Fundamental Algorithms for System Modeling, Analysis, and Optimization

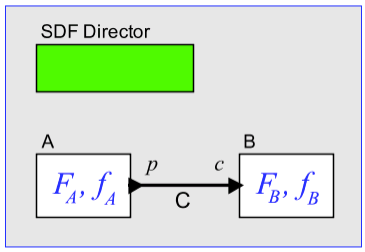
Edward A. Lee, Jaijeet Roychowdhury, Sanjit A. Seshia
UC Berkeley
EECS 144/244
Fall 2010

Lecture M: Scheduling

Start with Dataflow Models

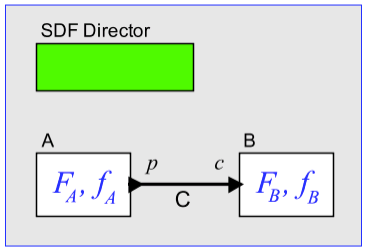
- Computation graphs [Karp & Miller - 1966]
- Process networks [Kahn - 1974]
- Static dataflow [Dennis - 1974]
- Dynamic dataflow [Arvind, 1981]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Synchronous languages [Lustre, Signal, 1980's]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- Heterochronous dataflow [Girault, Lee, & Lee, 1997]
- Parameterized dataflow [Bhattacharya and Bhattacharyya 2001]
- Scenarios [Geilen, 2010]
- …

Synchronous Dataflow (SDF)



If the number of tokens consumed and produced by the firing of an actor is constant, then static analysis can tell us whether we can schedule the firings to get a useful execution, and if so, then a finite representation of a schedule for such an execution can be created.

Balance Equations



- Let q_A, q_B be the number of firings of actors A and B.
- Let p_C, c_C be the number of tokens produced and consumed on a connection C.
- Then the system is *in balance* if for all connections C

$$q_A p_C = q_B c_C$$

where A produces tokens on C and B consumes them.

Relating to Infinite Firings

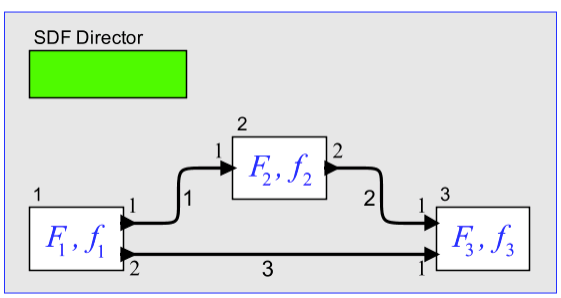
Of course, if $q_A = q_B = \infty$, then the balance equations are trivially satisfied.

By keeping a system in balance as an infinite execution proceeds, we can keep the buffers bounded.

Whether we can have a bounded infinite execution turns out to be decidable for SDF models.

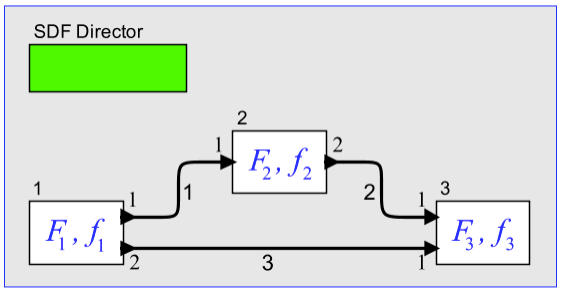
Example

Consider this example, where actors and arcs are numbered:



The balance equations imply that actor 3 must fire twice as often as the other two actors.

Compactly Representing the Balance Equations



production/consumption matrix

Connector 1

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix}$$

Actor 1

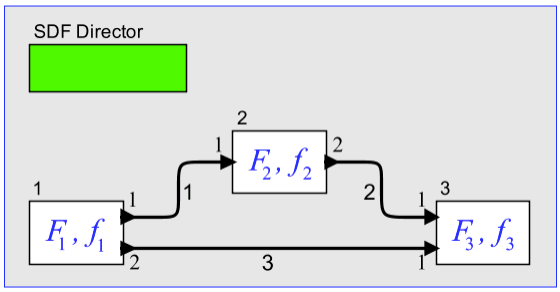
$$q = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

