# Program Analysis

Scribes-Week 8

Lecturer Dr Rupesh Nasre Scribes: Mohnish Uyyuru, Varun Chauhan



## Naïve vs. Replication-based

| Pros | Cons |
|---|---|
| No Merging | Costly merging |
| Lesser iterations | More iterations |
| General Purpose | Monotonic, unordered |
| Lower memory requirement | Higher memory requirement |

| Cons | Pros |
|---|---|
| Costly conflict-detection | No conflict-detection |
| Limited parallelism | Adaptive parallelism |
| Unbalanced load | Better load-balancing |
| Lower parallel performance | Better parallel performance |

# An example of the replication based approach

## Constraints

p = &a
a = &x
b = &y
c = &z
d = &w
q = p
a = b
e = a
r = q
a = c
s = r
e = *a
t = s
a = d
*e = a

# Replication-based Parallel Points-to Analysis: 4 Threads

| T | Stmt | Itr 1 | Merge 1 | Itr 2 | Merge 2 | Itr 3 | Merge 3 |
|---|---|---|---|---|---|---|---|
| 1 | q=p | q'→{a} | | | | | |
| | e=a | e'→{x} | | e'→{y,z,w} | | | |
| 2 | r=q | | | r'→{a} | | | |
| | a=c | a''→{z} | a→{y,z,w} | | e→{y,z,w} | | y,z,w→{x,y,z, w} |
| 3 | e=*a | | e→{x} | | x→{x,y,z, w} | | s,t→{a} |
| | s=r | | q→{a} | | r→{a} | | |
| | t=s | | | | | s',t'→{a} | |
| | a=d | a'''→{w} | | | | | |
| 4 | *e=a | | | | | Y',z',w'→ {x,y,z,w} | |
| | a=b | a'→{y} | | | | | |

# In iteration 1:

*In thread 1*, we have

q=p, we know that p→{a}. So, q' →{a}.

e=a, we know that a →{x}. So, e'→{x}.

*In thread 2*, we have

r=q, we know that q→{}. So r'→{}.

a=c, we know that c→{z}. So, a''→{z}.

*In thread 3*, we have

e →*a, we know that a→{x} and x→{}. So e'''→{}.

s=r, we know that r→{}. So, s'→{}.

t=s, we know that s→{}. So, t'→{}.

a=d, we know that d→{w}. So, a'''→{w}.

*In thread 4*, we have

*e=a, we know that e''→{}.

a=b, we know that b→{y}. So, a'→{y}.

## Merging

In every merge, thread can merge local copies of every single variable. After iteration 1, we do the **merging**. Now after merging a→{y,z,w}, e→{x}, q→{a}.

We repeat this process, until we get the fixed point.

## Total Steps Taken :

In first iteration, it takes 2 steps because we got two statements from the thread 1, which is maximum of all the threads.
In second iteration, it takes 1 steps because there is one statement in all threads.
In third iteration, it takes 3 steps because we got three statements from the thread 4, which is maximum of all the threads.

And there are 3 Merges. For every merge, we can count it as a single step because every thread can merge local copies of every single variable. So if no of variables are equal to no of threads then it will take atmost 1 step. So, conservatively we count it as a 1 step. So, it takes 3 steps.

So, total no of steps taken are **9 steps**.

# Replication-based Parallel Points-to Analysis: 5 Threads

| T | Stmt | Itr1 | Merge1 | Itr2 | Merge2 | Itr3 | Merge3 |
|---|---|---|---|---|---|---|---|
| 1 | e=a | e→{x} | | e'→{y,z,w} | | | |
| 2 | q=p | q'→{a} | | | | | |
| 3 | r=q | | | r'→{a} | | | |
| | a=c | a''→{z} | a→{y,z,w} | | e→{y,z,w} | | y,z,w→{x,y,z,w} |
| 4 | e=*a | | e→{x} | | x→{x,y,z,w} | | s,t→{a} |
| | s=r | | q→{a} | | r→{a} | | |
| | t=s | | | | | s't'→{a} | |
| | a=d | a'''→{w} | | | | | |
| 5 | *e=a | | | x'→{x,y,z,w} | | y'z'w'→{x,y,z,w} | |
| | a=b | a'→{y} | | | | | |

## In iteration 1:

*In thread 1*, we have

e=a. We know that a→{x}. So e'→{x}.

*In thread 2*, we have

q=p. We know that p→{a}. So q'→{a}.

