CS6843 Program Analysis
Even Semester 2016-17

IIT Madras
Dept. of Computer Engineering

Week 9: Mar 6, 2017

Lecturer: Rupesh Nasre <rupesh@cse.iitm.ac.in>

Scribe: Shreyas Phanse, Ayush Gupta

9.1 Parallelization

By default, in any program there is always hardware defined parallelization i.e. pipelining. But Hardware is not aware of the higher level semantics of the language.

Consider,

```
p = malloc(...)
q = malloc(...)
```

Is it possible to execute these instruction in parallel?

Answer:

1. Yes, if we can ensure that malloc will give different addresses. There is an internal state(of disk) accessed by both malloc. We must ensure that both malloc return different addresses. One way in which this can happen is diffent threads in the program access disjoint memory spaces. In this case there will be no data race and it will be safe to parallelize.
2. No, if malloc is operating on free list.

If we parallel than than both can get the same address. So we will have to use synchronization. Yes, if both the threads use the Different local memory and hence there is no problem of data race.

9.1.1 Semantics and Parallelization

It is possible that paralleliztion is possible because of semantics of the program, but the hardware doesn't know it. And therefore it cannot hint the hardware to take advantage of parallelization. So, programmer should give all the hints of parallelization.

consider the code:

```
for(int i=0;i<n;i++)
{
    for(int j=0;j<n; j++)
    {
        S1;
        }
    }
```

Normally, here we would expect sequential execution of code. However, the semantics of the program may allow us to execute the loop in parallel. for example:

```
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n; j++)
        {
            a[i][j]=0;
        }
    }
```

Here, the elements of an array are being set to 0. By the semantics, we can easily say that the code is parallelizable.

Question: Are there cases where sequence of $rand()$ matter?
Answer: Yes, for the user of $rand()$ the sequence of rand might not mater. But for the writer of $rand()$ function, sequence of execution matters.

9.1.2 Auto Parallelization

Auto parallelization is the process of relieving human involvement in parallelization. The goal of automatic parallelization is to relieve programmers from the hectic and error-prone manual parallelization process.

9.2 Speed Up

Definition 9.1. *Speedup of a parallel execution is defined as:*

$$Speedup = t_s/t_p \tag{9.1}$$

where t_s is the overall sequential time and t_p is the overall parallel time.
note: Speedup can be less than 1.

9.2.1 Amdahl's Law

The theoretical speedup is always limited by the part of the task that cannot benefit from the improvement($i.e.$ the sequential part).

Question: There are 100 instructions in the program. there are 4 threads in the program with 10, 20, 30, 40 instructions respectively. Calculate the bound on speedup.
Solution: Let t be the time to execute 1 instruction. Applying Amdahl's law on each thread,

$$speedup \leq 100t/10t \leq 10 \tag{9.2}$$

$$speedup \leq 100t/20t \leq 5 \tag{9.3}$$

$$speedup \leq 100t/30t \leq 3.34 \tag{9.4}$$

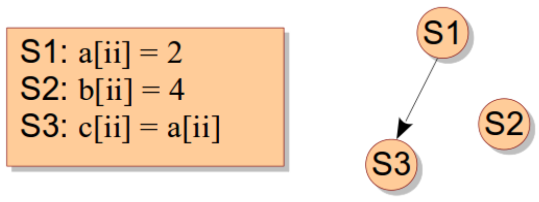
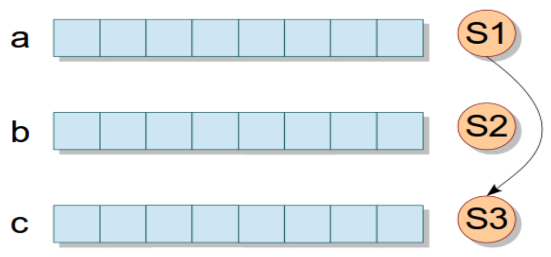
$$speedup \leq 100t/40t \leq 2.5 \tag{9.5}$$

Taking intersection,

$$speedup \leq 2.5 \tag{9.6}$$

9.3 Instruction Parallelism vs Data parallelism

Table 9.1: Comparison between instruction and data parallelism

Instruction	Data
Parallelism extracted from multiple instructions on the data items. 	Parallelism extracted from the same task on different data items. 

9.4 Control Dependence

Consider the code:

```
if(x==4)
{
    y=10;
}
else
{
    y=1;
}
```

We say that $y = 10$ is control dependent on $x == 4$.. We are concerned about control dependence because amount of parallelization will depend on control dependence also. For example here, $y = 10$ and $y = 1$ are control dependent on $x == 4$, so we cannot execute them in parallel.

9.5 Data Dependence

A data dependency is a situation in which a program statement (instruction) refers to the data of a preceding statement. For example:

```
const float pi=3.14;
float r=3.0;
float area=pi*r*r;
```

Here there is data data dependency between $area$, pi and r.