firing vector

balance equations

$$\Gamma q = \vec{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Example

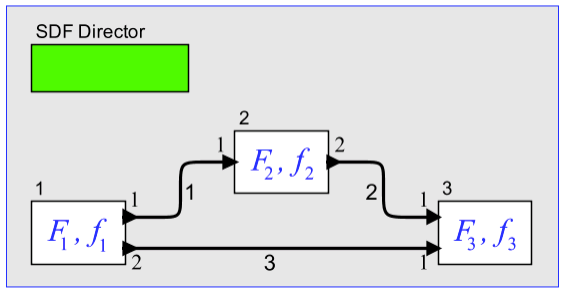


A solution to balance equations:

$$q = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \qquad \Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \qquad \Gamma q = \vec{0}$$

This tells us that actor 3 must fire twice as often as actors 1 and 2.

Example

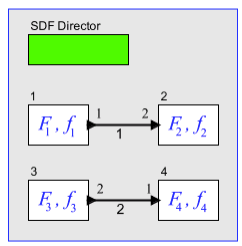


But there are many solutions to the balance equations:

$$q = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \quad q = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad q = \begin{bmatrix} 2 \\ 2 \\ 4 \end{bmatrix} \quad q = \begin{bmatrix} -1 \\ -1 \\ -2 \end{bmatrix} \quad q = \begin{bmatrix} \pi \\ \pi \\ 2\pi \end{bmatrix} \qquad \Gamma q = \vec{0}$$

We will see that for “well-behaved” models, there is a unique least positive solution.

Disconnected Models



For a disconnected model with two connected components, solutions to the balance equations have the form:

Solutions are linear combinations of the solutions for each connected component:

$$\Gamma = \begin{bmatrix} 1 & -2 & 0 & 0 \\ 0 & 0 & 2 & -1 \end{bmatrix} \qquad q = \begin{bmatrix} 2n \\ n \\ m \\ 2m \end{bmatrix} = n \begin{bmatrix} 2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + m \begin{bmatrix} 0 \\ 0 \\ 1 \\ 2 \end{bmatrix}$$

$$\Gamma q = \vec{0}$$

Disconnected Models are Just Separate Connected Models

Define a *connected model* to be one where there is a path from any actor to any other actor, and where every connection along the path has production and consumption numbers greater than zero.

It is sufficient to consider only connected models, since disconnected models are disjoint unions of connected models. A schedule for a disconnected model is an arbitrary interleaving of schedules for the connected components.

Least Positive Solution to the Balance Equations

Note that if p_C, c_C , the number of tokens produced and consumed on a connection C, are non-negative integers, then the balance equation,

$$q_A p_C = q_B c_C$$

implies:

- q_A is rational if an only if q_B is rational.
- q_A is positive if an only if q_B is positive.

Consequence: Within any connected component, if there is any solution to the balance equations, then there is a unique least positive solution.

Rank of a Matrix

The rank of a matrix Γ is the number of linearly independent rows or columns. The equation

$$\Gamma q = \vec{0}$$

is forming a linear combination of the columns of G. Such a linear combination can only yield the zero vector if the columns are linearly dependent (this is what is means to be linearly dependent).

If Γ has a rows and b columns, the rank cannot exceed $\min(a, b)$. If the columns or rows of Γ are re-ordered, the resulting matrix has the same rank as Γ.

Rank of the Production/Consumption Matrix

Let a be the number of actors in a connected graph. Then the *rank* of the production/consumption matrix Γ must be a or $a - 1$.

Γ has a columns and at least $a - 1$ rows. If it has only $a - 1$ columns, then it cannot have rank a.

If the model is a *spanning tree* (meaning that there are barely enough connections to make it connected) then Γ has a rows and $a - 1$ columns. Its rank is $a - 1$. (Prove by induction).

Consistent Models

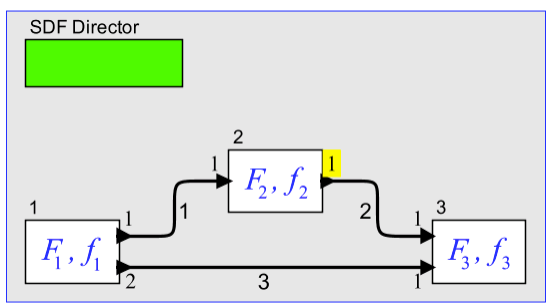
Let a be the number of actors in a connected model. The model is *consistent* if Γ has rank a - 1.

