PROGRAM ANALYSIS

CS6843

Scribe - Week 7 (20-24 February 2017)

Shreyas Harish (CS13B062)

Aman Sharma (CS15S049)

Lecturer

Dr. Rupesh Nasre

March 9, 2017

Contents

Section 1

Pointer Analysis Optimizations

1.1 Propagation Order

Variations in the order of propagation of information may sometimes have benefits depending upon the target at hand. A topological ordering is such an ordering in a graph where edges coming into a node are serviced before edges going out of the node. This sort of ordering is beneficial in the case of points to information propagation. Similarly a depth first propagation allows easy reuse of the points to information difference between adjacent nodes.

1.2 Constraint Order

Similar to the propagation order, the order of evaluation of constraints can be shifted around. Here we shall attempt to order the constraints optimally so that the points to information analysis can be completed in minimum time. We shall see that this is an NP-Complete problem and we shall look at some heuristics for constraint ordering.

1.2.1 Reducible from Set Cover Problem

The **Set Cover problem** is defined as follows:
$SC(U, S, K)$
U : Universal set of elements
S : Set of Subsets S_i
K : Some Number
The problem requires us to find K subsets in S which cover all the elements in U

The **Points To Analysis problem** is defined as follows:
$PTA(C, S, K)$
C : Set of all copy constraints
S : is a variable of interest, for which we need to reach a fixed point
K : The number of steps in which the fixed point is reached
The problem requires us to verify whether the above conditions hold for C, S and K

$Example$:
$SC(U, S, K)$
$U : \{1, 2, 3, 4, 5\}$
$S : \{\{1, 4\}, \{2, 5\}, \{2, 4, 5\}, \{3\}\}$
$K : 3$
2 Solutions to the problem are:
$\{\{1, 4\}, \{2, 5\}, \{3\}\}$
$\{\{1, 4\}, \{2, 4, 5\}, \{3\}\}$

Reduction :
Idea-
Copy Constraints map onto Subsets
Adding copy constraints gives the variable of interest
Adding subsets give the universal set
The reduction can then be done in linear time as shown

Algorithm-

- $SC(U, S, K) \geq PTA(C, S, K)$
- Linear Time Reduction
  - $\forall s \in S_i$ add s to ptsto(S_i)
  - $\forall$ sets S_i create a copy statement $S = S_i$
- Solution to PTA $\implies$ Solution to SC
  - The set cover problem has been mapped onto the PTA problem.
  - If we can solve the PTA problem, then for any k copy statements, we know the points-to sets for each element.
  - Thus, through the PTA problem we can find a set of k sets which together cover the universal set. This is analogous to using k copy statements to have S point to all elements.
- Solution to SC $\implies$ Solution to PTA
  - The mapping from PTA to SC is implicit. The points to information from each copy constraint is mapped onto a set containing those elements.
  - If any k sets cover the universal set, those k copy constraints may be ordered first.
  - with respect to the variable of interest this helps us reach a fixed point in k steps.

The set cover problem is an established NP-Hard problem. This means that the set cover problem is at least as hard as every other problem in the class of NP problems. Therefore every NP-Hard problem can be reduced to the set cover problem. Below we show that the set cover problem can be reduced to the constraint ordering problem. Thus, the constraint ordering problem is also NP-Hard. Solutions to the constraint ordering problem can be verified in polynomial time. Thus the verification of a solution is an

NP problem. By definition that makes the constraint ordering NP-Complete as the problem is NP-Hard and its solutions are verified in polynomial time.

Example :

$SC(U, S, K)$

$U : \{a, b, c, d\}$

$S : \{\{a, b, c\}, \{b, d\}, \{a, c, d\}\}$

$K : k$

can be reduced to

$PTA(C, S, K)$

$S : \{a, b, c, d\}$

$S_1 = \&a$, $S_1 = \&b$, $S_1 = \&c$

$S_2 = \&b$, $S_2 = \&d$

$S_3 = \&a$, $S_3 = \&c$, $S_3 = \&d$

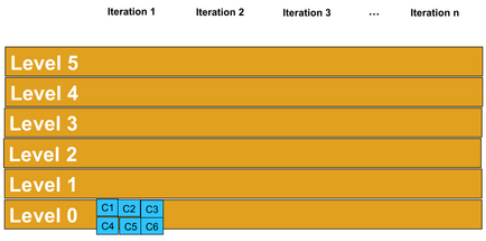
$C : \{\{S = S_1\}, \{S = S_2\}, \{S = S_3\}\}$

