. In many cases, there is no sound knowledge about whether or not a technique is scalable or even feasible for large program executions.

When extracting traces from parallel programs, a single data set may produce a set of possible traces due to program and/or system nondeterminism. Given an extracted trace, one should ask questions such as "is the trace representative of all possible traces that could have resulted?". In this paper, we focus on the fundamental problem of comparing two parallel program executions.

Sequence comparison seeks to compute both optimal correspondences and distances between sequences. Existing sequence comparison techniques are not sufficient to compare traces. Accurate comparison methods exist however they have quadratic worst case time complexity. This is unreasonable for parallel program traces due to their potentially large size. While techniques that have a linear worst case time complexity exist, they produce inaccurate results.

We present an algorithm for corresponding program executions that has linear worst case complexity and negligible space usage. In previous work, we characterize parallel program behavior with regard to instrumentation [3]. Our algorithm is based upon this work and relies on information gathered from static program analysis to facilitate a fast and accurate correspondence. We prove the algorithm correct and show that is has a linear worst case time complexity. We demonstrate how the correspondence algorithm can be used to compute distance measures. We discuss the possibility of a reasonable algorithm to compute optimal correspondences between parallel program execution. We conjecture that such algorithm

* Supported by a grant from Convex Computer Corporation

does not exist and provide motivation for our conjecture.

1. Introduction

Event traces are a common tool for capturing the behavior of a program on a specific computer. Traces may be analyzed to determine the way that the program behaved, or they may be used to characterize program load and used to drive a simulation system. A multiprocessor trace is a composition of a set of sequential traces for a parallel program; each schedulable unit of computation produces a sequential trace, and the multiprocessor trace is the result of interleaving the sequential traces in the realtime order that their respective events occurred.

A multiprocessor trace from a nondeterministic system and/or program represents one possible global order on the occurrence of the events; if the program were to be executed repeatedly on the same system with the same data set, then one would expect to obtain a set of different traces (reflecting the different order that events occurred due to nondeterminism). In light of these differences, performance specialists that use traces need to ask several theoretical questions about a trace, for example: is a particular trace representative of the set? Are the differences among the set of traces significant with regard to the analysis? What is the complexity of an algorithm to compare two traces from the same program? The major result from the paper is an algorithm that can compare two different multiprocessor traces in linear time (in a single pass over each trace) and with negligible space usage.

In this paper, we focus on the more general notion of a program *execution*. We base our notion of execution on Lamport's model of executions (see [8]) and present a detailed version of our model in [3]. Given a parallel program and a set of instructions for the program, an execution is a set of instruction instances together with a temporal and shared data dependence order.

The goal of this paper is to provide a reasonable algorithm for comparing parallel program executions. Because of the potentially large size of executions (and traces) of parallel programs, possibly gigabytes [1], we contend that any reasonable algorithm for must execute on-the-fly (i.e., linear time and negligible space usage). For this reason, current sequence analysis techniques cannot be applied to parallel program execution because they yield either inaccurate results or have quadratic execution time.

Executions cannot be corresponded, in a reasonable amount of time and space, based solely on their information content. We propose that a reasonable algorithm can only be achieved by using information gathered from program source to a priori correspond executions[1]. That is, the program can be a valuable

1 It is in general against the grain of sequence comparison to a priori identify a correspondence between sequences. However in this case as we have pointed out, there is no other reasonable algorithm.

source of information which, as we show, can direct the correspondence of two executions. In this paper, we present an algorithm for corresponding executions based upon program source information. It should be noted that this correspondence is not guaranteed to be optimal however the comparison is computed on-the-fly. The algorithm computes a correspondence between two executions and can be modified to produce a distance measure.

Given the assumption that parallel programs are structured (i.e., they contain no **goto** statements and no **fork** statements that have no corresponding **join**) we make two key observations about executions. First, two executions will exhibit periods of convergence and divergence. During a period of convergence, two executions are identical while during a period of divergence they are disjoint (e.g., share no common instruction instances). Second, the instructions causing convergence or divergence can be identified at program source level and later located in any execution. Divergence points were introduced in [6][2] by Holliday and Ellis and further characterized in [4] [3] where they are referred to as *trace change points*. Trace change points (TCPs) are exactly the instructions in a parallel program whose outcome can change over subsequent runs of the program (on the same data) due to dependence, either direct or indirect, on shared variable values. Convergence points are a new concept that we introduce in this paper. A convergence point (CP) is the first instruction that any execution is *guaranteed* to execution after a TCP is encountered. Using TCP and CP information, we can locate the portions of any two executions that are identical and those that are disjoint resulting in a correspondence of the executions.

In this paper, we prove properties of TCPs and CPs that allow the on-the-fly calculation of a correspondence between program executions. We give the conditions sufficient for any correspondence to be considered reasonable or correct. We present an algorithm to compute one possible correspondence and demonstrate the correctness and complexity of the algorithm. We present one possible measure of difference for parallel program executions and demonstrate the use of the correspondence algorithm in computing this difference. Finally, we discuss other possible choices of correspondence and the existence of an algorithm computing an optimal correspondence given the time and space constraint necessary for parallel program executions. We conjecture that such an algorithm does not exist and justify our particular choice of correspondence.

2. Background

During the 1960's sequence comparison was an active area of research for such practical problems as string correction and editing. During the 1970 and early 80s, sequence comparison found applications

2 Holliday and Ellis refer to divergence points as *address change points*.

in molecular biology, human speech recognition, encoding information and error checking, and a large variety of other scientific applications. Today, there is a large body of work in sequence comparison. Hall and Dowling [5] provide a good overview of sequence comparison in computer science, while Sankoff and Kruskal [9] provide a more detailed presentation of sequence comparison.

Sequence comparison seeks to compute *distance measures* and determine optimal *correspondences* between sequences. An optimal correspondence for two sequences is one that minimizes a particular distance. A sequence is an ordered list of elements taken from an alphabet. If two sequences differ, then it is a result of either substitutions, insertions or deletions, compressions or expansions, or transpositions (or swaps). A substitution is a replacement of one element for another. Insertions or deletions is simply the insertion or deletion of an element to or from a sequence. A compression is a replacement of two or more elements with a single element while an expansion is a replacement of a single element by two or more elements. A transposition is an interchange of two adjacent elements.

Distance calculations, such as Hamming distance (the number of positions in which two sequences differ) and Euclidean distance ($[\sum_{i=1}^{n}(a_i - b_i)^2]^{1/2}$) are intended to be used on sequences of the same length. For both of these calculations, it is assumed that no further correspondence of the sequences need be computed. Knowing the correspondence between two sequences a priori simplifies the comparison process in that a correspondence identifies comparable elements in two sequences. Given this information, a distance calculation, such as the two above, can be computed in time linear in the size of the input sequences. In addition, the space requirements for such a calculation are constant.

Unfortunately, it is possible that either a correspondence is not known a priori (e.g., the sequences differ in length) or a more accurate distance measure is needed. In either case, a correspondence must be computed. Levenshtein distances (and other similar distances), are an example of potentially more accurate distance measures. These are the smallest number of substitutions, inserts and deletes require to transform one sequence into another and the smallest number of insertions and deletions required to transform one sequence into another. These distance calculations require selection of a correspondence that minimizes the distance functions and for this reason require quadratic time in the size of the input sequences and storage of these sequences.

In mathematics, the term *distance* refers to any function satisfying the following axioms as presented in [9] :

1. *nonnegative property* d(**a**,**b**) 0 for all **a** and **b**
2. *zero property* d(**a**,**b**) = 0 if and only if **a** = **b**
3. *symmetry* d(**a**,**b**) = d(**b**,**a**) for all **a** and **b**

4. *triangle inequality* $d(\mathbf{a},\mathbf{b}) + d(\mathbf{b},\mathbf{c}) \; d(\mathbf{a},\mathbf{c})$ for **a**, **b** and **c**

Currently, we know of one example of a comparison algorithm that could be directly applied to two program executions. This is the UNIX* *diff* file comparison filter. This filter, among other things, provides a Levenshtein distance between two files. Therefore, if it were possible to store executions on file, then diff would provide a measure in $O(n^2)$ time.

3. Motivation

Our goal is to be able to reasonably compute distances between parallel program executions. Since parallel program executions have varying size, the problem fits well into the category of sequence comparison problems. Unfortunately, the potentially large size of parallel program executions, possibly gigabytes, rules out the use of quadratic algorithms. This implies that Levenshtein and other accurate distance calculation will be unreasonable for use with large parallel program executions. A simple element by element correspondence, such as that used in the Euclidean and Hamming distances, will not suffice. Consider two executions of the same program that differ only in the number of iterations of a single loop. A simple alignment of the two executions will fail to identify that the executions are identical from the end of the loop to the end of the program. That is, the extra loop iterations will not be treated as an insertion of extra elements into a sequence. We contend that any reasonable algorithm corresponding parallel program executions must correspond these executions in linear time and negligible space.

In this paper, we present a reasonable algorithm for computing a correspondence. An algorithm choosing an optimal correspondence requires quadratic time as well as enough space to store the executions. For this reason, we compute a good correspondence. We discuss our choice of correspondence in Section 8. The algorithm relies on the assumption that every statement has a single entry and exit point. That is, programs are assumed to be structured. Using this assumptions, statements are *pinned* together at their entry and exit points. This pinning results in a correspondence. For example given two executions in which the outcome of two conditional statements is the same, then the distance between these executions is zero (for this portion of the executions). If however the outcome of the conditionals differs then we can pin the entry and exit of the conditional statements together, since in any structured program, they will be the same statement, and the instruction instances between these statements are measured for difference. This difference may be in the form of inserts and deletions or a substitution. Similarly if the outcome of an iterative statement is different in each execution, we choose to correspond iterations *in the*

*UNIX is a trademark of Bell Laboratories.

order that they occur. This leaves any extra iterations with no correspondence and represents an insertion or deletion between the two executions.

