Dependence Analysis for Parallelization of Sequential Programs

Sandeep Dasgupta, Barnali Basak, Amey Karkare

State-of-the-Art

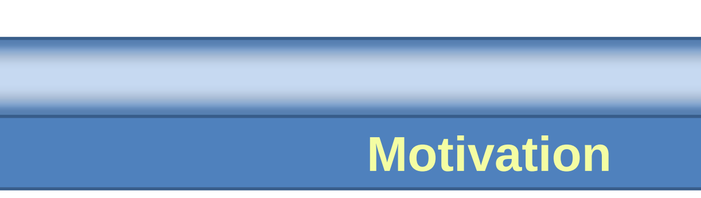
Automatic Parallelization Of Sequential Programs (using *Compile Time Analysis*) :

- Static data structures (static arrays or stack directed pointers).

```
for(i = 0; i <= n; i++) {
   ...
   A[i + 1] = A[i]
   ...
}
```

```
A[1] =  A[0] /* Iteration 1 */
        flow dependence
A[2] =  A[1] /* Iteration 2 */
        flow dependence
A[3] =  A[2] /* Iteration 3 */
```

- Dynamic memory (Heap) structures

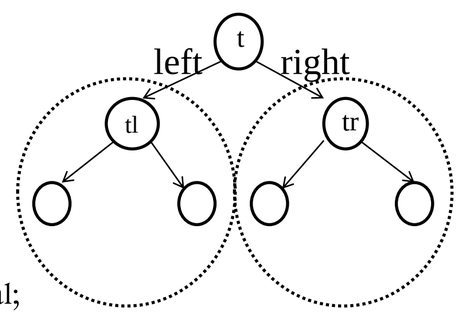


Challenges!!

Heap structure
- Unknown at compile time
- Potentially Unbounded
- Not limited by the creating scope.

Application

```
void treeAdd(tree * t)
{
    if(NULL == t)
        return;
    tl = t→left;
    treeAdd(tl);
    tr = t→right;
    treeAdd(tr);
    t→val = tl→val + tr→val;
}
```



Observation :
- Computation of *t* depends on that of *tl* and *tr*.
- Computation of *tl* and *tr* can be done in parallel.
- With n processing units (n→ number of tree nodes), time required = *O(log n)*

Dependence Detection

Path Abstraction :

n + *2* Field Array, *n* = number of data store fields

Associated Pointer	# Field 1	#Field 2	...	#Field n	Index of first ponter field (1 ≤ i ≤ n)

Double linked list : two pointer fields as next, prev
one data field as num
path p →next →next →prev abstracted as <p, 2(#next), 1(#prev), 0(#num), 1> .

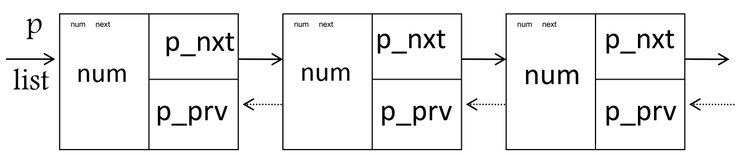
	Associated Pointer	Tag	next	prev	num	Index	Tag	next	prev	num	Index
p=list;	p		0	0	0	0					
while()				Iter1					Iter2		
temp=p->num;	p	Read	0	0	1	0	Read	1	0	1	1
if(cond){											
p_nxt = p->next	p_nxt		1	0	0	1		2	0	0	1
p_nxt->num=temp;	p_nxt	Write	1	0	1	1	Write	2	0	1	1
else{											
p_prv=p->prev;	p_prv		0	1	0	2		1	1	0	1
p_prv->num=temp;	p_prv	Write	0	1	1	2	Write	1	1	1	1
p=p->next;	p		1	0	0	1		2	0	0	1

Paths (p)<0,0,1,0,0> conflicts with (p_prv)<1,1,1,1,1>
& (p_nxt)<1, 0, 1, 1> conflicts with (p)< 1, 0, 1, 1 >
as inferred by the **path alias information**