$K : k$

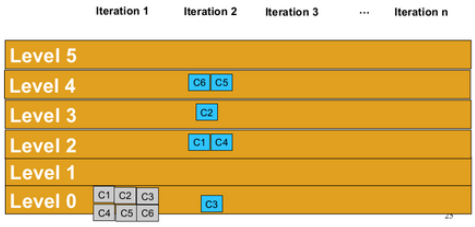
From this we can see that the Points To Analysis problem is reducible from the Set Cover problem. Further we notice that the Points To Analysis problem is NP-Complete. Therefore, in order to solve the optimal constraint ordering problem in reasonable time we will use the following heuristics.

1.2.2 Bucketization

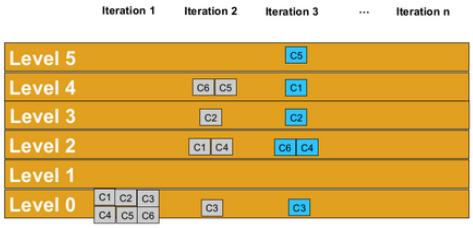
A simple heuristic is assigning a priority to each constraint. The priority in iteration i might be based on the amount of new points-to-information added by that constraint in iteration i-1. Based on the priorities, the constraints are grouped into different levels and are ordered on the basis of them. Below is an illustrated example of bucketization in constraint ordering over a few iterations.



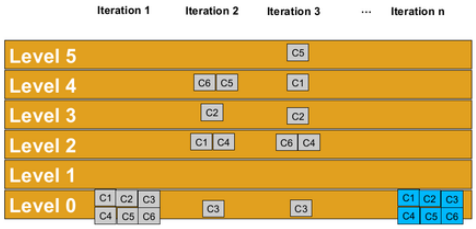
(a) sample bucketization iteration 1



(b) sample bucketization iteration 2



(c) sample bucketization iteration 3



(d) sample bucketization iteration n

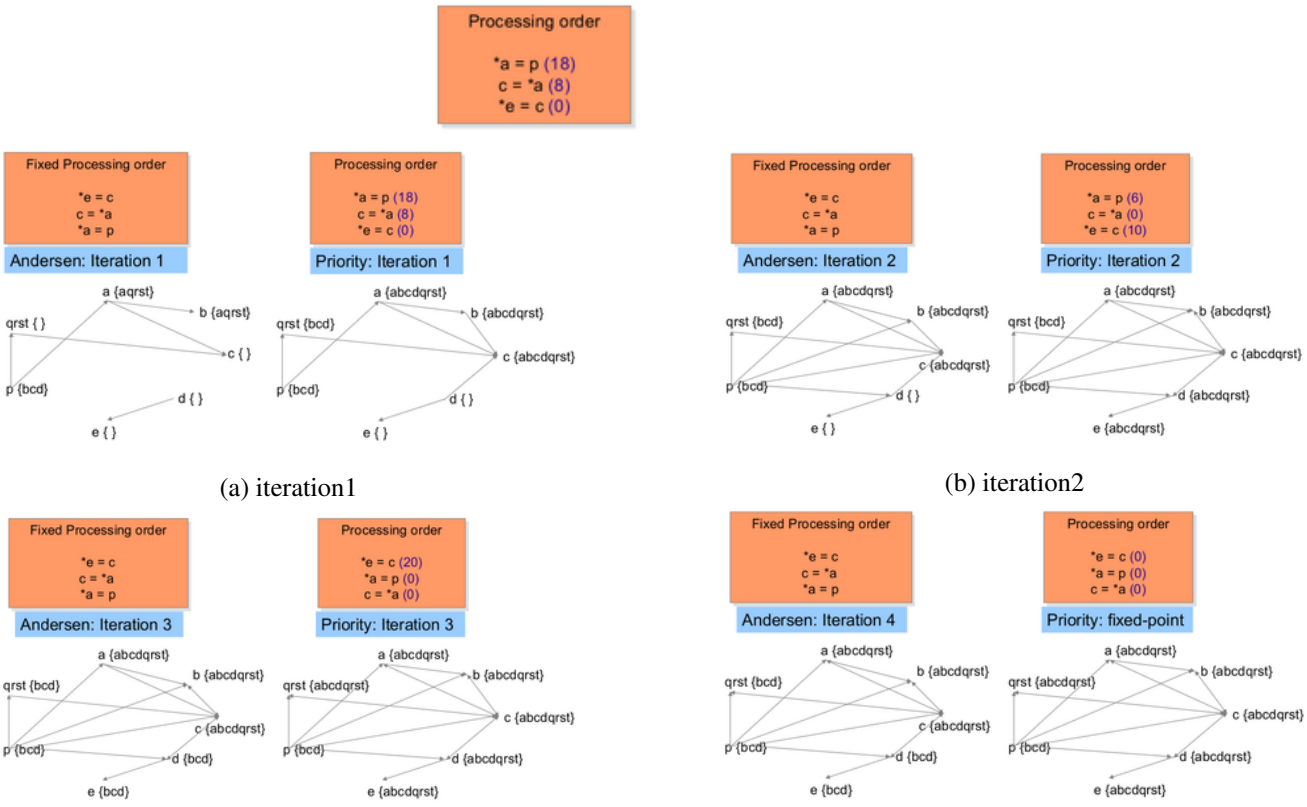
In the figure above all constraints start off in level 0 for the first iteration. Based on the amount of information that each of them generate in that first iteration each of them is moved to a different level for the next iteration. Constraints in higher levels contributed more information in the previous iteration $i-1$ and are evaluated first in iteration i. This continues till iteration n where all constraints settle back to level 0 because they have reached a fixed point.