To illustrate these ideas, consider the example in Figures 1a-d. The parallel program in Figure 1a contains two processes that first compete for access to a shared variable and then execute a conditional statement B4 and an iterative statement C4 that are dependent on this shared value. Figure 1b demonstrates that this dependency can lead to different possible executions depending on the interleaving of the shared variable access operations. Notice that in the first execution, the value of j is 2 after statement B3 completes and the conditional statement is true, while the iterative statement C4 iterates a single time. In the second execution, the value of j is 1 and the condition fails, while the iterative statement C4 iterations twice. Notice that this results in an increase in the length of the second execution in Figure 1b and represents an insertion of additional elements. Figure 1c is a demonstration of how we correspond these executions. First, statements A0-A1 correspond in both executions (1), as with statements A2-A3 (8). Statements C0-B3 and B0-C3 correspond in both executions (2) because the same statements are being executed. We correspond the first iteration of the iterative statement C4 (3) and the second check on the index value (6). However the second iteration of the loop in execution 2 (7) has no correspondence to the

```
A0: begin
      shared int x = 0;
      shared int lk = 0;
      int n = 2;
A1: task_create(n)

task 0                              task 1
      int j = 0;                          int sum = 0;
      int k = 0;                          int index = 0;
                                          int i = 0;
B0: lock(lk);                       C0: lock(lk);
B1: x = x + 1;                      C1: x = x + 1;
B2: j = x;                          C2: index = x;
B3: unlock(lk);                     C3: unlock(lk);
B4: if j == 2 then                  C4: for i = 1 to index do
B5:     j = j - 1;                  C5:     sum = sum + i;
B6: else
B7:     k = k + 1;

A2: task_terminate(n);
A3: end;
```

Figure 1a. A parallel program.

first execution (and represents an insertion of an additional loop iteration). The instances of the conditional statement B4 correspond in each execution (4) however B5 and B6-B7 do not correspond but must be compared when calculating a measure (5). We will return to Figure 1d shortly.

There exist other possible choices for correspondence between executions. We contend that our choice is a good choice in that it is the best correspondence that can be made given the time and space constraints. We justify our choice of correspondence in Section 8.

Execution 1	Execution 2
A0	A0
A1	A1
C0	B0
B0	C0
C1	B1
C2	B2
C3	B3
B1	C1
B2	C2
B3	C3
C4	C4
C5	C5
B4	B4
B5	B6
C4	B7
A2	C4
A3	C5
	C4
	A2
	A3

Figure 1b. Two possible executions.

	Execution 1	Execution 2
(1)	A0	A0
	A1	A1
(2)	C0	B0
	B0	C0
	C1	B1
	C2	B2
	C3	B3
	B1	C1
	B2	C2
	B3	C3
(3)	C4	C4
	C5	C5
(4)	B4	B4
	B5	B6
(5)		B7
(6)	C4	C4
(7)		C5
		C4
(8)	A2	A2
	A3	A3

Figure 1c. Correpondence of executions.

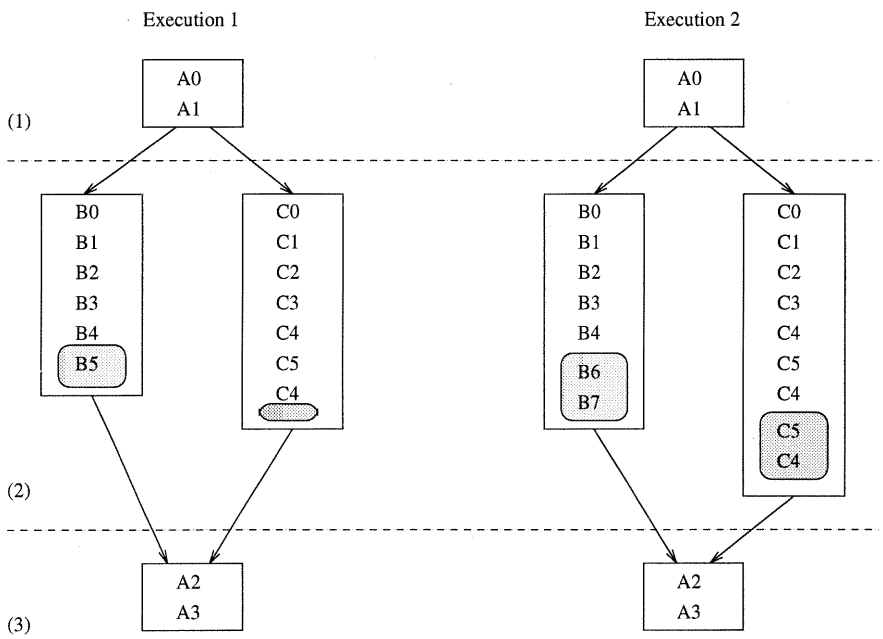


Figure 1d. The graph of the execution given in Figure 1b.

4. Preliminaries

Assumptions

It is necessary to make a few assumptions about programs in order to guarantee the correctness of our algorithm. Assumptions A1 and A2 ensure that programs are correct and that they are structured. The former ensures that programs halt and that we do not have to worry about syntactic difficulties. The latter ensures that programs do not contain **goto** statements (i.e., statements have a single entry and exit point). Assumption A3 allows us to identify parts of an execution as *disjoint* without having to determine if they have instructions in common (even though they represent executions of different program statements). This implies, for example, that two load instructions on the same variable are not equivalent unless they are instances resulting from the *same* program statement. While A3 may not be a reasonable assumption in general, for the purpose of identifying sections of code that are effectively disjoint and need further comparison, the assumption is reasonable.

A1: All programs are syntactically correct and halt on any possible input data.

A2: Programs are properly structured.

A3: Two instruction instances are equivalent only if they come from the same program statement.

We assume that the execution architecture hardware is a shared memory multiprocessor with sequential memory consistency[3] [7] (e.g., the Convex SPP and the KSR-2). This restriction rules out nondeterminism due to memory references that are not caused by program nondeterminism.

Our analysis depends on program source. We assume a canonical form of the program in a simple procedural language modified to include **task_create** and **task_terminate** constructs, and **lock** and **unlock** statements for concurrency. This language uses arrays, local and shared variables. All standard arithmetic operations are available as model, we make no assumptions about function behavior. Any program must have at most one shared variable or array reference per statement, and hence at most one point in a statement that can effect the execution order. For a detailed discussion of this language see [3].

A program is any sequence of statements with an initial **begin** statement and a final **end** statement. For every **task_create** statement, there is a corresponding **task_terminate** statement. For every **lock** statement, there is a corresponding **unlock** statement. A *program event* is an execution instance of one or

3 In a sequentially consistent multiprocessor each processor issues memory requests in the order specified by the executing program and requests from the set of processors are serviced in the order in which they are received (first-in-first-out).

more consecutively executed intermediate code statements (or instructions). Given a fixed data set, each program has a set of possible executions.

We use Lamport's theory of concurrent systems [8]. Each event conceptually has a start and finish time. The relation $a \rightarrow_T b$ implies that event a executes *and finishes* before event b begins. Therefore, a can causally affect b, but b cannot affect a. Whenever two events execute *concurrently*, we have $\neg(a \rightarrow_T b) \wedge \neg(b \rightarrow_T a)$. That is, neither a nor b necessarily completes before the other begins — their executions overlap. A *program execution*, P, is a triple $P=<E, \rightarrow_T, \rightarrow_{sd}>$where E is a finite set of events, $\rightarrow_T$ is the temporal ordering relation over E, and $\rightarrow_{sd}$ is the shared data dependence relation also defined over E such that $\rightarrow_T$ and $\rightarrow_{sd}$ are irreflexive partial ordering relations both of which satisfy the semantics of the language. For a detailed discussion of the program execution axioms see [3].

Given the intermediate code language, it is possible that more than one *thread* of execution exists at a given time. We call these threads of execution *processes* and denote the set of events for a particular process p with E_p. We use the notation **task_create**(E) to denote the set of **task_create** instruction instances in E (we use similar notion for **task_terminate** instruction instances).

Graph of an Execution

For the purposes of comparing executions, we propose simplifying a program execution by removing parallel thread interaction. Consider the example in Figures 1a-d. For the two executions in Figure 1b we can remove parallel thread interaction resulting the comparison in Figure 1d. Here, The processes associated with tasks 1 and 2 in each execution are corresponded (2) and the main thread of execution (1 and 3) are corresponded. In this example, statements B0-B4 correspond in each execution because each execution executes these statements with the same outcome. The first difference appears after the condition at B4 branches true or false. The highlighted box signifies that statements B5 in execution 1 and B6 and B7 in execution 2 are different and must be measured for difference. Similarly, the dark highlighted box in execution 2 with corresponding empty highlighted box in execution 1, signifies that there are no corresponding statements for this extra iteration. Notice that information about interleaving is irrelevant.

Intuitively removing parallel thread interaction from an execution accentuates the structure of the execution. Thread interaction serves only as a record of the *cause* (an interleaving of instruction instances) of a change in the execution path. Since we are interested only in the actual changes that occur from one execution to another, this record of the causes of change is not important. Removing all parallel

thread interaction, from an execution, results in the *graph*[4] of an execution.

Definition 1: Given a program execution $P=<E,\rightarrow_T,\rightarrow_{sd}>$ the *graph* of P, $P_g=<E,\rightarrow_{T_g},\rightarrow_{sd_g}>$ is the result of the following transformation on P:

For each relation $\rightarrow_T$ and $\rightarrow_{sd}$ perform the following resulting in $\rightarrow_{T_g}$ and $\rightarrow_{sd_g}$ respectively,

for each $a,b \in E$-(**task_terminate**$(E)\cup$**task_create**(E)) such that $a \rightarrow_T b$, if $a \in P_i(E)$ and $b \in P_j(E)$ and $i \neq j$, remove (a,b) from $\rightarrow_T$ and if $a \rightarrow_{sd} b$ then remove (a,b) from $\rightarrow_{sd}$.

■

Definition 1 simply removes all members of the temporal and shared data dependence relations that relate events from different threads of execution. For the remainder of this chapter, we use the term execution to refer to the graph of an execution.

Paths in an Execution

We define the notion of a *path* in a program execution. Paths will play an important role in formal discussions of executions. Intuitively, a path represents as subset of an execution that is contiguous in time (i.e., a subsequence of an execution). For example, execution 1 in Figure 1d contains the a path from C0 to C2.

4 We call this transformed execution the graph of an execution because it accentuates the structure of an execution, which is a graphical structure.