9.5.1 Types of data dependencies

1. True/Flow/RAW(read after write) dependence: **S1** δ **S2**
   These are the dependencies of the type:
   x=...;
   ...=x;

2. Anti/WAR(write after read): **S1** δ^{-1} **S2**
   These are the dependencies of the type:
   ...=x;
   x=...;

3. Output/WAW(write after write): **S1** δ^{0} **S2**
   These are the dependencies of the type:
   x=...;
   x=...;

Using various techniques we can parallelize different dependencies. register renaming can be used to eliminate Anti and Output dependencies. For example: in our example of WAW dependency, the second instance of x can be renamed to (say) x_1 and rename all the later variables to access x_1. True dependency cannot be avoided using register renaming.

9.6 Program Order and Instruction reordering

consider the code:

```
for(int i=0; i<10; i++)
{
    a[i] = 0;
}
```

Here sequential order imposed by programmer is too restrictive. Only partial flow needs to be satisfied here, we don't need to follow the complete flow.
Now consider:

```
for(int i=0; i<10; i++)
{
    a[i] = a[i-1] + 1; //data dependence
}
```

In this case, we cannot parallelize.
Hence, ***Reorder flow, Maintain dependence.***

9.6.1 Importance of Reordering

- Improved locality
  1. Spatial: matrix operations
  2. Temporal: xinit(); yinit(); xcompute(); ycompute();
- Improved load balance.
  small1(); big1(); small2(); big2();
- Improved parallelism.
  xuse(); xdef(); yuse(); ydef();

9.7 Iteration Vector

Consider the code:

```
for(int i=0; i<10; i++)
{
    for(int j=0; j<10; j++)
    {
        for(int k=0; k<10; k++)
        {
            a[i][j][k] = 0;
        }
    }
}
```

Definition 9.2. *Assuming that the variables i, j and k are normalized, the spaced formed by the variables i, j and k as axes, is called as iteration space. and $[i, j, k]$ forms an iteration vector. A point (S) in the iteration space denotes an iteration for the given loop. The variables (dimensions) in iteration vector should be written from outermost loop to innermost loop in that order.*

An iteration space is illustrated in figure 9.1

Let S be any arbitrary point in the iteration space then, in the previous code, if $S(k)$ and $S(k-1)$ are dependent, then we can parallelize upto n^2 only (outer 2 loops).
In general,

Definition 9.3. *$S(\vec{i})\ \delta\ S(\vec{j})$ iff (where δ is depends on):*

1. *$i < j$ or (i == j and $S1 \rightarrow\rightarrow\rightarrow S2$ path in the loop body)*
2. *Both access the same memory location*
3. *At least one of the accesses is a write*

If we know the dependence between $S(\vec{i})$ and $S(\vec{j})$, we can parallelize

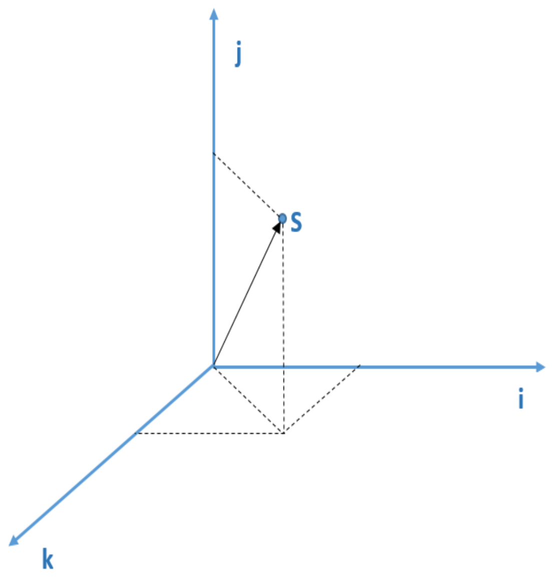


Figure 9.1: Iteration Space

9.8 Loop Transformtions

consider the code:

```
for(int i=0....)
{
    a[i] = 0;
    for(int j=0...)
    {
        S1;
    }
}
```

In order to parallelize this loop, we will have to ***transform*** the loop into the following one i.e. perform loop transformation.

```
 for(int i=0....)
    a[i] = 0;

 for(int i=0...)
    {
        for(int j=0...)
        {
            S1;
        }
    }
}
```

9.8.1 Safe Transformation

Theorem 9.4. ***Loop Dependence Theorem***
There exists a dependence from statement S1 to statement S2 in a common nest of loops iff there exist two iteration vectors $\vec{i}$ and $\vec{j}$ for the nest, such that S1($\vec{i}$) δ S2($\vec{j}$).

Two computations are said to be ***equivalent*** if on the same inputs they produce the same output. For example, programs that sort an array are all equivalent as for a given input, all of them produce the same output.
A transformation is ***safe*** if it leads to an equivalent program. For example, the transformation shown in this section is a safe transformation.

