CS6843-Program Analysis **Jan-May 2017**

Scribe Week #10

Lecturer: Prof. Rupesh Nasre *Mani Raja & Puneet Saraf*

1 ILP (Contd.)

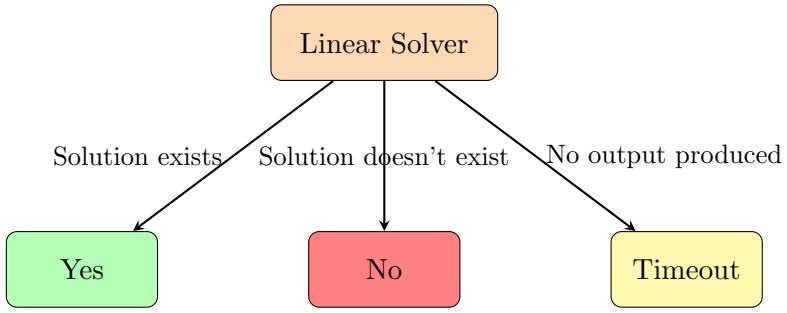
In the previous scribe, we have seen forming dependency constraints in the form of linear inequalities. But what actually is the connection between these inequalities formed and the dependencies? We have formed the inequalities based on these three conditions

1. Indices constraints
2. Same memory location constraints
3. Dependency constraints

If we are able to satisfy all the linear equalities/inequalities that means these constraints hold true, hence the dependency is there in the program.

1.1 Possible solution of Linear Solver

There can be three possible solutions, while solving the linear inequalities



Solution exists: This signifies that all inequalities have been satisfied by linear solver, which means the mentioned dependency exists.

Solution doesn't exist: This signifies that all inequalities cannot be satisfied at the same time, hence the dependency doesn't exist. So, we can apply further optimization.

Timeout: This signifies that we are not able to solve for the mentioned inequalities within a time limit (we will be having specific time bound for linear solver). In such cases, we need to approximate the linear solver output. So, to be on safer side, we apply a conservative analysis and say that dependency exists. If we say the other possibility (i.e. dependency doesn't exist) and apply some optimization over it, then it may lead to an unsound analysis.

1.2 Examples from Slides

Example 1: For checking Anti-dependency using ILP

```
for(ii=0; ii<10; ++ii)
  {
    a[2*ii]=...a[ii+1]...
  }
```

We have to find, if there is an anti-dependence between different iterations.

$$0 <= ii_r < ii_w < 10 \tag{10.1}$$
$$2 * ii_w = ii_r + 1 \tag{10.2}$$

Decomposing the above equations

Indices constraints

$$0 <= ii_r \tag{10.3}$$
$$ii_w <= 9 \tag{10.4}$$

Anti-dependency *constraint*

$$ii_r <= ii_w - 1 \tag{10.5}$$

Same memory location constraint

$$2 * ii_w <= ii_r + 1 \tag{10.6}$$
$$ii_r + 1 <= 2 * ii_w \tag{10.7}$$

The equations from 10.3 to 10.7 can be written in the form of a matrix to solve for ii_r and ii_w.

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \\ 1 & -1 \\ -1 & 2 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} ii_r \\ ii_w \end{bmatrix} <= \begin{bmatrix} 0 \\ 9 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

In the above example for no values of ii_r and ii_w these all inequalities are satisfied. Hence, we can say that no anti-dependency exists in the above program.

Example 2: For checking true dependency using ILP

```
for(ii=0; ii<10; ++ii)
  {
    a[2*ii]=...a[ii+1]...
  }
```

We have to find, if there is a true dependence between different iterations.

$$0 <= ii_w < ii_r < 10 \tag{10.8}$$
$$2 * ii_w = ii_r + 1 \tag{10.9}$$

Decomposing the above equations

Indices constraints

$$0 <= ii_w \tag{10.10}$$

$$ii_r <= 9 \tag{10.11}$$

True dependency constraint

$$ii_w <= ii_r - 1 \tag{10.12}$$

Same memory location constraint

$$2 * ii_w <= ii_r + 1 \tag{10.13}$$

$$ii_r + 1 <= 2 * ii_w \tag{10.14}$$

The inequalities from 10.10 to 10.14 can be written in the form of a matrix to solve for ii_r and ii_w.

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ -1 & 1 \\ -1 & 2 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} ii_r \\ ii_w \end{bmatrix} <= \begin{bmatrix} 0 \\ 9 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

ii_r	ii_w
0	–
1	–
2	–
3	2
4	–
5	3
6	–
7	4
8	–
9	5

By solving these inequalities, we get 4 different values of ii_r and ii_w for which the inequalities holds(green rows) Hence, there exist true dependency in the above program.