Definition 2: Given a program execution $P=<E,\rightarrow_T,\rightarrow_{sd}>$ a path of P is a triple $p(h,t)=<E_{(h,t)},\rightarrow_{T_{(h,t)}},\rightarrow_{sd_{(h,t)}}>$ such that

1. $E_{(h,t)}$ is a non empty subset of E,
2. $\rightarrow_{T_{(h,t)}}$ is a subset of $\rightarrow_T$,
3. and $\rightarrow_{sd_{(h,t)}}$ is a subset of $\rightarrow_{sd}$.

such that the following are true for any $a,b \in E_{(h,t)}$,

4. $a \rightarrow_T b$ if and only if $a \rightarrow_{T_{(h,t)}} b$ and $a \rightarrow_{sd} b$ if and only if $a \rightarrow_{sd_{(h,t)}} b$,
5. and $a \rightarrow_{T_{(h,t)}} c$ and $c \rightarrow_{T_{(h,t)}} b$ if and only if $c \in E_{(h,t)}$,

with the following conditions on h and t,

6. $h \in E_{(h,t)}$ and there does not exist $a \in E_{(h,t)}$ such that $a \rightarrow_{T_{(h,t)}} h$,
7. and $t \in E_{(h,t)}$ and there does not exist $a \in E_{(h,t)}$ such that $t \rightarrow_{T_{(h,t)}} a$.

■

h and *t* are the *head* and *tail* respectively of the path. A *prefix* of an execution is any valid path such that *h* is the initial instruction for the program. Likewise, a *suffix* of a program execution is any valid path such that *t* is the final instruction for the program. In Figure 1d, execution 1 has the path p(A0,B1) which is a prefix and the path p(C2,A3) which is a suffix.

We define two paths to be equivalent whenever they are the *same* path and are disjoint whenever they have no equivalent instruction instances in common (where equivalence for instruction instances is defined by assumption A3). In Figure 1d, the paths p(B0,B4) are equivalent in both executions. The paths p(B0,A2) in execution 1 and p(B0,A2) in execution 2 are not equivalent and the paths p(B1,B5) in executions 1 and p(B6,B7) in execution 2 are disjoint.

Definition 3: Two paths $p(h_1,t_1)=<E_{(h_1,t_1)},\rightarrow_{T_{(h_1,t_1)}},\rightarrow_{sd_{(h_1,t_1)}}>$ and $p(h_2,t_2)=<E_{(h_2,t_2)},\rightarrow_{T_{(h_2,t_2)}},\rightarrow_{sd_{(h_2,t_2)}}>$

1) are *equivalent*, denoted $p(h_1,t_1) = p(h_2,t_2)$ whenever

$E_{(h_1,t_1)} = E_{(h_2,t_2)}$,

$\rightarrow_{T_{(h_1,t_1)}} = \rightarrow_{T_{(h_2,t_2)}}$,

and $\rightarrow_{sd_{(h_1,t_1)}} = \rightarrow_{sd_{(h_2,t_2)}}$.

2) are *disjoint*, whenever $E_{(h_1,t_1)} \cap E_{(h_2,t_2)} = \varnothing$.

■

The *successor* of an instruction instance e in an execution is the instruction instance that immediately follows e in time (in the execution). Likewise, the *predecessor* is an instruction instance e is the instruction instance that immediately preceded the instruction instance in time (in the execution). In Figure 1d, the successor of instruction instance B5 in execution 1 is instruction instance A2 and the predecessor of instruction instance B5 is instruction instance B4.

Definition 4: Given an execution $P=<E,\rightarrow_T,\rightarrow_{sd}>$ for any instruction instance e,

1) the *successor* of e is an instruction instance s such that there exists a path $p(e,s)=<E_{(e,s)},\rightarrow_{T_{(e,s)}},\rightarrow_{sd_{(e,s)}}>$ such that $E_{(e,s)} = \{e,s\}$.

2) the *predecessor* of e is an instruction instance p such that there exists a path $p(p,e)=<E_{(p,e)},\rightarrow_{T_{(p,e)}},\rightarrow_{sd_{(p,e)}}>$ such that $E_{(e,s)} = \{p,e\}$.

■

5. Corresponding Executions

Convergence and Divergence in Executions

Our goal is to correspond executions. That is, we want to *pin* two executions in such a way as to identify their differences. Executions have periods of *convergence*, at which they are equivalent (i.e., they execute the same code), and of *divergence*, at which they are disjoint (i.e., they execute completely different sections of code). Returning to the example of Figure 1d, the highlighted areas represent periods of divergence between the two example executions. Execution 1 executes statement B5 while execution 2 executes statements B6 and B7, and execution 2 executes a second iteration of the loop while execution 1 executes no additional statements. We formally define divergence and convergence for executions in

Definition 5.

Definition 5: Given two program executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ with two fixed but arbitrary paths $p(h_1,t_1)=<E_{(h_1,t_1)},\rightarrow_{T_{(h_1,t_1)}},\rightarrow_{sd_{(h_1,t_1)}}>$ and $p(h_2,t_2)=<E_{(h_2,t_2)},\rightarrow_{T_{(h_2,t_2)}},\rightarrow_{sd_{(h_2,t_2)}}>$ (respectively),

1) there is a *divergence point instance* d if $p(h_1,t_1) \neq p(h_2,t_2)$, the predecessor of t_1 is c_1, the predecessor of t_2 is c_2, the paths $p(h_1,c_1)$ and $p(h_2,c_2)$ are equivalent, and $c_1=c_2=d$.

2) there is a *convergence point instance* d if $p(h_1,t_1) \neq p(h_2,t_2)$ and they are not disjoint, the predecessor of t_1 is c_1, the predecessor of t_2 is c_2, the paths $p(h_1,c_1)$ and $p(h_2,c_2)$ are disjoint, and $t_1=t_2=d$.

■

Intuitively a divergence point occurs whenever two paths are equivalent up to some instruction instance and then execute a different instruction instances or have a different outcome on the same instruction instance. A convergence point occurs whenever two paths are disjoint, or completely different, up to some instruction instance and then execute an instance of the same instruction with the same outcome.

The notions of divergence and convergence in executions are critical to our ability to correspond executions with reasonable time and space usage. Divergence points for executions of parallel programs were first characterized in [6] and further studied in [3] [4] and are called *trace change points*. A TCP is any program statement or instruction whose outcome is dependent on the value of a shared variable, either directly or indirectly. Notice that statement B4 in Figure 1a is dependent on the value of j which is dependent on the value of shared variable x making statement B4 a TCP. Similarly, statements B0, C0, and C4 are TCPs. TCPs consist of program conditional, iterative, **lock**, array reference, and pointer indexing statements whose outcome depends on the value of a shared variable. In addition, the program **begin** statement and any **task_create** statements are TCPs.

The problem of detecting a TCP instance in an execution is actually that of locating the TCP instance and determining if divergence has occurred. We use the notation $outcome(P,s)$ to denote the outcome of instruction instance s in execution P. Notice that array and pointer reference TCPs behave differently than the conditional (and iterative) TCPs. This is because an instruction instance can cause divergence by having a different outcome *and* for the same instruction instance. While conditional (and

iterative) TCPs will diverge due to executing a *different* instruction instance after a comparison is made. Not surprisingly, conditional TCPs can be easily located and their outcome easily determined. Array and pointer references pose a problem and our solution is to conceptually view such a TCP as a pair of instructions: a **no-op** instruction marking the TCP, and the actual reference instruction. Even though this marker does not exist in any real executions, we assume that it does. This assumption guarantees that a period of divergence is caused by a single TCP and that convergence will occur before the next period of divergence can begin. In Section 3, we show how this marker manifests itself in our algorithm.

We use the notation $TCP(E)$ to denote the set of TCP instances for an event set E. It will be necessary to refer to the sequence of TCPs executed by a program during a particular execution. We use the notion of the *TCP successor* of a TCP instance and *TCP predecessor* of a TCP instance in referring to this sequence. The TCP successor (and predecessor) of a TCP instance is the next (previous) TCP instance that occurs in the graph of a program execution. More formally,

Definition 6: Given a program execution $P=<E,\rightarrow_T,\rightarrow_{sd}>$ for each instruction instance $a \in TCP(E)$ there exist $s,d \in TCP(E)$, the TCP successor and TCP predecessor of a respectively, such that

Either a is the initial statement or there exists a path $p(d,a)=<E_{(d,a)},\rightarrow_{T_{(d,a)}},\rightarrow_{sd_{(d,a)}}>$ such $(E_{(d,a)}-\{d,a\})\cap TCP(E)=\varnothing$ and

Either a is the final statement or there exists a path $p(a,s)=<E_{(a,s)},\rightarrow_{T_{(a,s)}},\rightarrow_{sd_{(a,s)}}>$ such $(E_{(a,s)}-\{a,s\})\cap TCP(E)=\varnothing$ and

■

We extend the work of [6] and [3] by defining the set of convergence points (CPs) for a parallel program execution. A CP is defined to be the *first* instruction or statement that any program execution *must* execute after a period of divergence. Lemma 7 below demonstrates the convergence points exist and shows that convergence points are continuations of either conditional statements, iterative statements, **lock** statements, or array and pointer reference statements. We use the notation $CP(E)$ to denote the set of CP instances in an event set E. We use the notation $TCP(P)$ and $CP(P)$ to denote the set of TCPs and CPs (respectively) for program P.

Lemma 7: Any program P with reduced TCP set T = $TCP(P)$ - {**task_create, begin**} has a set of convergence points (as defined in Definition 6), denoted $CP(P)$ each of which is either the continuation of an iterative, conditional, **lock**, or array reference statement.

■

Proof: With the **task_terminate** and *end* statements removed from the set of TCPs of a program, the set of TCPs are either array reference, iterative, **lock**, or conditional statements. Hence, it suffices to show that there is a CP for each. There are four cases:

1. Without loss of generality, consider the TCP containing statement a[i] = c. Any execution instance of this statement will end with an instance of the instruction **ld** @a+i which represents a possible point of divergence for any two executions. Any execution in which the statement is executed must execute the continuation of the array reference statement (the statement immediately following it) immediately following the **ld** instruction. If this statement is not itself an array reference TCP, then is a CP. A special case occurs in the event that the continuation of an array reference TCP is itself an array reference TCP. Since the program cannot have an infinite string of array references that are TCPs, the CP for each TCP in this sequence is the first statement in the sequence that is not and array reference TCP.

2. Without loss of generality, consider an TCP containing statement

```
if (c) then
   a
else
   b
d
```

Any two executions in which this statement was executed must have the branch on *c* in common and therefore be subject to a possible divergence. Any two execution where the outcome of *c* is different diverge (since *a* and *b* are disjoint by Assumption 3) must execute *d* immediately following the execution of either *a* or *b* since by Assumption 1, the program is structured. Therefore the continuation of a conditional statement, *d*, is a CP. Once again, there is a special case whenever the continuation for a conditional statement is an array reference TCP. In this case, the continuation for the conditional statement is the continuation of the array reference TCP (see Case 1).

3. Because a **lock** statement does not change the program flow of control, it follows from Case 1 that each **lock** statement has a CP.