9.8.2 Reordering Transformation

Definition 9.5. *A reordering transformation is any program transformation that merely changes the execution order of the code, without adding or deleting any executions of any statements.*

A reordering transformation preserves a dependence if it preserves the relative execution order of the source and the sink of that dependence. The following example shows two codes reordering transformation preserves dependence:

```
int a = 5;          |     int a = 5;
int b = 6;          |     int b =6;
int c = a + 10;     |     int d = b-3;
int d = b - 3;      |     int c = a + 10;
```

Theorem 9.6. *Any reordering transformation that preserves every dependence in a program leads to an equivalent computation.*

9.8.3 Valid Transformations

Definition 9.7. *A transformation is valid to the program it applies if it preserves all the dependencies in the program.*

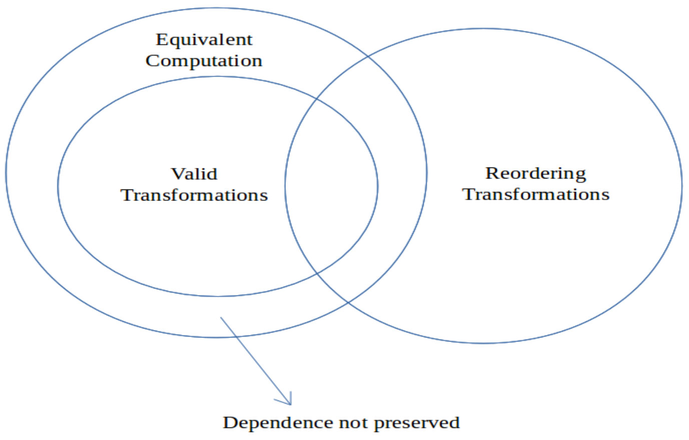


Figure 9.2: Relation between Equivalence Computation, Valid Transformations and Reordering Transformations

Question: Write a simple transformation that maintains computation equivalence but does not preserve dependence (try as small code as possible).
Solution:

```
            int b = 5;          |          int b = 5;
            int a = 5;          |          int a = 5;
            int b = a;          |          int c = b;
            int c = b;          |          int b = a;
```

9.9 Loop Parallelization

Most of the time taken for execution of a program will be taken inside a loop. Hence loop parallelization is very important, which will be discussed in this section.

Theorem 9.8. *It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence. (also refer to Definition 8.3)*

The following loop can be paralleized:

```
for(int i = 0; i<n; i++)
{
    a[i] = b[i];
    b[i] = a[i] + 1;
}
```

This loop cannot be parallelized:

```
for(int i = 1; i<n; i++)
{
    a[i] = a[k-1];
}
```

9.9.1 General Strategy

Consider the code:

```
for(ii = 1; ii<n; ii++)
{
    for(jj =0; jj<n ; jj++)
    {
        a[f(ii,jj)][g(ii,jj)] = .....
         ..... = a[h(ii,jj)][k(ii,jj)]
    }
}
```

Our task is to identify dependencies. Following are the conditions for flow dependence from iteration (ii$_w$, jj$_w$) to (ii$_r$ to jj$_r$) :

$$0 \leq ii_w < n \tag{9.7}$$

$$0 \le jj_w < m \tag{9.8}$$

$$0 \le ii_r < n \tag{9.9}$$

$$0 \le jj_r < m \tag{9.10}$$

$$(ii_w, jj_w) \le (ii_r, jj_r) \tag{9.11}$$

$$f(ii_w, jj_w) = h(ii_r, jj_r) \tag{9.12}$$

$$g(ii_w, jj_w) = k(ii_r, jj_r) \tag{9.13}$$

Explanation:

1. Equations/inequalities 9.7 - 9.10 are constraints on the indices.
2. 9.11 is the condition for true dependence.
3. 9.12, 9.13 represent the condition for accessing same memory location.

If f, g, h, k are affine functions of loop variables, then dependence testing can be formulated as an **Integer Linear Program** (ILP).

9.9.2 ILP Formulation

```
for(ii = 1; ii <10; ii++)
{
    a[2 * ii] = ... a[2 * ii + 1] ...    //S1
}
```

Question: Is there a flow dependence between iterations of S1?
Solution: Equations for flow dependence for the loop are:

$$0 \le ii_r < ii_w < 10 \tag{9.14}$$

$$2 * ii_w = 2 * ii_r + 1 \tag{9.15}$$

which can be written as:

$$0 \le ii_w \tag{9.16}$$

$$ii_w \le ii_r - 1 \tag{9.17}$$

$$ii_r \le 9 \tag{9.18}$$

$$2 * ii_w \le 2 * ii_r + 1 \tag{9.19}$$

$$2 * ii_r + 1 \le 2 * ii_w \tag{9.20}$$

note: dependency exists if the system of equations has a solution.
equations 9.16 to 9.20 can be written as:

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

The system is not satisfiable, so flow-dependence does not exist.

Question: Is there an anti-dependence between different iterations of S1?
Solution: Dependence equations for the loop are:

$$0 \leq ii_w < ii_r < 10 \tag{9.21}$$

$$2 * ii_w = 2 * ii_r + 1 \tag{9.22}$$

which can be written as:

$$0 \leq ii_r \tag{9.23}$$

$$ii_r \leq ii_w - 1 \tag{9.24}$$

$$ii_w \leq 9 \tag{9.25}$$

$$2 * ii_w \leq 2 * ii_r + 1 \tag{9.26}$$

$$2 * ii_r + 1 \leq 2 * ii_w \tag{9.27}$$

or,

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

The system is not satisfiable, so anti-dependence does not exist.

References

[1] Rupesh Sir's slides