Till now we have seen cases, when there is a single statement inside the for loop. Let us now see an example with multiple statements.

Example 3: For checking true dependency between multiple statements using multiple ILP

```
for(ii=0; ii<10; ++ii)
  {
    a[2 * ii] = ... a[ii + 1] ...
    a[3 + ii] = ... a[5 * ii] ...
  }
```

Since, there are multiple statements in the loop more than one true dependency is possible. We will have to model equations across all inter-iteration pairs of reads/writes. Possible pairs for true dependencies are

1. 2* ii and ii + 1
2. 3 + ii and 5 * ii
3. 2 * ii and 5 * ii

4. 3 + ii and ii + 1

To check for dependency across all possible pairs, we need to form 4 different ILP because there is an OR condition between the pairs.

Indices and True dependency constraints are same across ILP1-4, but *Same memory location constraints* will be different.

Indices constraints

$$0 <= ii_w \tag{10.15}$$

$$ii_r <= 9 \tag{10.16}$$

True dependency constraint

$$ii_w <= ii_r - 1 \tag{10.17}$$

Same memory location constraints for ILP1

$$2 * ii_w <= ii_r + 1 \tag{10.18}$$

$$ii_r + 1 <= 2 * ii_w \tag{10.19}$$

Same memory location constraints for ILP2

$$3 + ii_w <= 5 * ii_r \tag{10.20}$$

$$5 * ii_r <= 3 + ii_w \tag{10.21}$$

Same memory location constraints for ILP3

$$2 * ii_w <= 5 * ii_r \tag{10.22}$$

$$5 * ii_r <= 2 * ii_w \tag{10.23}$$

Same memory location constraints for ILP4

$$3 + ii_w <= ii_r + 1 \tag{10.24}$$

$$ii_r + 1 <= 3 + ii_w \tag{10.25}$$

There are 2 more possible pairs viz. 2 * ii and 3 + ii, ii + 1 and 5 * ii but these are WAW and RAR respectively, RAR is not a dependency and WAW is not our concern here because above we were asked about the RAW dependency only.

1.3 Extra examples discussed in class

Example 4: For checking WAW dependency using multiple ILP

```
for(i=0; i<10; ++i)
{
   for(j=0; j<10; ++j)
   {
      a[2*i+3j-5]=.....
         .......=a[4*i-8j];
   }
}
```

We have to find, if there is a WAW dependence between different iterations.
Now there are 3 possible ways in which we can parallelize the above code

1. Parallelizing across both iteration variables i and j.
2. Parallelizing across only outer iteration variable i.
3. Parallelizing across only inner iteration variable j.

Case 1: Parallelizing across both iteration variables i and j.
It means if we have sufficient number of threads available with us then we can execute all the iterations in parallel at a time. Here, the number of possible iterations is 10*10=100, so if we know that there is no dependency among them then we can parallelize all 100 iterations.

Indices constraints

$$0 <= i_{w1} <= 9 \tag{10.26}$$

$$0 <= i_{w2} <= 9 \tag{10.27}$$

$$0 <= j_{w1} <= 9 \tag{10.28}$$

$$0 <= j_{w2} <= 9 \tag{10.29}$$

WAW dependency constraint

$$i_{w1} <= i_{w2} - 1 \tag{10.30}$$

OR

$$i_{w1} == i_{w2} \ \ and \ \ j_{w1} <= j_{w2} - 1 \tag{10.31}$$

Same memory location constraint

$$2 * i_{w1} + 3 * j_{w1} - 5 <= 2 * i_{w2} + 3 * j_{w2} - 5 \tag{10.32}$$

$$2 * i_{w2} + 3 * j_{w2} - 5 <= 2 * i_{w1} + 3 * j_{w1} - 5 \tag{10.33}$$

Case 2: Parallelizing across only outer iteration variable i.
In this case, since we are parallelizing only among outer loop variable i, then if there is no dependency among outer iterations then we can parallelize all 10 iterations of i but for each outer iteration, all its inner iteration operations will happen in sequential order in a single thread.

In this case all the constraints will remain same except there will be some change in the dependency constraints.
In the case 1 dependency constraints, 10.31 will be removed.