4. Because an iterative statement a repetitive branching construct, Case 3 follows from Case 2.

□

The intuition for this proof is that given the assumption that programs are structured, any statement (conditional, iterative, **lock**, or referencing) has single entry and exit point. Therefore each statement has a single continuation and this point is a CP if the statement was itself a TCP.

For structuring purposes, we add the program **end** statement and all **task_terminate** statements to the set of CPs for a program.

Definition 8: The **end** statement and any **task_terminate** are members of the set of convergence points for a program. These convergence points correspond to the **begin** statement TCP and to the appropriate **task_create** statement (respectively).

■

Given Lemma 7 and Definition 8, it follows that there exists a correspondence between program TCPs and CPs such that for every program TCP, there exists a corresponding program CP.

Corollary 9: For any program P, there exists a correspondence between the sets $TCP(P)$ and $CP(P)$ and this correspondence is an onto mapping.

■

Notice that we cannot claim that this mapping is one-to-one since some TCPs will share the same CPs. For example, nested conditional statements will share the same TCP. Notice that it is also possible that a CP is itself a TCP.

While Lemma 7 and Definition 8 identify the correspondence between program TCP and CP pairs, it will be necessary to locate these instance pairs in an execution. In particular, we want to find the *correct* CP instance for every TCP instance. Consider the example of a conditional statement containing a TCP. If the TCP is the conditional instruction itself, then we do not want to match, or *correspond*, an instance of this instruction with an instance of its CP that is a result of a later execution of the conditional

statement. Simply put, we want to match the correct TCP and CP instances. Lemma 10 demonstrates that there exists a path between a TCP instance and its corresponding CP instance and that there are no further instances of either the CP or TCP on the path. This is a direct result of assumption 2.

Lemma 10: Given a program execution $P=<E,\rightarrow_T,\rightarrow_{sd}>$ and given an $s \in TCP(E)$ there exists a $c \in CP(E)$ if s and c correspond, then there exists a path $p(s,c)=<E_{(s,c)},\rightarrow_{T_{(s,c)}},\rightarrow_{sd_{(s,c)}}>$ and there does not exist $s' \in TCP(E_{(s,c)})$ such that s' and s are instances of the same program TCP nor does there exist $c' \in TCP(E_{(s,c)})$ such that c' and c are instances of the same program CP.

■

Proof: The lemma is trivially true for the program **begin** and **end** TCP and CP pair (since there is only a single instance of each). We demonstrate that such a path exists for the remaining TCP and CP pairs and that the path is the shortest path between a TCP instance, call this instance s, and an instance of its corresponding CP, call this instance c. Let $p(s,c)=<E_{(s,c)},\rightarrow_{T_{(s,c)}},\rightarrow_{sd_{(s,c)}}>$ be this path. Suppose that the lemma is not true. Then there must be either an instance of the TCP that s is an instance of, call this instance s', or an instance of the CP that c is an instance of, call this instance c', or both on the path from s to c. Clearly if c' exists and $c' \in E_{(s,c)}$ then $p(s,c)$ is not the shortest path from s to an instance of its CP ($p(s,c')$ is the shortest path). Consider that $s' \in E_{(s,c)}$. If this is the case then the program in question violates Assumption 2 since there is no way to reach s' before c except through the use of a **goto** statement. Hence if $p(s,c)$ is the shortest path from s to an instance of its corresponding CP, s' cannot be on the path. Finally, that c is the corresponding CP instance for TCP instance s follows from Assumption 2.

□

Notice that the Lemma implies not only that we can determine the correct TCP and CP instance correspondence in an execution, but also that once a TCP instance is located, it is a simple matter to locate its matching CP instance. In fact, it is simply a matter of scanning the execution for the next instance of the CP for this TCP. Corollary 11 follows directly from Lemma 10 and Corollary 9.

Corollary 11: For any execution of a program there is a correspondence between the TCP instances and CP instances and this correspondence is an onto mapping.

■

Corollary 11 follows from Lemma 10.

The Behavior of Executions

Thus far, we defined TCP and CPs in parallel programs. Instances of these points represent divergent and convergent points in actual executions. We have demonstrated that a logical correspondence between TCP and CP exists and that corresponding instance pairs can be easily located. Next we turn our attention to the behavior of convergence and divergence in executions. First, in Lemma 12, we demonstrate that whenever two executions have a TCP instance in common and the outcome of those instances are the same, then the paths (in each execution) from the TCP instance to its TCP successor are equivalent.

Lemma 12: Given two program executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ for any TCP instance $a \in TCP(E_1) \cap TCP(E_2)$ with TCP successor $s_1 \in TCP(E_1)$ and $s_2 \in TCP(E_2)$ respectively. If $outcome(P_1,a)=outcome(P_2,a)$ then $p(a,s_1)=p(a,s_2)$

■

Proof: There are no TCPs other than a and s in either path since if there is an embedded TCP, s is not the successor TCP for a. Given this, there can be no change in execution path from a to s in either execution. Hence $outcome(P_1,a)=outcome(P_2,a)$ implies that $p(a,s_1)=p(a,s_2)$.

□

To see how Lemma 12 works, consider the example of Figure 1d. The paths P(B0,B4) in each execution are an example of equivalent paths where the head of the path, B0, is a TCP which has the same outcome in each execution, and the tail of the path is the TCP successor of B0, B4.

A similar result for divergence shows that if two executions have a TCP instance in common and the outcome of the instances are different in each execution, then the path from the TCP instance to the corresponding CP instance (in each execution) will be disjoint with the exception of the CP and TCP instances themselves.

Lemma 13: Given two program executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ for any TCP instance $a \in TCP(E_1) \cap TCP(E_2)$ and corresponding CP $c_1 \in E_1$ and $c_2 \in E_2$ respectively, if $outcome(P_1,a) \neq outcome(P_2,a)$ then for $p(a,c_1)$ and $p(a,c_2)$, $E_{(a,c_1)} \cap E_{(a,c_2)}=a,c$ where $c_1=c_2=c$.

■

Proof: Assume that the lemma is false. First it follows from Lemma 10 that $c_1=c_2$ (they are instances of the same instruction). It must be the case that there is another CP on paths $p(a,c_1)$ and $p(a,c_2)$ that both paths have in common. Then by Lemma 10, c is not the corresponding CP for TCP a.

□

To see how Lemma 13 works, the path p(B4,A2) in each execution (in the example of Figure 1d) is an example of a path that begins with a TCP, for which the outcome is different in each execution, and ends with a convergence point (A2). Notice that if we remove the head and tail of this path in each execution we have the paths p(B5,B5) in execution 1 and p(B6,B7) in execution 2 which are disjoint.

Finally it follows that whenever two executions have a CP instance in common, then the paths from that CP to the next TCP in each execution are equivalent.

Corollary 14: Given two executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ if there exists a path in P_1, $p_1(h,t)=<E_{(h,t)_1},\rightarrow_{T_{(h,t)_1}},\rightarrow_{sd_{(h,t)_1}}>$ and a path in P_2, $p_2(h,t)=<E_{(h,t)_2},\rightarrow_{T_{(h,t)_2}},\rightarrow_{sd_{(h,t)_2}}>$ and there does not exist an $i_1 \in E_{(h,t)_1} \cap E_{(h,t)_2}$ such that if $h \rightarrow_{T_{(h,t)_1}} i$ and $ih \rightarrow_{T_{(h,t)_1}} t$ or $h \rightarrow_{T_{(h,t)_2}} i$ and $ih \rightarrow_{T_{(h,t)_2}} t$ then $p_1(h,t)=p_2(h,t)$.

■

We now have the two critical pieces for corresponding executions. First, we know that TCP and CP instance pairs exist and that they can be located. This implies among other things that we can traverse an execution and know "where we are" at all times. Second, we know that the behavior of executions is such that they will exhibit periods of convergence (during which they are equivalent) and divergence (during which they are disjoint) and nothing more. Given this, it remains to be shown that we can traverse two executions making the correct correspondence of TCP instances and CP instances between the executions (i.e., that we can *pin* the executions together).

Correspondence

Before any results about corresponding executions can be proved, we need a formal notion of correspondence. Our goal is to use correspondence to efficiently and correctly compare executions. Therefore, any reasonable correspondence must adhere to the following three rules. First, Corollary 11 implies that any reasonable correspondence will be an onto mapping between TCP and CP instances. Second, whenever two event instances are corresponded, they should be instances of the same instruction. Finally, a correspondence must obey the temporal and shared data dependence relations for the execution.

Definition 15: Given two executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ (from the same program) with common TCP and CP sets $s_{TCP}=TCP(E_1)\cap TCP(E_2)$ and $s_{CP}=CP(E_1)\cap CP(E_2)$. A *correspondence* of those executions $M(P_1,P_2)$, is a mapping from $s_{TCP}\cup s_{CP}$ to $s_{TCP}\cup s_{CP}$ such that:

1) M is onto.

2) whenever M maps one instruction instance to another, these instructions are an instance of the same instruction.

3) if M maps $a\in E_1$ to $b\in E_2$ and $c\in E_1$ to $d\in E_2$ then $a\rightarrow_{T_1}c$ implies that $b\rightarrow_{T_2}d$ and $a\rightarrow_{sd_1}c$ implies that $b\rightarrow_{sd_2}d$.

■

The Theorem of Correspondence

Given Definition 15, the Theorem of Correspondence (below) demonstrates that a correspondence between executions adhering to Definition 15 does exist. Lemma 16 is the base case for the Theorem of Correspondence. The base case consists of proving that two arbitrary executions can be corresponded at the first level of nesting of **task_create/task_terminate** statements. Simply put, we can correspond iterative statements, array and pointer references, and conditional statements, but we can only pin the outside of **task_create/task_terminate** statements (i.e., we cannot pin the contents of these statements).

Lemma 16: Given any two executions of the same program there exists a correspondence **M** of the executions up to but not including the body of **task_create/task_terminate** statement instances.

■

Proof: We prove Lemma 16 by induction on the number of TCP instances that two executions have in common.

Base Case: Every execution has the begin statement in common. If the program contains no other TCPs, then by Lemmas 10 and 12 it is guaranteed that the end statement will be located and that the correspondence includes only the begin and end statements from each execution. Clearly this correspondence fulfills conditions 1-3 of Definition 15.