*In thread 3*, we have

r=q, we know that q→{}. So r'→{}.

a=c, we know that c→{z}. So a''→{z}

*In thread 4*, we have

e →*a, we know that a→{x} and x→{}. So e''→{}.

s=r, we know that r→{}. So, s'→{}.

t=s, we know that s→{}. So, t'→{}.

a=d, we know that d→{w}. So, a'''→{w}.

*In thread 5*, we have

*e=a. We know that e'''→{}.

a=b, we know that b→{y}. So, a' →{y}.

## Merging

After, every iteration we do the merging. In every merge, thread can merge local copies of every single variable.

After iteration 1, we do the **merging**. Now after merging, we get a→{y,z,w}, e→{x}, q→{a}.

## Total Steps Taken:

In first iteration, it takes 1 step because there is one statement in all the threads.
In second iteration also, it takes 1 step because there is one statement in all the threads.
In third iteration, it takes 3 steps because we got three statements from the thread 5, which is maximum of all the threads.
And there are 3 Merge. For every merge, we can count it as a single step. So, total no of steps taken are **8 steps**.

| |
|---|
| In sequential it takes 13 steps,<br>in parallel it takes 9 steps,<br>in replication with 3 threads, it takes 12 steps ,<br>in replication with 4 threads it takes 9 steps and<br>in replication with 5 threads it takes 8 steps |

In naive method, we are not able to divide the constraints into maximum possible number of threads. But in replication based approach, we can divide the constraints in to maximum possible no of threads.

# Optimizations

Now, we discuss key optimisations which improve the overall parallelism of the analysis.

## Load Balancing

Replication allows an arbitrary distribution of constraints to threads. However, to achieve good performance, proper load-balancing of work is necessary. Unfortunately, the amount of points-to information propagated from one pointer to another differs significantly across pointers and iterations. Therefore, a static constraint partitioning, which assigns a constraint evaluation to a fixed thread throughout the analysis, achieves only limited success. On the other extreme, re-calculating the partitions for a perfect load-balance makes the analysis slower than no load-balancing at all! Therefore, we employ a greedy, incremental approach, which achieves an approximately load-balanced threads at a much-reduced cost.

Our algorithm first distributes copy and load constraints to threads in an even (round-robin) manner, since both kinds of constraints have a singleton write-set. It then makes a single pass over the (costly) store constraints to distribute those to threads again in an even manner. The store constraint may update the points-to sets of multiple pointers. Each thread also maintains a single number indicating its load (amount of work), based on the constraints assigned.

In each iteration, as a constraint evaluation results in new points-to information, each thread updates its load-indicator, keeping track of how much information each constraint changed in that iteration. If the new load-indicator is more than its value in the previous iteration by a threshold (pre-determined based on the number of constraints and threads), the thread orphans a few constraints that added the maximum points-to information and adds those to a shared worklist. Other threads, at the end of each iteration, check this worklist and adopt a few constraints if their load-indicator is less than the threshold.

Note that checking for overload is a (thread-) local strategy, which avoids costly thread-communication overhead. The strategy works reasonably well in practice because the amount of new points-to information added by a constraint in each iteration can be predicted based on that in the previous iteration.

## Parallel Online Cycle Elimination

An important property of the pointers in a cycle is that all of them (eventually) have the same points-to information. Therefore, to reduce unnecessary propagations, cycles are collapsed.

Cycle elimination involves replacing the nodes in the cycle by a representative node with its points-to information as a union of the points-to information from all the replaced nodes. Further, the incoming edges to and the outgoing edges from the replaced nodes need to be updated to be to and from the representative node respectively. Our algorithm collapses disjoint cycles in parallel. We reuse the same threads as for solving constraints to collapse cycles, since cycle detection and collapsing is done when no threads are solving any constraints.

It is possible for two threads collapsing disjoint cycles to update the incoming edges from the same node, potentially resulting in a conflict. However, this happens infrequently and therefore, we use locking over the nodes to be updated while collapsing cycles.

## Reducing Replication Cost

Reducing the number of copies also reduces the number of iterations to reach the fixed-point.

It is unnecessary to make a copy of a variable's points-to set when there is a single writer thread. We detect this situation by maintaining the number of writer threads for each variable and using directly the master copy when no more than one thread writes to the variable.

## Limited Scheduling

The amount of new points-to information computed is high in the initial iterations and gradually reduces as the analysis progresses. In fact, towards the end of the analysis, only a few constraints add more points-to information.

REFERENCES

[1] Sandeep Putta, Rupesh Nasre Parallel Replication-based Points-to Analysis

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