WAW dependency constraint

$$i_{w1} <= i_{w2} - 1 \tag{10.34}$$

Case 3: Parallelizing across only inner iteration variable j.
In this case, since we are parallelizing only among inner loop variable j, then all iterations of outer variable i will happen in sequential order (*i.e.* if i=1 iteration is happening then i=2 will only start when i=1 finishes its execution) but for a single iteration of i, if there is no dependency among its inner iteration variable j then all 10 operations of its inner iteration can happen in parallel.

In this case all the constraints will remain same except there will be some change in the dependency constraints.
In the case 1 dependency constraints, 10.30 will be removed.

WAW dependency constraint

$$i_{w1} == i_{w2} \;\; and \;\; j_{w1} <= j_{w2} - 1 \tag{10.35}$$

Example 5: For checking true dependency using multiple ILP

```
for(i=0; i<10; ++i)
{
   for(j=i+1; j<10; ++j)
   {
      a[3*i+4j+1]=.....
         .......=a[9*i+j-8];
   }
}
```

We have to find, if there is a true dependence between different iterations.
Now there are 3 possible ways in which we can parallelize the above code

1. Parallelizing across both iteration variables i and j.
2. Parallelizing across only outer iteration variable i.
3. Parallelizing across only inner iteration variable j.

Explanation for all the cases of this example is same as the previous one. Only the constraints for all the cases are shown below:

Case 1: Parallelizing across both iteration variables i and j.

Indices constraints

$$0 <= i_w <= 9 \tag{10.36}$$
$$0 <= i_r <= 9 \tag{10.37}$$
$$i_w + 1 <= j_w <= 9 \tag{10.38}$$
$$i_r + 1 <= j_r <= 9 \tag{10.39}$$

True dependency constraint

$$i_w <= i_r - 1 \tag{10.40}$$

OR

$$i_w == i_r \;\; and \;\; j_w <= j_r - 1 \tag{10.41}$$

Same memory location constraint

$$3 * i_w - 4 * j_w + 1 <= 9 * i_r + j_r - 8 \tag{10.42}$$
$$9 * i_r + j_r - 8 <= 3 * i_w - 4 * j_w + 1 \tag{10.43}$$

Case 2: Parallelizing across only outer iteration variable i.
In this case all the constraints will remain same except there will be some change in the dependency constraints.
In the case 1 dependency constraints 10.41 will be removed.

$$\begin{aligned} &\textit{True dependency constraint} \\ &i_w <= i_r - 1 \end{aligned} \tag{10.44}$$

Case 3: Parallelizing across only inner iteration variable j.
In this case all the constraints will remain same except there will be some change in the dependency constraints.
In the case 1 dependency constraints 10.40 will be removed.

$$\begin{aligned} &\textit{True dependency constraint} \\ &i_w == i_r \;\; \textit{and} \;\; j_w <= j_r - 1 \end{aligned} \tag{10.45}$$

2 Managing Races

2.1 What is a Data Race?

A Data race occurs during the execution of a multi-threaded process. Below are the conditions for data race:

- two or more threads in a single process access the same memory location concurrently, and
- at least one of the accesses is for writing, and
- the threads are not using any exclusive locks to control their accesses to that memory.

When these three conditions hold, the order of accesses is non-deterministic, and the computation may give different results(inconsistent) from run to run depending on that order.

A Critical Section is a code segment that accesses shared variables and has to be executed in mutually exclusive manner. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.

Example for Data Race:
Let us assume that two threads want to increment the value of a global integer variable by one.

```
Counter++;
```

even though it looks like a single statement but actual execution was split into three statements.

1. Read counter
2. increment counter
3. store updated value of counter

Ideally, the following sequence of operations would take place as shown below:

Thread 1	**Thread 2**		**Integer Value**
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

In the case shown above, the final value is 2, as expected. However, if the two threads run simultaneously without locking or synchronization, the outcome of the operation could be wrong. An alternative sequence of operations below demonstrates this scenario:

Thread 1	**Thread 2**		**Integer Value**
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

In this case, the final value is 1 instead of the expected result of 2. This occurs because here the increment operations are not atomic.

There are multiple Synchronization mechanisms to avoid data race, these are:

- Locks
- Atomics
- Barriers

2.2 Inserting Locks

A lock is an abstraction that allows at most one thread to own it at a time. Holding a lock is how one thread tells other threads: "I'm changing this thing, don't touch it right now."

Data-race between iterations p and q for element a[f(i)].

```
if(i==p || i==q){
  lock(f(i));
  ...perform operation...
  unlock(f(i));
}
```

Example where locks are required to maintain a mutual exclusion between the threads.
Producer-consumer problem

```
produce(){
  while(...){
    item.add(...);
  }
}
```

```
consume(){
  e=item.remove();
}
```

In producer-consumer, we have a shared array in which producer produces an item and consumer consumes from it. When there is no lock then inconsistency can happen, the process of producing/consuming an item from the array is not atomic, hence there can be situation when a producer is still in the process of incrementing the number of items in the array but before that consumer consumes an item then there is an inconsistency.