Induction Hypothesis: Given fixed but arbitrary executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$, assume that there exists a correspondence $\mathbf{M_k}$ corresponding prefixes $p_1(begin,s_k)=<E_{p1},\rightarrow_{T_{p1}},\rightarrow_{sd_{p1}}>$ and $p_2(begin,s_k)=<E_{p2},\rightarrow_{T_{p2}},\rightarrow_{sd_{p2}}>$ and given $\mathbf{S_k}=(TCP(E_{p1})\cap TCP(E_{p2}))\cup(CP(E_{p1})\cap CP(E_{p2}))$, $s_k\in\mathbf{S_k}$ and $|\mathbf{S_k}|(bv=k)$.

Induction Step: There are three possible cases,

1. $s_k\in(CP(E_{p1})\cap CP(E_{p2}))$. There are two cases. First, if s_k is the instance of the program **end** statement, the $\mathbf{M_k}$ corresponds the two executions P_1 and P_2. Otherwise, given the TCP successor of s_k, call these TCPs $s_{1,k+1}$ and $s_{2,k+1}$ respectively, Lemma 14 demonstrates that there exist equivalent paths $p_1(s_k,s_{1,k+1})$ and $p_2(s_k,s_{2,k+1})$ and that $s_{1,k+1}=s_{2,k+1}$. This implies that there exists correspondence $\mathbf{M_{k+1}}=\mathbf{M_k}\cup s_{1,k+1},s_{2,k+1}$ corresponding prefixes $p_1(begin,s_{1,k+1})$ and $p_2(begin,s_{2,k+1})$.

2. $s_k\in(TCP(E_{p1})\cap TCP(E_{p2}))$ and is an instance of a **task_create** statement. Recall that **M** is a correspondence that corresponds E_1 and E_2 but does not correspond the body of **task_create/task_terminate** statements. It is reasonable then to assume that the paths from s_k to its corresponding CP, call these events $c_{1,k+1}$ and $c_{2,k+1}$ respectively, in each executions are disjoint (excluding s_k and c_{k+1} from the paths). Lemma 10 ensures that $c_{1,k+1}$ and $c_{2,k+1}$ can be located and that $c_{1,k+1}=c_{2,k+1}$. Therefore we have correspondence $\mathbf{M_{k+1}}=\mathbf{M_k}\cup c_{1,k+1},c_{2,k+1}$ corresponding prefixes $p_1(begin,c_{1,k+1})$ and $p_2(begin,s_{2,k+1})$.

3. $s_k \in (TCP(E_{p1}) \cap TCP(E_{p2}))$ (but is not an instance of a **task_create** statement). There are two possible cases depending upon the outcome of s_k in each execution. First, consider the case where Outcome(P_1,s_k) = Outcome(P_2,s_k). Then given the TCP successor of s_k, call these TCPs $s_{1,k+1}$ and $s_{2,k+1}$ respectively, Lemma 12 demonstrates that there exist equivalent paths $p_1(s_k, s_{1,k+1})$ and $p_2(s_k, s_{2,k+1})$ and that $s_{1,k+1}=s_{2,k+1}$. This implies that there exists correspondence $\mathbf{M_{k+1}}=\mathbf{M_k} \cup s_{1,k+1}, s_{2,k+1}$ corresponding prefixes $p_1(begin, s_{1,k+1})$ and $p_2(begin, s_{2,k+1})$. Next, consider the case where Outcome(P_1,s_k) ≠ Outcome(P_2,s_k). Then by Lemma 13 there exists $c_{k+1} \in (CP(E_1) \cap CP(E_2)$ and that given the predecessor of c_{k+1} in E_1, call this event c_{1p}, the predecessor of c_{k+1} in E_2, call this event c_{1p}, the successor of s_k in E_1, call this event s_{1p}, and the successor of s_k in E_2, call this event s_{2p}, then there exist paths $p_1(s_{1p}, c_{1p})$ and $p_2(s_{2p}, c_{2p})$ that are disjoint. Since there can be no correspondence between disjoint paths we have correspondence $\mathbf{M_{k+1}}=\mathbf{M_k} \cup s_{1,k+1}, s_{2,k+1}$ corresponding prefixes $p_1(begin, s_{1,k+1})$ and $p_2(begin, s_{2,k+1})$.

Given Assumption 1, there exists a correspondence **M** between any two program executions (of the same program on the same data set) that corresponds the executions up to the body of **task_create/task_terminate** statement instances.

□

The Theorem of Correspondence is proved using induction on the nesting level of **task_create/task_terminate** statements using Lemma 16 as the base case.

Theorem of Correspondence: Given any two executions of the same program there exists a correspondence **M** of the executions.

■

Proof: We prove the theorem by induction on the **task_create/task_terminate** nesting level of the executions.

Base Case: The basis for the induction is Lemma 16.

Induction Hypothesis: Given fixed but arbitrary executions $P_1=<E_1, \rightarrow_{T_1}, \rightarrow_{sd_1}>$ and $P_2=<E_2, \rightarrow_{T_2}, \rightarrow_{sd_2}>$ with common TCP and CP instances $\mathbf{S}=(TCP(E_1) \cap TCP(E_2)) \cup (CP(E_1 \cap CP(E_2))$. For the induction hypothesis assume that there

exists $\mathbf{M_k}$ corresponding P_1 and P_2 up to but not including the body of **task_create/task_terminate** instances at nesting level k.

Induction Step: Without loss of generality, choose a **task_create/task_terminate** instance pair $s_{1,k} \in \mathbf{S} \cap E_1$ and $c_{1,k} \in \mathbf{S} \cap E_1$ such that $s_{1,k}$ is a **task_create** instance at nesting level k and $c_{1,k}$ the correspoinding **task_terminate** instance at nesting level k. Then by the induction hypothesis there exist $s_{2,k} \in \mathbf{S} \cap E_2$ and $c_{2,k} \in \mathbf{S} \cap E_2$ such that $(s_{1,k}, s_{2,k}) \in \mathbf{M_k}$ and $(c_{1,k}, c_{2,k}) \in \mathbf{M_k}$. There exist paths $p_1(s_{1,k}, c_{1,k})$ and $p_2(s_{2,k}, c_{2,k})$ which contain n threads between which there exists a one-to-one mapping. Lemma 16 implies that for each thread there is a correspondence up to nesting level one, or $\bigcup_{i=1}^{n} \mathbf{m_i}$. This implies that the correspondence defined by $\mathbf{M_k} \cup (\bigcup_{i=1}^{n} \mathbf{m_i})$ corresponds E_1 and E_2 up to **task_create/task_terminate** instances at nesting level $k+1$. Therefore, there exists a correspondence $\mathbf{M}$ between any two executions of the same program (on the same data sets).

□

6. An Algorithm for Correspondencing Executions

In this section we present an algorithm for corresponding two program executions based upon the Theorem of Correspondence. The algorithm takes as input a parallel program with its TCP and CP sets already computed and two executions from that program. The algorithm uses a preprocessing phase and a correspondence phase. In the preprocessing phase, the program is transformed into an intermediate form that will, during the comparison phase, serve to direct the traversal of the executions. In this section, we present the correspondence algorithm in the form of a function that returns a correspondence. Naturally, the correspondence itself can require a large amount of space for storage. In reality, the correspondence algorithm should be coupled with an algorithm computing some measure of difference for the executions and the correspondence should not be stored. We discuss this coupling in the next section.

6.1. The Preprocessing Phase

Computing the set of program TCP and CPs

The problem of computing TCPs is one that requires static program analysis to determine possible program behavior during runtime. Like may such problems, it is undecidable. In [2] we outline a global analysis algorithm for approximating TCPs based on a constant propagation algorithm in [10]. In addition, we present a simple graphical representation of iterative statements that yields an asymptotically

faster algorithm. Both algorithms provide a *conservative* estimate of the set of program TCPs (i.e., a set is computed that is guaranteed to contain all possible program TCPs but which might also contain some non-TCP statements).

The size of input to the two algorithms is given by N, E, and V. Given a program flow graph, N is the number of nodes and E the number of edges. V is the number of program variables. The global analysis algorithm has runtime $O(N \times E \times V^2)$ and $O(N \times V)$ space usage. The faster algorithm has an $O(N \times V)$ runtime and $O(N+V)$ space usage.

Given the set of TCPs for a program, computing the set of CPs can be accomplished with a simple linear time stack based algorithm at compile time. Either of the algorithms for computing TCPs can be modified to collect CPs while locating TCPs with no change to their time bounds. Both of these sets are fixed for any program and need only be calculated once.

Computation of both of these sets is out of the scope of this paper. The reader is referred to [2] for algorithms to compute these sets.

The TCP Basic Block Flow Graph

Holliday and Ellis first introduced a variant of the basic block flow graph intermediate form for a program, the *address basic block flow graph* (ABB flow graph), based upon TCPs [6]. In the preprocessing phase of the algorithm the program, together with its TCP and CP sets, is transformed into the *TCP basic block flow graph* (TBB flow graph) of the program. The TBB flow graph is itself a variation of the ABB flow graph.

Recall that a basic block is a section of code that has a single entry and exit point. Basic blocks are groupings of logically indivisible statements. Basic blocks can be used to form a basic block flow graph of a program. A *TCP basic block* (TBB) represents a section of code that produces a logically indivisible *execution* outcome. For example, a basic block would have embedded array references and a TBB would have a single array reference (that is a TCP) per block. The TBB flow graph is a flow graph in which the nodes of the graph are the TBBs of the program. The conditions for delimiting a TBB[5] are:

1) Each block contains at most a single TCP.

2) Each block begins either with a CP or the target of a branch statement. The exception is the

5 There are several differences between our TBB flow graph and the ABB flow graph of introduced by Holliday and Ellis [6]. Most notable is the addition of CP information *and* CPs as delimiters for ABBs as well as the requirement that a single TCP occupy each block.

first block containing the begin statement.

3) Each block ends with an TCP, a branching instruction whose target is a CP, or an instruction immediately preceding a CP. The exception is the last block containing the end statement.

In addition, we add a **no-op** instruction to each block containing an array reference or pointer reference TCP immediately preceding the reference instruction.

The TBB flow graph will serve to direct the rapid traversal of the executions being compared. In order to facilitate this process, each TBB is decorated with additional information. Associated with each block is an instruction count for the block (not including any **no-op** instructions), a flag denoting whether or not the block contains a TCP, a flag denoting whether or not the block contains a CP, and a pointer to the block containing the CP corresponding to the TCP in this block (if there is a TCP in this block).