If the rank is a, then the balance equations have only a trivial solution (zero firings).

When Γ has rank a - 1, then the balance equations always have a non-trivial solution.

Example of an Inconsistent Model: No Non-Trivial Solution to the Balance Equations

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix}$$



This production/consumption matrix has rank 3, so there are no nontrivial solutions to the balance equations.

Solving the Balance Equations: Brute Force

Brute force:

- Construct the Γ matrix (order ea) for a actors, e edges.
- Use standard algorithms solving systems of equations (Gaussian elimination, LU factorization, or sparse matrix techniques).

However, we can use an understanding of the problem structure to get a quite efficient algorithm.

Solving the Balance Equations: Fractional Iteration Vector

Use the graph representation:

- Define a fractional data type as `int[2]`.
- Pick any actor A and set $q_A = \{1, 1\}$ representing $1/1$.
- Solve $q_A p_C = q_B c_C$ for connected actor B getting

$$q_B = q_A p_C / c_C$$

- If any path to an actor B yields a different answer, then declare the graph inconsistent.
- Continue until all actors have an iteration fraction and all edges have been traversed.
- Complexity: order e where e is the number of edges.

Solving the Balance Equations: Finding the Least Integer Solution

- Find the least common multiple of the denominators.
- Multiply each fraction by this LCM (order a).

Finding the LCM:

$$\mathrm{LCM}(u,v) = (uv) / \mathrm{GCD}\ (u, v)$$

Moreover:

$$\begin{aligned}\mathrm{LCM}(u,v,w) &= \mathrm{LCM}(u, \mathrm{LCM}(v,w)) \\ &= \mathrm{LCM}(\mathrm{LCM}(u,v), w) \\ &= \mathrm{LCM}(v, \mathrm{LCM}(u,w))\end{aligned}$$

Euclid's Algorithm for the Greatest Common Divisor (GCD)

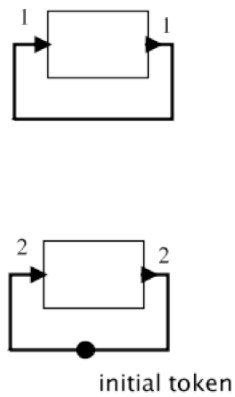
```
public static int gcd (int u, int v) {
    int t;
    // Make both numbers nonnegative.
    if (u < 0) u = -u;
    if (v < 0) v = -v;
    while (u > 0) {
        // Make sure u >= v.
        if (u < v) {
            t = u;
            u = v;
            v = t;
        } else {
            // subtract n*v from u.
            u = u % v;
        }
    }
    return v;
}
```

Euclid's algorithm is based on the principle that the greatest common divisor of two numbers does not change if the smaller number is subtracted from the larger number.

The earliest surviving description of the Euclidean algorithm is in Euclid's Elements (c. 300 BC), making it one of the oldest numerical algorithms still in common use.

Consistency is Necessary but not Sufficient for Existence of a PASS

Deadlock is a possibility:



Symbolic Execution: Check for Deadlock & Build a Schedule

Consider a model with 3 actors. Let the *schedule* be a sequence $v : N_0 \rightarrow B^3$ where $B = \{0, 1\}$ is the binary set. That is,

$$v(n) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

to indicate firing of actor 1, 2, or 3.

Symbolic Execution to find a Periodic Sequential Schedule

Assume there are m connections and let $b : N_0 \rightarrow N^m$ indicate the buffer sizes prior to the each firing. That is, $b(0)$ gives the initial number of tokens in each buffer, $b(1)$ gives the number after the first firing, etc. Then

$$b(n+1) = b(n) + \Gamma v(n)$$

A *periodic admissible sequential schedule* (PASS) of length K is a sequence

$$v(0) \ldots v(K-1)$$

such that $b(n) \geq \vec{0}$ for each $n \in \{0, \ldots K-1\}$, and

$$b(K) = b(0) + \Gamma\left[v(0) + \ldots + v(K-1)\right] = b(0)$$

Periodic Sequential Schedule

Let $q = v(0) + \ldots + v(K-1)$
and note that we require that $\Gamma q = \vec{0}$.