For multiple data items a[f(i)] and a[g(i)], we can use

- Single lock
- Multiple locks

Maintaining mutual exclusion when there are multiple data items can be done with both single lock and multiple locks, but using single lock might compromise with the level of parallelism, on the other hand, using multiple locks has their own disadvantages.
Using multiple locks may lead to

- overhead of maintaining locking and unlocking of multiple locks.
- may even lead to Deadlock.

Deadlock may be allowed if it is improving the parallelism.

Does allowing deadlock preserve the correctness of program?
There is a possibility that functionality of a program may not hold theoretically, but since there is no termination of program then we can ensure *partial* correctness of the program. This notion of correction may differ from system to system, for example in real-time system like flight controlling system, we need to take care of every scenario very precisely, but still we may land up in a partially correct scenario. Partially correct solution may lead to an unsound analysis.
If we try to avoid the deadlock then that may lead to live lock.

What is a live lock?
Live lock is a situation in which if a system has recovered from a deadlock then there is a high risk that it may again lead to a deadlock situation. Let us take an example to clarify, if there are two threads **T1** and **T2** and currently they have acquired lock l_1 and lock l_2 respectively, but to proceed

they both need l_1 and l_2 locks. So, to overcome the deadlock, one of the threads(T1) releases it's lock, so that it can be used by the other(T2) but at the same time T2 also releases its lock and again they both try to acquire the lock l_1 and l_2 but again they land up in the initial stage *i.e.* T1 with l_1 lock and T2 with l_2 lock. This scenario is known as live lock.

What is a mutex variable?
Mutex is a lock variable which is used to maintain mutual exclusion between two threads. The operations on mutex variable are atomic *i.e.* if mutex is changed from 0 to 1 then this will be done with indivisible operation. But there are scenarios when using a mutex variable is not enough, if we want to allow a fixed number of threads inside a critical section then we cannot use mutex variable. Mutex variable do not allow two or more threads for read operation at same time which is a dependency free operation, for this we have to use a read-write lock.

Sometimes, a lock may be used for a simple operation as shown below

```
if(i==p || i==q){
  lock(f(i));
  sum+=a[i];
  unlock(f(i));
}
```

In the above program we are using a lock for a simple operation of addition, if we can convert this operation into an atomic operation then we don't need a lock which is a bit complex to maintain. The above addition operation can be done without a lock if we have an *atomic_add* to perform the addition. Then we can simply do the add operation as $atomic\_add(\&sum, a[i])$ without acquiring lock on $f(i)$. Atomic_add is an example of an atomic instruction provided by the library. These internally use Test and Set or Atomic CAS which is hardware supported atomic instruction.

```
if(i==p || i==q){
  atomic_add(&sum,a[i]);
}
```

2.3 Inserting Atomics

An operation can be done atomically, if the operation is simple and it is preferably of the following form:

- *Primitive type*: It means that the variable on which the operation is performed must be of a primitive type such as int, float or some simple form.
- *Single element*: It means that there should be a single element on which the operation is being performed, like in the previous $atomic\_add(\&sum, a[i])$, *sum* was the single element on which operation is performed.
- *Relative update/read-write*: It means that there is a single variable on which we are performing both the read and write, like in the previous $atomic\_add(\&sum, a[i])$, we first read sum variable and then write it.

Example of atomic operation: Producer-consumer with single element update.
Types of atomic operations: increment, decrement, add, sub, min, max, exch, CAS.

Does the atomic instruction remove the need of a lock?
No, it doesn't remove the need of locks, it is just that if the functionality of the critical section is simple enough that it can be implemented using atomic instruction then why we should complicate it by using locks. Locks can produce an unbounded wait but on the other hand atomic instructions are bounded wait.

If fairness is guaranteed then can locks be bounded wait?
Yes, then locks can be bounded wait but we can't say that *eventually* the process will come out of critical section, we have to give some bound on it.

What is a CAS instruction?
Compare-and-Swap (CAS) is an atomic instruction used in multi threading to achieve synchronization. It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value.
Example: $old\_value = CAS(\&x, v, v_2)$
If the memory location x has the value v, then replace it by v_2 and return the old value v. Lets suppose there are multiple threads trying to change x from v to v_2(say T_1, T_2, T_3), here only one of the threads will see $\&x$ as old value(v) and changes it to v_2. Now the other two threads don't see the old value, they see the new value. Since CAS is an atomic instruction hence all the three threads can't execute instruction $old\_value = CAS(\&x, v, v_2)$ simultaneously.