This information is designed to facilitate rapid traversal of groups instruction instances that are instances of a particular TBB. That is, consider a TBB b with instruction count n, and consider an instance of block b in an execution. If it has been determined that the information in the block instance is uninteresting, then it can be rapidly traversed by moving forward n instruction instances in the execution or if the block needs to be traversed to the end, then n-1 instruction instances are traversed. Given Lemma 12, whenever two equivalent TCP instances with the same outcome are located in the executions being compared, the executions are equivalent up to the next TCP. That is, the instruction instances up to the next TCP are uninteresting and can be rapidly traversed using the TBB flow graph by traversing blocks until a block with a TCP is located, using the TCP flag, and moving to the end of that block. Corollary 14 implies that a similar traversal will work for traversing from a CP instance to a TCP instance. Finally, disjoint sections of executions can be rapidly traversed independently by traversing an execution until the block containing the CP, for the TCP instance at the head of the disjoint section, is located. The CP pointer stored in each block is used to locate the correct block. In fact, the only association of TCP and CP pairs is through this pointer.

We forgo an algorithm to compute the TBB flow graph of a program and point out that it is a simple matter to construct this data structure given a program and the set of TCP and corresponding CPs for the program. The time to construct this data structure is $O(l)$ where l is the length of the input program.

6.2. The Comparison Phase

Algorithm 2 below, computes a correspondence M for two program executions. This correspondence is the one discussed in the previous section. The algorithm relies on the assumption that an

execution is in graph form rather than interleaved.

---

given: Executions E_1 and E_2 and the TBB flow graph for a program p_{TBB}

compute: M — the correspondence of E_1 and E_2

algorithm:

```
  Compute_M(E1, E2, pTBB):
1     record State = {
            boolean advance = TRUE;
                instruction * s1,s2;
                tbb * b1,b2;
                };
      correspondence M;
2     while not end(E1) or end(E2) do
3           if State .advance then
                State = get_TCP(E1, E2, pTBB , State );
4               M = M ∪ {s1,s2};
5           if (s1≡s2 == task_create) then
6               for each process pair p do
7                   M = M∪ Compute_M(E1(p ),E2(p ),pTBB (p ));
8               State = get_CP(E1, E2,State );
9               M = M ∪ {s1,s2};
10          if outcome(E1,s1) != outcome(E2,s2) then
11              State = get_CP(E1, E2, State );
12              M = M ∪ {s1,s2};
      end while
13    return M;
  end.
```

Algorithm 1. Compute a correspondence M given two executions and a program in TBB form.

The correspondence algorithm is a recursive algorithm. Three data structures are used in making the correspondence. First the TBB flow graph of the program p_{TBB} is used to direct the traversal of each execution. Second (line 1) a data structure sufficient to represent the correspondence function M is used. In general, the correspondence and a measurement should be computed on-the-fly and the correspondence itself need not be stored. Here we demonstrate that the correspondence can be calculated. Finally, the state of the executions and the current position in the TBB flow graph if each execution is retained in the record *State* . This record maintains a pointer to the current location in each execution, a pointer to the corresponding locations in the TBB flow graph for each execution, and a flag with which to determine whether or not the executions need to be advanced to locate the next TCP instruction.

The key to this algorithm is using the TBB flow graph of the program to *direct* the traversal of the executions. Because we have instruction count for each block we know exactly how many instructions to pass over in each execution in order to arrive at the next TCP instances. This is also the case when locating CP instances. In this manner, the TBB flow graph serves as a *road map* for traversing the traces. We use two functions to accomplish this traversal: *get_TCP* and *get_CP*. These functions return the current state of the correspondence in the record *State* . Recall that the TBB has a single TCP per block. The comparison begins at the first block in the program, containing the **begin** statement, and uses the information in this block to traverse to the next TCP instances in both executions. Here, the executions are identical at least until the second TCP instance is reached. In the event that **end** statement instances are reached, *get_TCP* returns the state at the end of each execution. The case of locating a CP, in which the executions are guaranteed to be disjoint, illustrates the need for two separate pointers into the TBB flow graph. This function determines whether or not the CP instances located are also TCP instances. If this is the case,t hen *State.advance* is false and *get_TCP* does not advance the state at line 3. Finally, the function *end* is used to detect the end of each trace, and in the case of recursion , then end of each process execution.

The algorithm works as follows. At the beginning of each iteration of the main loop (line 2) both executions are in an equivalent section of instruction instances. Initially, this section is that of the beginning of the program to the second TCP (the **begin** statement being the first). A TCP instance is located (line 3) and added to M (line 4). There are three cases for the TCP instances. First, if the TCP are **task_create** instances (line 5) then each pair of process executions are compared. The result of each of these comparisons are added to M. This is accomplished by recursively calling *Compute_M* with pointers to each pair of process executions and a pointer to the beginning block (in the TBB flow graph) for these executions. Notice that since the executions begin with the same block of the TBB flow graph, a single pointer suffices. Finally, the CP corresponding to the **task_create** statement is located in each execution and added to M (lines 8-9).

The next case is one in which the TCPs have a different outcome (line 10). Here, the executions are traversed and the CP for this TCP instance is located in each execution. At this point (line 11) *State* contains the updated state of the traversal (i.e., the updated pointers into each execution and the pointers to the CP in the TBB flow graph). The CP instances are added to M (line 12).

In the final case, the outcome of the TCP instances are the same. In this case, the loop drops through and continues. Here, the executions are again equivalent at least until the next TCP instance.

6.3. Correctness and Complexity

Correctness

To prove the correctness of Algorithm 1, we first show that the loop at line correctly maintains the following invariant:

(I1) At each iteration of the loop, a prefix of the executions is correctly corresponded.

We then demonstrate that the loop terminates. Lemma 17 below shows that given the assumption that lines 6-7 of the algorithm do nothing more than move the executions past the p process threads created by a **task_create** statement then the following invariant is maintained:

(I2) At each iteration of the loop, a prefix of the executions is correctly corresponded excluding the contents of **task_create** and **task_terminate** statement instances.

Lemma 17: Given any two executions of the same program Algorithm 1 maintains invariant I2.
■

Proof: We prove Lemma 17 by induction on the number iterations of the loop at line 2.

Base Case: Clearly it is not possible for the loop to fail to iterate since a program must at least have a **begin** and **end** statement. Therefore, the first iteration of the loop locates the **begin** statement instances in each execution. Line 4 adds these statements to the correspondence M_1. The conditions at lines 5 and 9 fail and the next iteration begins. Since M_1 does not violate conditions 1-3 of Definition 15, the invariant I2 is maintained.

Induction Hypothesis: Given fixed but arbitrary executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ assume that there exists a correspondence $\mathbf{M_i}$ that does not violate invariant I2 and assume that the executions are corresponded up to the kth common TCP or CP in each execution. That is, there exist corresponding prefixes $p_1(\textbf{begin},s_k)=<E_{p1},\rightarrow_{T_{p1}},\rightarrow_{sd_{p1}}>$ and $p_2(\textbf{begin},s_k)=<E_{p2},\rightarrow_{T_{p2}},\rightarrow_{sd_{p2}}>$ and $\mathbf{S_k}=(TCP(E_{p1})\cap TCP(E_{p2}))\cup(CP(E_{p1})\cap CP(E_{p2}))$, $s_k \in \mathbf{S_k}$ and $|\mathbf{S_k}| = k$.

Induction Step: For the induction step, if s_k is not an instance of the program **end** statement, then it is either a CP or a TCP. If s_k is a CP, then Corollary 14 guarantees that the executions are the same up

to (and including) the next TCP instance in each execution. Therefore, these equivalent instances will be located at line 3. The same holds true is s_k is a TCP by Lemma 12. Line 4 creates the correspondence $\mathbf{M_{i'}}=M_i \cup (s_{k+1},s_{k+1})$ which does not violate conditions 1-3 of Definition 15 since by the induction hypothesis $\mathbf{M_i}$ does not and since there exist paths $p_1(s_k,s_{k+1}) = p_2(s_k,s_{k+1})$. Given this, there are three cases for the TCP located, s_{k+1}:

1. *outcome* (p_1,s_{k+1})=*outcome* (p_2,s_{k+1}). In this case, the conditions at lines 5 and 9 both fail and the loop falls through. Since $\mathbf{M_{i+1}}=\mathbf{M_{i'}}$ is a correct correspondence, condition I2 is maintained.

2. *outcome* $(p_1,s_{k+1})\neq$*outcome* (p_2,s_{k+1}). Then the condition at line 5 fails and the condition at line 10 is true. Then by Lemma 13 there exists $c_{k+1} \in (CP(E_1) \cap CP(E_2)$ and Lemma 10 ensures that c_{k+1} can be located. Given the predecessor of c_{k+1} in E_1, call this event c_{1p}, the predecessor of c_{k+1} in E_2, call this event c_{1p}, the successor of s_k in E_1, call this event s_{1p}, and the successor of s_k in E_2, call this event s_{2p}, then there exist paths $p_1(s_{1p},c_{1p})$ and $p_2(s_{2p},c_{2p})$ that are disjoint. Condition 2 of Definition 15 implies that there can be no correspondence between disjoint paths. Hence, line 12 yields the correct correspondence $\mathbf{M_{i+1}}=\mathbf{M_{i'}} \cup (c_{k+1},c_{k+1})$ and invariant I2 is maintained.

3. s_{k+1} is an instance of a **task_create** statement. In this case, by assumption, lines 6 and 7 move over the process executions in each execution. The **task_terminate** CP, c_{k+1}, is located by Lemma 10 and line 8 creates the correspondence $\mathbf{M_{i+1}}=\mathbf{M_{i'}} \cup (c_{k+1},c_{k+1})$. Since corresponding (c_{k+1},c_{k+1}) does not violate Definition 15, the invariant I2 is maintained.

□

Next, given Lemma 17, we prove Theorem 18.

Theorem 18: Given any two executions of the same program Algorithm 1 maintains invariant I1.

■

Proof: The proof proceeds by induction on the **task_create/task_terminate** nesting level of an execution.

Base Case: The proof of Lemma 17 is sufficient to demonstrate that invariant I1 is maintained by function *compute_M* for executions with no common **task_create/task_terminate** statement instances.

Induction Hypothesis: Given fixed but arbitrary executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ with common TCP and CP instances

$\mathbf{S}=(TCP(E_1)\cap TCP(E_2))\cup(CP(E_1\cap CP(E_2))$, assume that there exists $\mathbf{M_i}$ not violating invariant I2 corresponding P_1 and P_2 up to but not including the body of **task_create/task_terminate** instances at nesting level i.