1.2.3 Skewed Evaluation

Skewed evaluation is a heuristic which involves a slight modification on bucketization.Copy constraints in some levels may be evaluated more than once in a single iteration. Invariably higher level copy constraints may be evaluated more than once an iteration. For example, in the above figure in iteration 2 C2, C5 and C6 may be evaluated more than once. In iteration 3 C2, C1 and C5 may be evaluated more than once.

1.2.4 Prioritized Points to Analysis vs Andersen's

The reordering of constraints in prioritized points to analysis helps ensure that more points to information is propagated in fewer iterations. We can see that prioritized points to analysis reaches a fixed point before Andersen's analysis does, through the following illustration.



(a) iteration1

(b) iteration2

(c) iteration3

(d) iteration4

Figure 1.2: Andersen's Analysis vs Prioritized Points to Analysis

Section 2

Applications

Here we will discuss a few areas where pointer analysis can be used. Some possible goals of pointer analysis are:

- Dead Code Elimination
- Common Subexpression Elimination
- Parallelization
- Escape Analysis

2.1 Dead Code Elimination

This is a process of identifying code that will never be executed at run time and discarding it during compile time. Pointer analysis is used here to identify whether certain conditions are satisfied. Thus we identify whether control flow is ever transferred to segments of code whether they are 'dead'. The example below shows sample code and the corresponding conditions that need to be checked using pointer analysis. The following code snippet is analyzed for potential dead code.

```
a = s1.arr;
b = s2.ptr;
q = &a[ii];
p = &b[jj];
if (p == q) {
    x = 10;
    y = 100;
} else {
    x = 20;
    y = 30;
}
```

The following conditions need to be tested statically for the dead code elimination in the above mentioned code snippet:

1. p == q
    - p and q are then written as a and b shifted by ii and jj elements respectively (as defined earlier).
2. a + ii * typesize == b + jj * typesize
    - a and b are then written as s1.arr and s2.ptr (as defined earlier).
3. s1.arr + ii * typesize == s2.ptr + jj * typesize

2.2 Common Subexpression Elimination

This is a process of identifying expressions which evaluate to the same value and avoiding repeated calculation. Pointer analysis is used here to identify whether certain expressions point to the same locations or reference the same variable. Several iterations of pointer analysis maybe required to identify common subexpressions. When such expressions are identified, the compiler can optimize running time by reducing the number of times the expression is evaluated. The following code snippet is analyzed for potential common subexpression elimination.

```
p = s1.arr;
q = s1.ptr;
if (p + i == q + j) {
        x = 10;
        y = 100;
} else {
        x = 20;
        y = 30;
}
```

The following conditions need to be tested statically to identify if the expression is common:

1. p + i == q + j
    - p and q are then written as s1.arr and s1.ptr (as defined earlier). i and j are converted to their values multiplied by the type size of their elements.
2. s1.arr + i * typesize_i == s1.ptr + j * typesize_j

2.3 Parallelization

Parallelization involves identifying segments of code that are independent of each other and executing them simultaneously. Pointer analysis is crucial here in identifying segments of code which are dependent and ensuring that they aren't executing in parallel(unsound optimization). In any 2 definitions if one of them writes to a variable common to both definitions, they cannot be executed in parallel. The following code snippet is analyzed for parallelization.

```
f() {
    *p = 10;
}
g() {
    *q = 20;
}
main() {
        f();
        g();
}
```