A PASS will bring the model back to its initial state, and hence it can be repeated indefinitely with bounded memory.

A necessary condition for the existence of such a schedule is that the balance equations have a non-zero solution. Hence, a PASS can only exist for a consistent model.

SDF Theorem 1

We have proved:

For a connected SDF model with a actors, a *necessary* condition for the existence of a PASS is that the model be consistent.

SDF Theorem 2

We have also proved:

For a consistent connected SDF model with production/consumption matrix Γ, we can find an integer vector q where every element is greater than zero such that

$$\Gamma q = \vec{0}$$

Furthermore, there is a unique least such vector q.

SDF Sequential Scheduling Algorithms

Given a consistent connected SDF model with production/consumption matrix Γ, find the least positive integer vector q such that $\Gamma q = \vec{0}$.

Let $K = \mathbf{1}^{\mathrm{T}} q$, where $\mathbf{1}^{\mathrm{T}}$ is a row vector filled with ones. Then for each of $n \in \{0, \ldots K-1\}$, choose a firing vector

$$v(n) \in \left\{ \begin{bmatrix} 1 \\ 0 \\ \ldots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \ldots \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ \ldots \\ 1 \end{bmatrix} \right\}$$

The number of rows in $v(n)$ is a.

SDF Sequential Scheduling Algorithms (Continued)

.. such that $b(n+1) = b(n) + \Gamma v(n) \geq \vec{0}$ (each element is non-negative), where $b(0)$ is the initial state of the buffers, and

$$\sum_{n=0}^{K-1} v(n) = q$$

The resulting *schedule* ($v(0)$, $v(1)$, …, $v(K-1)$) forms one cycle of an infinite periodic schedule.

Such an algorithm is called an *SDF Sequential Scheduling Algorithm (SSSA).*

SDF Theorem 3

If an SDF model has a correct infinite sequential execution that executes in bounded memory, then any SSSA will find a schedule that provides such an execution.

Proof outline: Must show that if an SDF has a correct, infinite, bounded execution, then it has a PASS of length K. See Lee & Messerschmit [1987]. Then must show that the schedule yielded by an SSSA is correct, infinite, and bounded (trivial).

Note that every SSSA terminates.

Creating a Schedule

Given a connected SDF model with actors $A_1, \ldots, A_a$:

Step 1: Solve for a rational q.

Recap: To do this, first let $q_1 = 1$. Then for each actor A_i connected to A_1, let $q_i = q_1 m/n$, where m is the number of tokens A_1 produces or consumes on the connection to A_i, and n is the number of tokens A_i produces or consumes on the connection to A_1. Repeat this for each actor A_j connected to A_i for which we have not already assigned a value to q_j. When all actors have been assigned a value q_j, then we have a found a rational vector q such that

$$\Gamma q = \vec{0}$$

Creating a Schedule (continued)

Step 2: Solve for the least integer q.

Recap: Use Euclid's algorithm to find the least common multiple of the denominators for the elements of the rational vector q. Then multiply through by that least common multiple to obtain the least positive integer vector q such that

$$\Gamma q = \vec{0}$$

Let $K = \mathbf{1}^{\mathrm{T}} q$.

Creating a Schedule (continued)

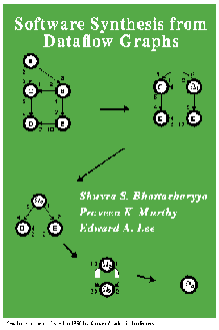
Step 3: Symbolic Execution.

For each $n \in \{0, \ldots, K-1\}$:

1. Given buffer sizes $b(n)$, determine which actors have firing rules that are satisfied (every source actor will have such a firing rule).
2. Select one of these actors that has not already been fired the number of times given by q. Let $v(n)$ be a vector with all zeros except in the position of the chosen actor, where its value is 1.
3. Update the buffer sizes:

$$b(n+1) = b(n) + \Gamma v(n)$$

A Key Question: If More Than One Actor is Fireable in Step 2, How do I Select One?