Induction Step: For the induction step, without loss of generality, choose a **task_create/task_terminate** instance pair $s_{1,i}\in\mathbf{S}\cap E_1$ and $c_{1,i}\in\mathbf{S}\cap E_1$ such that $s_{1,i}$ is a **task_create** instance at nesting level i and $c_{1,i}$ the corresponding **task_terminate** instance at nesting level i.Then by the induction hypothesis there exist $s_{2,i}\in\mathbf{S}\cap E_2$ and $c_{2,i}\in\mathbf{S}\cap E_2$ such that $(s_{1,i},s_{2,i})\in\mathbf{M_i}$ and $(c_{1,i},c_{2,i})\in\mathbf{M_i}$. There exist paths $p_1(s_{1,i},c_{1,i})$ and $p_2(s_{2,i},c_{2,i})$ which contain n process executions between which there exists a one-to-one mapping, line 6. Lemma 17 implies that for each process execution there is a correspondence up to nesting level one, or $\bigcup_{j=1}^{n}\mathbf{m_j}$ computed in line 7. This implies that the correspondence $\mathbf{M_{i+1}}=\mathbf{M_i}\cup(\bigcup_{j=1}^{n}\mathbf{m_j})$ upholds invariant I2.

□

Finally, we are guaranteed by Assumption A1 that programs terminate and hence there is an instance of an **end** statement in each execution. If the statement is a CP of some statement other than just the **begin** statement, then it is located at line 10 and the loop terminates on the next iteration. Otherwise, it is located at line 3, lines 5 and 9 fail and the loop terminates on the next iteration.

Complexity

Function *Compute_M* has linear time complexity because each instruction instance in each execution is visited only once and immediately discarded. The function need not be recursive and is written this way solely for ease of understanding. In addition, the comparison algorithm is intended to be a vehicle for making accurate distance measure calculations. For this reason, the correspondence itself need not be stored. Given this, a correspondence can be made using O(l) space where l is the size of the program input. For parallel programs, the size of the program is likely to be negligible in relation to the size of the executions it produces.

We demonstrate that function *Compute_M* has linear time complexity. Function *get_TCP* traverses each execution either from a TCP instance to its TCP successor instance or from a CP instance to the next TCP instance in the execution. Lemma 12 and Corollary 14 (respectively) ensure that a simple linear search of the executions is sufficient for function *get_TCP* used at line 3. Similarly, Lemma 13 and Lemma 10 ensure that using a simple linear search will suffice for function *get_CP* used at lines 8 and 11. Finally, the recursive call at line 7 will traverse each pair of process executions a single time. These

executions have not been visited before the recursive call and will not be visited after. Since lines 3, 7, 8, and 11 all represent linear traversals of the execution and since no other lines consume execution instances, the function has linear time complexity in the length of the input traces. Coupled with the preprocessing phase, if the executions are of length n and m and the program is of length l, then the complexity of function *Compute_M* is $O(l + m + n)$.

Given that the correspondence is not stored and that correspondence algorithm need not be recursive, we demonstrate that the space complexity for function *Compute_M* is $O(l)$. First, storing the program in intermediate form accounts for the majority of storage consumption. Lines 3, 8, and 11 get a TCP or CP instance from each execution, which can be immediately discarded at lines 4, 9, and 12. Given that the information collected at line 3 can be discarded at line 5, the only information used after a recursive call is the CP pointer for the CP associated with the **task_create** TCP that caused the recursive call. Notice that this CP is the *current* instruction instance in each execution *and* is the current instruction of the intermediate form. Since it can be easily located, there is no reason to store a pointer to the CP in the intermediate form. Hence the function stores only the program intermediate form and the a single copy of the record *State* .

Two assumptions were made in order to achieve the above time and space bounds. The first, assumption A2, guarantees that a statement has a single entry and exit point. There may be no lemmas that correspond to Lemmas 10, 12, and 13 in the absence of this assumption. It seems almost a certainty that executions will require more than a linear search if the requirement is lifted.

The second assumption is that process executions are not stored in interleaved form. If the correspondence algorithm is required to undo the interleaving of process executions, then at worst case, for a **task_create** statement one half of all of the process executions in both executions must be stored before process executions to correspond with the process executions already stored are encountered. This implies not only that the space requirement for the correspondence is at worst case $O(n + m)$[6] but that a second traversal of each process execution may be needed. This does not however change the worst case linear time for the correspondence.

7. Measurement and Correspondence

Our immediate goal has been to correspond two parallel program executions in a reasonable amount of time and space. With this algorithm, we can now calculate distance measures between parallel program

6 In reality, the actual storage usage will be much less except in the rare case that the program consists of a single **task_create/task_terminate** statement.

executions. We hope to provide distance measures that can be used in trace driven simulations and other performance analysis areas. Ultimately, we hope that these measures will shed light on a quantification of a parallel program with respect to its possible execution traces. For example, one might wish to quantify the difficulty of collecting a trace from a particular parallel program, the difficulty of extrapolating that trace, the variance one might expect to see in simulation results using a particular program, etc.

In this section, we discuss possible difference measures for parallel program executions. We present three measures that are simplistic and can be easily calculated. These measures serve two purposes. First, to illustrate how a distance between two executions might be calculated. Second, even though these measures are simplistic, they represent the measurement of fundamental structural differences between executions.

Difference Measures for Parallel Programs

We have identified five types of distances in complexity between parallel program executions that we believe are useful (this list is by no means an exhaustive list of the types of differences in parallel program executions that one might measure). They are: nondeterministic complexity, trace complexity, tracing complexity, transformation complexity, and time distances. A nondeterministic distance should measure the distance between two executions with respect to the effect that nondeterminism had on each execution (i.e., how far the execution paths diverge). A trace complexity distance should measure the difference in complexity of basic operations (e.g., storage and retrieval) for the executions. A tracing complexity distance should provide a measure of the increase in difficulty of actually collecting one particular trace versus another using either instruction sampling at runtime or program instrumentation. These two measures are closely related. The transformation complexity distance for an execution provides a quantification of the difficulty of transforming one execution into another. Preliminary work in trace migration [6] [3] suggests that such a measure is closely related to the nondeterministic distance between two programs. Finally, a time distance should quantify the differences between the time trajectories for two executions due to differences in the executions (i.e., a time warp).

The nondeterministic complexity of a parallel program and its trace complexity are all dependent on program TCPs [3]. The *fanout* distance, a measure of the nondeterminism induced branching distance between two executions, is a nondeterministic complexity distance. This distance is based on the $\rightarrow_k$ family of parallel execution equivalences (see [3]). The fanout distance determines the largest possible branching distance that occurs between two executions during a period of divergence. That is, given two weights w_{t1} and w_{t2}, the fanout distance between two divergent sections of a parallel program execution is:

(1) $$f = l_{t1} \cdot w_{t1} + l_{t2} \cdot w_{t2}$$

Where l_{t1} and l_{t2} are the maximum nesting level due to nondeterminism reached in each divergent section.

Given two executions a and b, The *TCP* distance is a trace complexity difference for parallel program executions. This distance calculates the difference in actual TCPs executed during a period of divergence. Given a count of the number of TCPs executed during a period of divergence, c_a and c_b, the TCP distance between the sections *tcp* is:

(2) $$tcp = |c_a - c_b|$$

This distance measure is also closely related to the $\rightarrow_k$ family of parallel execution equivalences.

One possible tracing complexity distance is the *TCP frequency distance*. Given the minimum frequency of TCP occurrence during a single period of divergence in each execution, f_{t1} and f_{t2} respectively, the TCP frequency distance for this period is:

(3) $$tcp_f = |f_{t1} - f_{t2}|$$

Notice that the TCP frequency distance fits nicely into our framework, see [3], when considered as a weak structural equivalence.

We can combine these simple distance measures to create more accurate distance measures. For example, consider a distance measure that takes into consideration the difference in nondeterministic complexity, trace complexity, and actual trace contents (i.e., the number of divergent periods). We combine these three differences into a single measure using a weighted sum calculation. Given a *nondeterministic complexity weight* w_{nd}, a *trace complexity weight* w_{tr}, and a *divergence weight* w_d, and given the fanout and TCP differences between each divergent section, f and *tcp* respectively, we compute the following difference between two executions:

(4) $$d = \sum_{i=1}^{n} (w_d + f \cdot w_{nd} + tcp \cdot w_{tr})$$

This measure represents a more accurate measure of nondeterministic distance and corresponds our intuition regarding nondeterministic distance. To see this, notice that multiple instances of divergence due to nondeterminism, need to be accounted for in any accurate distance measure. The addition of the w_d calculation takes these divergent periods into account. The fanout measure alone gives little information as to how different two divergent periods really are. This is because it considered only the largest branching

distance that occurs during a divergent period. The TCP distance provides a more accurate account of the extent of the nondeterminism encountered during divergent periods. These two calculations coupled with the fanout distance represent a reasonably accurate account of the nondeterministic difference between two executions. For example, consider two executions that execute exactly the same set of TCPs (i.e., that are weakly structural equivalent [3]). For these executions, the distances f and tcp will always be zero leaving the distance calculation $d=\sum_{i=1}^{n}(w_d)$. This is an accurate distance in that the only nondeterministic difference between the two executions is d divergent sections. Next consider two different pairs of executions and for clarity, let w_{nd} and w_{tr} both be one. The first pair are the same except that they each execute a single TCP during each divergent section. The fanout distance at each section will be two and the TCP distance will be zero. We will refer to the distance between these two executions as d_1. Now consider either of these executions paired with an execution that has the same divergent sections but that executes more TCPs and that has a fanout value that is at least one during each divergent period. Let this distance be d_2. We have $d_1<d_2$ corresponding to our intuition that the second pair of executions have a greater nondeterministic distance than the first pair. The measure corresponds our intuition about nondeterministic distance. Finally, this distance measures is calculable in linear time and reasonable space using the correspondence algorithm.

Calculating Distance Measures

It is a simple matter to combine the correspondence algorithm with the distance calculation given in (3). In Algorithm 1, two functions actually traverse the input executions. These are *get_TCP* and *get_CP*. The former traverses executions only when they are equivalent and hence have a zero distance. The latter is the only place in Algorithm 1 that traverses disjoint sections of executions and traverses only disjoint sections. Any distance calculation will have to be done while this function traverses the executions.