The following conditions need to be tested statically to identify if the functions are parallelizable:

1. !alias(*p, *q)

2.4 Escape Analysis

Escape Analysis involves checking whether a function uses variables whose definitions aren't available within its direct scope. That is, it checks whether a given variable's definition 'escapes' the given function. Most commonly we must check whether a given variable is a global or heap variable (or has a broader scope than the given function). Pointer analysis is used here to identify whether the definition of an expression is found locally or escapes the function. The following code snippet is analyzed for escape analysis.

```
f() {
        *p = 10;
}
```

The following conditions need to be tested statically to identify if the definition escapes function f:

1. if p points-to any global / heap variable.
2. pointsto(p, x) where x globals or x heap-allocated

Section 3

Parallel Points-to Analysis [1]

3.1 Introduction

Pointer analysis is one of the most important static analyses during compilation. It tells us whether two pointers may point to the same location at run-time (aliases). If we look at the example of sequential points-to analysis (say Andersen's), we see that it takes 13 steps to reach the fixed point.

Points-to Analysis

Sequential analysis (Andersen's) takes 13 steps.

Constraint	Iteration 0	Iteration 1	Iteration 2	Iteration 3
p = &a	p ⟶ {a}			
a = &x	a ⟶ {x}			F
b = &y	b ⟶ {y}			I
c = &z	c ⟶ {z}			X
d = &w	d ⟶ {w}			E
q = p		q ⟶ {a}		D
a = b		a ⟶ {y}		
e = a		e ⟶ {x, y}	e ⟶ {z, w}	
r =q		r ⟶ {a}		P
a = c		a ⟶ {z}		O
s = r		s ⟶ {a}		I
e = *a				N
t = s		t ⟶ {a}		T
a = d		a ⟶ {w}		
*e = a		x,y ⟶ {x,y,z,w}	z,w ⟶ {x,y,z,w}	

Figure 3.1

Sequential Analysis : Computation

Updating points-to set information of a pointer is counted as one step.

- Iteration 0 - We process all the address-of constraints first. We are not counting them in the calculation of the number of steps.

- Iteration 1 - We process all the instructions one by one in sequence and update the points-to set information of the pointers. In the example given above, we start with $q = p$. Here p points-to a, we add this information to q's points-to set. Hence, q starts pointing to a. Since q's points-to set information has been updated, we count this as one step. Similarly, we process rest of the statements one by one till the last statement $*e = a$. In the last but four statement $e = *a$, a points to x and x points-to set is empty. So, $e = *a$ doesn't add anything to the points-to set of e. Out of the 10 statements, 9 statements updates the points-to set information. So, the number of steps counted are 9.

- Iteration 2 - We process all the instructions again, one by one in sequence and update the points-to set information of the pointers. The first two statements doesn't add anything new to the points-to set information of q and a. Third statement, e's points-to set information gets updated with the a's new points-to set information from the previous iteration. Rest of the statements do not add anything new to the points-to set information except the last one. Since e's points-to set is updated and it contains {z,w} now, z's and w's points-to set information are updated with the points-to set of a. Three pointers (e,w,z) points-to set information is updated in this iteration. So, the number of steps counted are 3.

- Iteration 3 - There are no new points-to set information that need to be updated. Hence, it reaches a fixed point - leaving the number of steps count as 13.

Sequential analysis is taking a lot of time and also it is not utilizing the available hardware resources (multiple cores). We need to see if we can do better than this with the available resources. Parallel algorithms are vital for simultaneous use of multiple cores. Since theoretically, any fair order of updating the transfer functions will eventually stabilize to a fixed point, we can take advantage of the available multiple resources to run the points-to analysis in parallel. So, any fair parallel update order should serve our purpose. However, the orderings might differ in the number of updates necessary to reach the fixed point and may reach the fixed point fast.

3.2 Parallel Points To Analysis - Approaches

We will look into the following two methods for parallel points-to analysis.

3.2.1 Naive Method

To take advantage of parallelism, we need to make sets of independent constraints. Constraints in one set will be independent of another and constraints within a set will have conflicting constraints. To get the conflicting constraint, we need to see the read and write set of all the constraints and analyze the data race between them. Figure 3.2 depicts the read-write set for all the normalized instructions.