Motivation

Given a **heap traversal code** and a **heap data strcuture**

```
p=list;
while(p→next != NULL){
S1:   temp = p→num;
      if(cond){
S2:     p_nxt =  p→next;
S3:     p_nxt→num = temp;
      }else{
S4:     p_prv = p→prev;
S5:     p_prv→num = temp;
      }
S6:   p = p→next; }
```

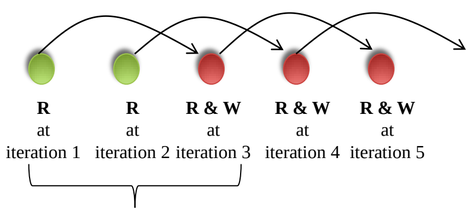


Can we Parallelize the above heap traversal code

Stopping Criteria

Max Depth of Dependence

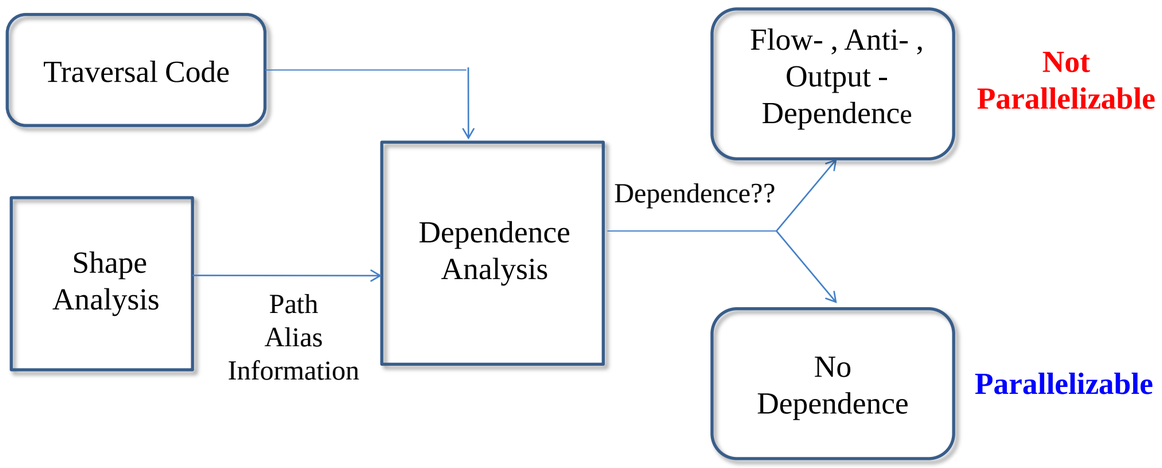
```
p=list;
while(p→next != NULL){
S1:   q = p;
S2:   ... = p→num;
S3:   q = q→ next;
S4:   q = q→ next;
S5:   q->num = ...;
S6:   p = p→next;  }
```




```
p=list;
while(p→next != NULL){
S1:   q = p;
S2:   ... = p→num;
S3:   while(i ≤ n) {
S4:     q = q→ next;
S5:     i++;
S6:   }
S5:   q->num = ...;
S6:   p = p→next;  }
```

Cannot Determine Max Depth Of Dependence ➔ **Conservative Descision (Not Parallelizable)**

How To??



- Two paths are alias if they access same node.
- Dependence analysis detects conflicting node accesses.

Conclusion And Future Work

- Dependence detection technique depends on feasiblity of depth of dependence computation.
- *Plug and Play* shape analysis framework.
- Fast dependence analysis; No shape graph manipulation.

Current Activity:
Implementing a prototype of our analysis for a subset of JAVA using SOOT framework.

Future Plans:
To support complete JAVA and show effectiveness on large benchmarks

References

1. Detecting Parallelism in C Programs with Recursive Data Structures,. R. Ghiya, L.J. Hendren and Yingchun Zhu.
2. A new Dependence Test based on Shape Analysis for pointer-based Codes,. A.Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plate and E.L. Zapata.
3. Heap reference analysis using access graphs , Khedker Uday P. and Sanyal, Amitabha and Karkare, Amey.

Indian Institute of Technology, Kanpur, India