Algorithm 2 below is a version of *get_CP* that in addition, calculates the distance calculation given in (3).

given: two executions E_1 and E_2 and a state *state_in*.

compute: Advance each execution to the CP for this block. Return this new state. Calculate the distances d_f and d_{TCP}.

algorithm:
global:

```
1     int pair d_f, d_TCP;
2     int w_f, w_1, w_2, w_tr;
3  get_CP(E_1, E_2, state_in):
4     instruction *destination_cp;
5     destination_cp = state_in.(b_1)→CP;
6     while (state_in.b_1 ↔ destination_CP) do
7        compute_d_f (state_in.b_1,1);
8        compute_d_TCP (state_in.b_1,1);
9        s_1 = s_1 + state_in.b_1.instr_count;
10       state_in.b_1 = next_block(state_in);
11    while (state_in.b_2 ↔ destination_CP) do
12       compute_d_f (state_in.b_2,2);
13       compute_d_TCP (state_in.b_2,2);
14       s_2 = s_2 + state_in.b_2.instr_count;
15       state_in.b_2 = next_block(state_in);
16    if (state_in.b_1.instr_count == 1 and state_in.b_1.TCP == TRUE) then
        state_in.advance = TRUE;
      else
        state_in.advance = FALSE;
17    d = d + resolve_d();
18    d_f = d_TCP = (0,0);
19    return(in_state);
end;
```

Algorithm 2. Function get_CP.

Recall that *get_CP* is invoked a single time for each period of divergence. Lines 8 and 9 and lines 13 and 14 calculate the distances given in (1) and (2) (respectively) for each execution. For the TCP distance, the calculation is a running sum of the TCPs encountered in each execution. For the fanout distance, the calculation keeps track of the largest nesting level reached due to nondeterminism. The function *resolve_d* calculates d for each period of divergence. Notice that *get_CP* uses the information present in the TBB to rapidly traverse each execution up until the CP is reached for the current TCP (see Section 6.1).

The actual distance calculation is given in Algorithm 3. The function *resolve_d* provides a running calculation of the distance given in (3).

given: a TBB pointer b and a flag *which* determining which execution is being measured.
compute: Calculate the distances d_f, d_{TCP}, and d.

algorithms:
global:
1 d_f, d_{TCP}, w_f, w_1, w_2, w_{tr};
2 TBB *stack[2,N] = (empty,empty);
static:
3 int pair nd_f;

4 compute_d_f (b ,which):
5 if ($b.CP$) then
6 pop(stack[which]);
7 if (nd_f.[which] > d_f) then
d_f = nd_f.[which];
8 nd_f.[which] = nd_f.[which] - 1;
9 if ($b.TCP$) then
10 push(stack[which],$b.CP$);
11 nd_f.[which] = nd_f.[which] + 1;
12 return();
13 end;

14 resolve_d_f ():
15 if ($b.CP$) then
16 pop(stack[1]);
17 if (nd_f.[1] > d_f) then
d_f = nd_f.[1];
18 pop(stack[2]);
19 if (nd_f.[2] > d_f) then
d_f = nd_f.[2];
20 nd_f = (0,0);
21 return(d_f.[1]*w_1 + d_f.[2]*w_2);
22 end;

23 compute_d_{TCP} (b ,which):
24 if ($b \rightarrow TCP$ == TRUE) then
d_{TCP}.[which] = d_{TCP}.[which] + 1;
25 return();
26 end

27 resolve_d ():
28 return(1 + resolve_d_f ()*w_f + | d_{TCP}.[1] - d_{TCP}.[2] | * w_{tr});
29 end;

Algorithm 3. Compute and resolve distances d_f, d_{TCP}, and d_{TG}.

Function *compute_d_f* at line 4 uses a stack to compute the current TCP nesting level maintained in the pair nd_f. The variable d_f maintains the largest depth that has currently been reached. Function

resolve_d_f at line 14 finishes the calculation by popping the last nesting level off of the stack. It returns the distance given in (1). Function *compute_d_{TCP}* at line 23 simply calculates the number of TCPs encountered in each execution keeping the running calculation in d_{TCP}. The distance given in (2) is calculated in function *resolve_d* at line 27 where the distance given in (1) is also calculated.

8. Computing Optimal Correspondences

As previously mentioned, our choice of correspondence was not the only possible choice. In this section, we justify the choice we have made. Ideally, one would like an algorithm that computes an optimal correspondence for any pair of executions w_1 and w_2 and measure m. We conjecture that there is no such algorithm that satisfies the time and space requirements imposed by executions.

We begin with the observation that any correspondence computation that requires searching the input executions, will fail to meet out time and space requirements. Given this, we have the following conjecture:

Conjecture 1: The constraints of reasonable space usage and linear time imply a fixed correspondence calculation.

Consider another choice of correspondence. In particular, the correspondence that corresponds loop iteration starting with the final iterations. That is, the final iteration in each execution is corresponded, then the second to final, and so on. This leaves any unmatched iterations as the first iterations of the loop. Even this simple correspondence requires some searching — that of locating the final iterations of each loop. This correspondence will fail our space requirements.

Next, we conjecture that there is no algorithm satisfying the linear time and reasonable space requirements that computes an optimal correspondence for any pair of executions and any measure. Let $m(w_1,w_2)$ represent the optimal measure of distance for m on w_1 and w_2. Let $m(\mathbf{C}(w_1,w_2))$ represent the optimal measure of distance for m given correspondence $\mathbf{C}$. Then we have the following conjecture:

Conjecture 2: Given any two executions w_1 and w_2, there is no linear time and reasonable space algorithm computing $\mathbf{C}$ such that

$$\exists \mathbf{C}\, \forall m.m(w_1,w_2) \geq m(\mathbf{C}(w_1,w_2)) \tag{4}$$

is true for all choices of w_1, w_2, and m.

To support our conjecture, consider the simple example program given in Figure 2 below. This program loops from 0 to i-1 and executes different sections of code based upon whether or not i-j is even or odd. Notice that there are 4 different possible executions in which the loop iterates either once, twice, three times or four times. Consider the comparison of two executions where i is even for one and odd for the other. Our method of correspondence will make a bad choice of correspondence. Here, the method outlined above would be more suitable. On the other hand, our method will give a better correspondence for executions in which i is either both odd or both even (e.g., i = 1 and i = 3 or i = 2 and i = 4). Unfortunately, any correspondence algorithm making a good correspondence in both cases will have to store the loop iterations and search for an optimal correspondence.

Conjecture 2 seems a reasonable one and given this and the time and space constraints, we have chosen the correspondence computed in Algorithm 1 for two reasons. First, this correspondence can be made more efficiently (time and space) than all other possible correspondences. Second, it is a reasonable correspondence for most loops (e.g., it seems reasonable to assume that most loops will perform the same calculation given the same index value).

```
                                A0: begin
                                      shared int x = 0;
                                      shared int lk = 0;
                                      int n = 3;
                                A1: task_create(n)

task 0                          task 1                  task 2
     int i = 0;
B0: lock(lk);                   C0: lock(lk);           D0: lock(lk);
B1: x = x + 1;                  C1: x = x + 1;          D1: x = x + 2;
B2: i = x;                      C2: unlock(lk);         D2: unlock(lk);
B3: unlock(lk);
B4: for j = 0 to i do
B5:    if (even(i-j)) then
B6:      do_something;
B6:    else
B6:      do_something_different;
B7:    k = k + 1;

                                A2: task_terminate(n);
                                A3: end;
```

Figure 2. Another parallel program.

9. Conclusion and Future Directions

Our intent with this research is to relate sequence analysis work to the fundamental problem of comparing parallel program executions.

We have argued that existing sequence analysis techniques are not reasonable for parallel program executions due to their potentially large size. We have shown that through the use of information collected in the program source, executions can be corresponded in linear time with negligible space usage. While this correspondence is not guaranteed to produce an optimal correspondence it is guaranteed to be linear. This correspondence is an improvement over existing linear correspondence methods that produce inaccurate measures and facilitates the calculation of new distance measures for parallel program executions. In addition this algorithm is an improvement over methods of correspondence from sequence comparison which have a quadratic worst case time and require the storage of the sequences being compared.

We have presented one possible measure of parallel program execution difference and demonstrated how this measure can be calculated using the correspondence algorithm.

We have justified our particular choice of correspondence by arguing that there is no single algorithm meeting our time and space constraints *and* supplying an optimal correspondence for any choice of executions and measure. Our choice of correspondence can be calculated more efficiently than any other choice (as it requires no storage of executions), and is a reasonable choice given the behavior of most loops.

10. Acknowledgements

The first author has been supported by a grant from the Convex Computer Corporation, who are also providing general support for all of our work on the parallel program tuning environment. We would also like to acknowledge Harini Srinivasan for discussions on the problem of computing TCPs.

References

1. A. Borg, R. E. Kassler and D. W. Wall, ''Generation and Analysis of Very Long Address Traces'', in *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, CA, May 1990, pp. 270-279.

2. Z. K. F. Eckert and G. J. Nutt, ''Locating Trace Change Points in Parallel Programs'', Tech. Rep. CU-CS-730-94 , Dept. of Computer Science, University of Colorado, June 1994.

3. Z. K. F. Eckert and G. J. Nutt, ''A Framework for Execution Order Distortion in Shared Memory Multiprocessor Event Traces'', Tech. Rep. CU-CS-708-94, Dept. of Computer Science, University of Colorado, March 1994.

4. Z. K. F. Eckert and G. J. Nutt, ''Parallel Program Trace Extrapolation'', in *To Appear Proceedings of the 1994 International Conference on Parallel Programming*, St. Charles, Illinois, August 16-20, 1994.

5. P. A. V. Hall and R. Dowling, ''Approximate String Matching'', *ACM Computing Surveys 12*, 4 (1980), pp. 381-402.

6. M. A. Holliday and C. S. Ellis, ''Accuracy of Memory Reference Traces of Parallel Computations in Trace-Driven Simulation'', *IEEE Transactions on Parallel and Distributed Systems 3*, 1 (Jan. 1992), pp. 97-109.

7. L. Lamport, ''How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs'', *IEEE Transactions on Computers c-28*, 9 (SEPT 1979), pp. 690-691.

8. L. Lamport, ''On Interprocess Communication Part I: Basic formalism'', *Distributed Computing 1* (1986), pp. 77-85.

9. D. Sankoff and J. B. Kruskal, *Time Warps, String Edits, and Macromoles: The Theory and Practice of Sequence Comparison*, Addison-Wesley Publishing Comapny Inc., Reading, MA, 1983.

10. M. N. Wegman and F. K. Zadeck, ''Constant Propagation with Conditional Branches'', *ACM Trans. on Programming Languages and Systems 13*, 3 (Jul 1991), pp. 319-349.