Read-set contains pointers whose points-to information is read and Write-set contains pointers points-to information is to be updated. If statements of the form :

$$p = q \; and \; q = r$$

Points-to Constraints		Read Set	Write Set
p = &q	address of	{ }	{ p }
p = q	copy	{ q }	{ p }
p = *q	load	{ q } U {x : q ⟶ {x}}	{ p }
*p = q	store	{ q,p }	{x : p ⟶ {x}}

Figure 3.2

are both present in the program, it may lead to a data race. To avoid data race, such statements should be executed in sequential manner, clubbed in a single thread single core.

Necessary condition for data race:

1. Same memory location is accessed.
2. One write operation.
3. More than 1 thread.
4. Simultaneous access.

To avoid data race, we need to make sure that one of the above listed conditions doesn't hold. To avoid the necessary condition 1, write should be done to copies of variables and then merging should take place. To avoid necessary 4, locks & other synchronization can be used.

Now considering the conflicting constraints in the example shown in Fig 3.1, we can come up with the following formulation of the constraints that is being executed in parallel by 2 threads.

Thread 1	Thread 2
a = b	q = p
a = c	r = q
a = d	s = r
e = *a	t =s
e = a	
*e = a	

Figure 3.3

Thread 1 : Computation

We process the instructions one by one in sequence and update the points-to set information of the pointers.

1. a = b : a's points-to set is updated with b's points-to set.

2. a = c : a's points-to set is updated with c's points-to set.
3. a = d : a's points-to set is updated with d's points-to set.
4. e = *a : e's points-to set is updated with (b,c,d,x)'s points-to set.
5. e = a : e's points to set is updated with a's points-to set.

   ***e= a and e –> {b,c,d,x}**
6. *e = a : b's points to set is updated with a's points-to set.
7. *e = a : c's points to set is updated with a's points-to set.
8. *e = a : d's points to set is updated with a's points-to set.
9. *e = a : x's points to set is updated with a's points-to set.

Thread 2 : Computation

Similar to thread 1, we process the instructions one by one in sequence and update the points-to set information of the pointers.

1. q = p : q's points to set is updated with p's points-to set.
2. r = q : r's points to set is updated with q's points-to set.
3. s = r : s's points to set is updated with r's points-to set.
4. t = s : t's points to set is updated with s's points-to set.

Thread 1 takes 9 steps and thread 2 takes 4 steps. As Amdahl's Law states that speedup is limited by the sequential part of the task. So even if the analysis is provided with 8 cores, the parallel analysis still requires 9 steps.

To summarize the naive method, following steps are performed in order:

1. Finding conflicting operations.
2. Schedule constraints.
3. Analyze in parallel.
4. Update points-to information.

Repeat steps 1 through 4 till fixed point is reached.

Finding conflicting constraints is a costly operation as the conflicting constraints will only increase. We merge the points-to information of all the threads after every iteration.

3.2.2 Replication Based Approach

In the replication based approach, we get rid of the costly operation in the naive method, i.e, finding conflicting operations. We arbitrarily partition the points-to constraints, maintain two kinds of copies of the points-to set that are to be written - one local to each thread and the other global copy. So, we replicate the points-to sets that are to be written for each thread, execute them in parallel with the replicate copies and finally merge the information from the local replicated copies into the global copy and keep repeating this till the fixed point is reached.

The following are the steps for computing the replication based approach.

1. Schedule constraints.
2. Analyze in parallel.
    a) Initial reads from the master copy.
    b) Writes to local replica.
3. Update points-to information.
    a) Merge local replicas with the master copy.

Repeat the steps from 1 until fixed point is reached.

Why Replication Works

1. Monotonically increasing computation - Points-to sets never shrink.

   We partition the constraints and process them in separate threads. We do only two things - either add information to the points-to set or do nothing. Since we are not deleting any points-to set information from any pointer, only adding information and also adding to the finiteness of the number of pointers, we can say that the computation is monotonically increasing.

2. Unordered algorithm - Constraints can be processed in any order. [1]

   We handle conflicting constraints by keeping multiple copies of the conflicting variables and their associated points-to sets. For instance, the constraints a = b, a = c and a = d conflict. However, if they are analyzed in separate threads Thread1, Thread2 and Thread3 respectively, this method creates a copy of a's points-to set in all the threads since their write sets contain a. The newly computed points-to set information of all the threads is merged with that of the master copy of a at the end of each iteration. It should be noted that it is possible to use multiple copies of variables since the analysis is monotonic. If the analysis is non-monotonic (for instance, if it is flow-sensitive) then naively making multiple copies of the points-to set information may not preserve the solution. The merge operation performs a union of points-to set information for each pointer. Merging of the points-to set information for multiple pointers is done in parallel. However, merging of local points-to set information of a pointer with its master copy is done in a sequential manner.