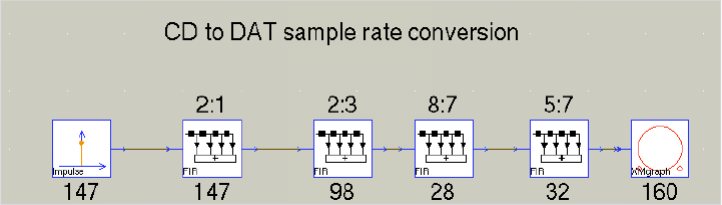


Optimization criteria that might be applied:

- Minimize buffer sizes.
- Minimize the number of actor activations.
- Minimize the size of the representation of the schedule (code size).

See S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs,* Kluwer Academic Press, 1996.

Minimum Buffer Schedule



ABABCABCABABCABCDEAFFFFFBABCABCABABCDE
AFFFFFBCABABCABCABABCDEAFFFFFBCABABCABC
DEAFFFFFBABCABCABABCABCDEAFFFFFBABCABCA
BABCDEAFFFFFBCABABCABCABABCDEAFFFFFEBCA
FFFFFBABCABCDEAFFFFFBABCABCABABCABCDEAF
FFFFBABCABCABABCDEAFFFFFBCABABCABCABABC
DEAFFFFFBCABABCABCDEAFFFFFBABCABCABABCA
BCDEAFFFFFBABCABCABABCDEAFFFFFEBCAFFFFFB
ABCABCABABCDEAFFFFFBCABABCABCDEAFFFFFBA
BCABCABABCABCDEAFFFFFBABCABCABABCDEAFFF
FFBCABABCABCABABCDEAFFFFFBCABABCABCDEAF
FFFFBABCABCABABCABCDEAFFFFFEBAFFFFFBCABC
ABABCDEAFFFFFBCABABCABCABABCDEAFFFFFBCA
BABCABCDEAFFFFFBABCABCABABCABCDEAFFFFFB
ABCABCABABCDEAFFFFFBCABABCABCABABCDEAF
FFFFBCABABCABCDEFFFFFFEFFFFF

Source: Shuvra Bhattacharyya

Code Generation (Circa 1992)

Block specification for DSP code generation in Ptolemy Classic:

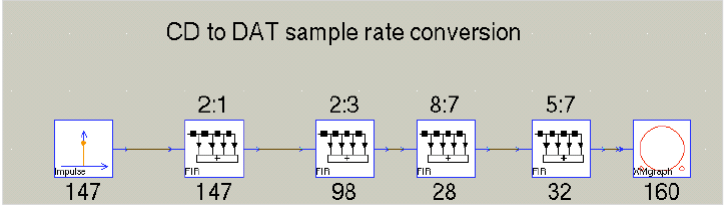
```
        codeblock(std) {
        ; initialize address registers for coef and
 delayLineove     #$addr(coef)+$val(coefLen)-1,r3
; insert here
        move     $ref(delayLineStart),r5
; delayLine
        move     #$val(stepSize),x1
        move     $ref(error),x0
        mpyr     x0,x1,a
        move     a,x0
        move     x:(r3),b          y:(r5)+,y0
        }

        codeblock(loop) {
        do       #$val(loopVal),$label(endloop)
        macr     x0,y0,b
        move     b,x:(r3)-
        move     x:(r3),b          y:(r5)+,y0
$label(endloop)
        }

        codeblock(noloop) {
        macr     x0,y0,b
        move     b,x:(r3)-
        move     x:(r3),b          y:(r5)+,y0
        }
```

macros defined by the code generator

alternative code blocks chosen based on parameter values

Scheduling Tradeoffs

(Bhattacharyya, Parks, Pino)



Scheduling strategy	Code	Data
Minimum buffer schedule, no looping	13735	32
Minimum buffer schedule, with looping	9400	32
Worst minimum code size schedule	170	1021
Best minimum code size schedule	170	264

Source: Shuvra Bhattacharyya

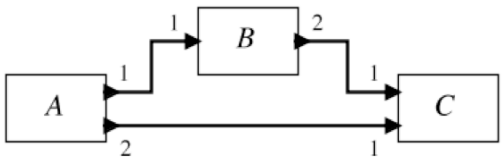
Parallel Scheduling

Consider three scenarios:

- Single CPU to execute dataflow graph (use PASS)
- Fixed number of CPUs to execute dataflow graph
- Separate hardware for every actor (circuit)
- Unbounded compute resources