2.4 Classwork

Write parallel single linked-list insertion routines using atomics?

Let us suppose we are given a singly linked-list with three nodes with start node pointed by pointer S and the end node pointed by E and there are two threads T_1 and T_2, who want to add node pointed by N_1 and N_2 respectively.
Now there are three different ways to insert these nodes.

1st way: Normal case
Both threads wants to add their node at the end of linked-list, so thread T_1 try to execute $E \rightarrow next = N_1$ and thread T_2 try to execute $E \rightarrow next = N_2$. But any one of the thread will execute 1st and the thread which executes 2nd will replace the changes made by 1st thread. This is a case of Data Race(as seen in section 2.1's data race example).

2nd way: Using locks
If we acquire same lock over the operation of both the threads.

T1	**T2**
$lock(E)$	$lock(E)$
$E \rightarrow next = N_1$	$E \rightarrow next = N_2$
$E = E \rightarrow next$	$E = E \rightarrow next$
$unlock(E)$	$unlock(E)$

Now any one of the thread can acquire the lock and hence both the threads will add their node one by one.

3rd way: Using atomic instruction CAS
Since the operation is simple enough, therefore we can insert the node using atomic instruction CAS.

T1	**T2**
do	do
$\{$	$\{$
$oldp\_next = CAS(E \rightarrow next, null, N_1)$	$oldp\_next = CAS(E \rightarrow next, null, N_2)$
$E = E \rightarrow next$	$E = E \rightarrow next$
$\}while(oldp\_next! = null)$	$\}while(oldp\_next! = null)$

In this case insertion is being done atomically without locks, since there are two statements in the *do-while* loop it seems as if we need a lock over the loop but in this case the insertion will happen correctly by any number of threads. This is because, even if any thread enters the loop when any other thread is still to execute $E = E \rightarrow next$ then that thread will go on looping since $oldp\_next! = null$ will be false until the previous thread executes $E = E \rightarrow next$.

3 Difference between Soundness and Completeness(off the topic)

3.1 Soundness

- An analysis is sound, if the data flow fact computed by our analysis is guaranteed to happen across all runs.
- Generally sound analysis will compute subset of Runtime information.

3.2 Completeness

- An analysis which does not miss any data flow fact that will happen in at least one run then it is said to complete
- Generally complete analysis will compute superset of Runtime information.

3.3 Sound and Completeness in terms of Aliasing relation

- *Must* aliasing is *sound* but not complete because as we can see in $figure1$ that *must* is a subset of Run time information but some more runtime information is there which is not computed by *must* therefore it is not *complete*.
- *May* aliasing is a *complete* but we cannot say anything about its soundness. So, to give soundness using *may* analysis, we can use its complement *i.e. must not*.
- *Must not* is the white region and white + yellow is the region which is *not happening at runtime* .
- Now we can say that *must not* is a subset of the data flow facts which is *not happening at runtime* as shown in $figure2$.
- *Must not* gives us the guarantee that it will not happen across any run, hence we can say that *must not* is a *sound* analysis.

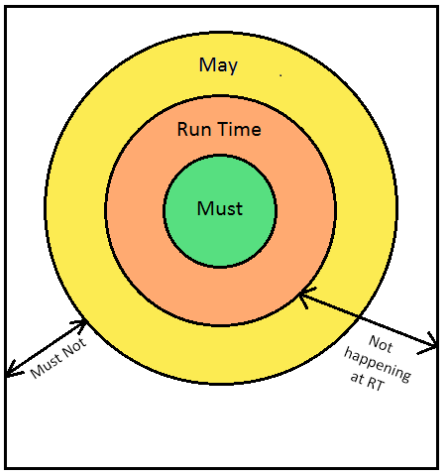


Figure 1

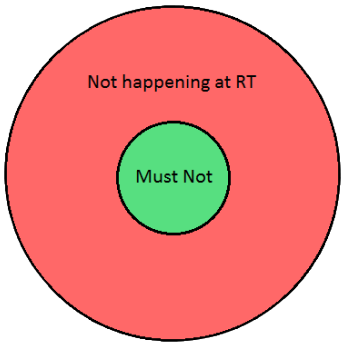


Figure 2

Reference

- Rupesh sir's Class Slides
- www.wikipedia.com