   The replication of points-to set information is not transparent to the threads. Each thread simply deals with its own copy whenever it modifies data. A thread does not need to know about other threads or the number of copies of the points-to information it accesses in the system.

Replication based Parallel PTA computation

First we arbitrarily partition the points-to constraints. In the example shown in Figure 3.4, the first four points-to constraints are assigned to thread 1, next three are assigned to thread 2 and the last three are assigned to thread 3. For all the pointers whose points-to set information will be updated, we maintain local copy of it's points-to set pertaining to each thread. As shown in figure 3.4, a's points-to set information is updated by each thread in Iteration 1. So, we are going to replicate the points-to set information from the global copy of a before the first iteration to a' in thread 1, a" in thread2 and a"' in thread3. These are different local copies for each thread, replicated from the global copy a. Once we have executed the statements in all the threads, we merge the information from the local replicated copies into the global copy and keep repeating this till the fixed point is reached. Let's see one iteration.

Iteration 1 - Thread 1 : Computation

We process the instructions one by one in sequence and update the points-to set information of the pointers to their local copies.

- q = p : q's local copy q' is updated with p's points-to set which is {a}.
- a = b : a's local copy a' is updated with b's points-to set which is {y}.
- e = a : Since a's local copy a' is available in the thread. e's local copy e' is updated with points-to set information of a', which is {x,y}.
- r = q : Since q's local copy q' is available in the thread. r's local copy r' is updated with points-to set information of q', which is {a}.

Thread 1 accounts for four steps as points-to set information of four pointers are updated.

Iteration 1 - Thread 2 : Computation

Similar to thread 1, we process the instructions in thread 2 one by one in sequence and update the points-to set information of the pointers to their local copies.

- a = c : a's local copy a" is updated with c's points-to set which is {z}.
- s = r : r's points-to set information is empty. So, no new information need to be updated.
- e = *a : a's local copy a" is available in the thread. We take the points-to information of a" which is {x,z}. Since none of the pointees of a" has any points-to information, no new information need to be updated.

Thread 2 accounts for one step as points-to set information of only one pointer is updated.

Iteration 1 - Thread 3 : Computation

Again, we process the instructions in thread 3 one by one in sequence and update the points-to set information of the pointers to their local copies.

- t = s : s's points-to set information is empty. So, no new information need to be updated.
- a = d : a's local copy a"' is updated with d's points-to set which is {w}.
- *e = a : e's points-to set information is empty. So, there are no pointees of e. Hence no update.

Thread 3 accounts for one step as points-to set information of only one pointer is updated.

Merge after Iteration 1

Points-to set information of a', a" and a"' are merged to form new global copy of a for future iterations. Similarly, points-to set information of e', q' and r' forms the new global copies of e, q and r respectively.

We keep repeating this considering the new copies as the new global copies for next iteration and we do this until the fixed point is reached.

Total Computation

- Iteration 1 accounts for 4 steps (Amdahl's Law).
- Iteration 2 accounts for 2 steps (Amdahl's Law).
- Iteration 3 accounts for 3 steps (Amdahl's Law).
- We have 3 merges, each accounting for 1 step.

The parallel computation with 3 threads shown in the figure 3.4 takes 12 steps as opposed to 13 steps in the sequential computation.

Replication based Parallel Points-to Analysis (3 threads)

Thread	Statement	Iteration 1	Merge 1	Iteration 2	Merge 2	Iteration 3	Merge 3
1	q = p	q' → {a}					
	a = b	a' → {y}					
	e = a	e' → {x,y}		e' → {w,x,y,z}			
	r = q	r' → {a}	a → {w,y,z}		e → {w,x,y,z}		w,z → {w,x,y,z}
2	a = c	a" → {z}	e → {x,y}		x,y → {w,x,y,z}		t → {a}
	s = r		q,r → {a}	s' → {a}	s → {a}		
	e = *a						
3	t = s					t' → {a}	
	a = d	a"' → {w}					
	*e = a			x',y' → {w,x,y,z}		w',z' → {w,x,y,z}	

Figure 3.4

References

[1] Rupesh Nasre. Parallel points-to analysis. *Program Analysis*, 2017.