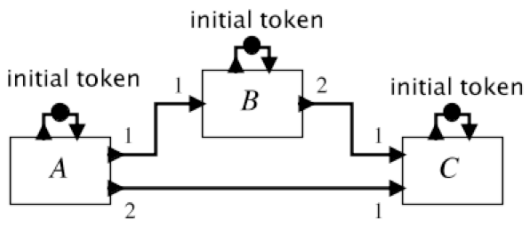
Unbounded Compute Resources

The following dataflow model can execute infinitely fast, in principle, if resources are unbounded:



Modeling Separate Hardware for Each Actor

The self loops below constrain the execution of each actor to be sequential:



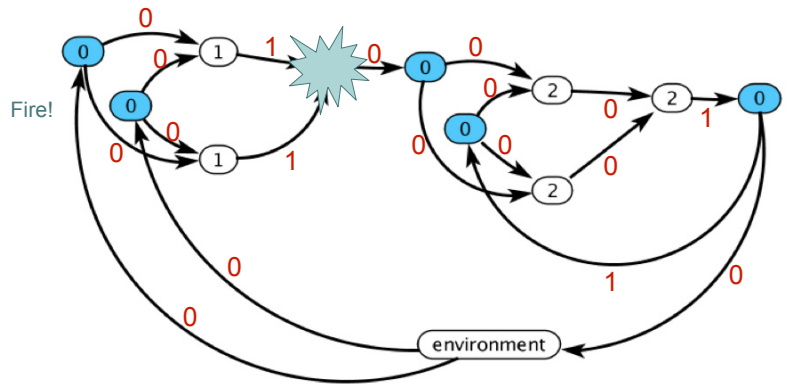
Modeling Synchronous Circuits as Dataflow

- Constrain every actor to consume and produce one token on every port (*homogeneous SDF*).
- Put a self loop on every actor (gate)
- Put an initial token wherever there is a register.

Recall that *retiming* can now be modeled as a finite set of *initial firings* that move around the initial tokens.

Retiming as Dataflow Firings

In a *dataflow* graph, nodes represent *actors*, which *fire* when input *tokens* are available. Firing performs a computation that takes time. Weights represent *initial* tokens. Retiming can be interpreted as a *preamble* to a periodic schedule, and may have the goal of maximizing parallelism so that the dataflow graph executes fast on a multicore machine.



Parallel Scheduling with Bounded Resources

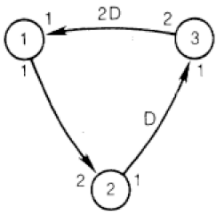
It is easy to create an algorithm that as it produces a PASS, it constructs an *acyclic precedence graph* (APG) that represents the dependencies that an actor firing has on prior actor firings for one or more iterations of a PASS.

Given such an APG, the parallel scheduling problem is a standard one where there are many variants of the optimization criteria and scheduling heuristics.

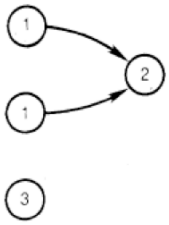
E.g.: Minimize the *makespan* (completion time of the whole schedule). This problem in NP complete.

Acyclic Precedence Graph

Dataflow graph:



APG for one cycle of a PASS:



[Lee & Messerschmitt, 1987]

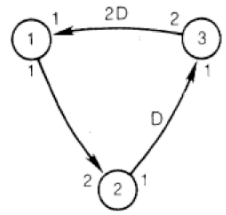
List Scheduling

List scheduling assigns to each element of an APG a *level* or *priority* and then greedily fires actors

Widely used list scheduling techniques are known as HLFET (Highest Levels First with Estimated Times), the earliest (and simplest) of which is the Hu Level Scheduling technique.

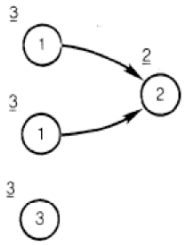
Acyclic Precedence Graph

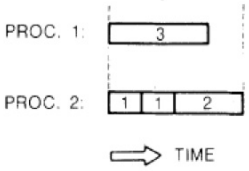
Dataflow graph:



Two processor schedule, showing a makespan of 4:

APG annotated with Hu Levels, assuming execution times are 1, 2, and 3 for actors 1, 2, and 3.

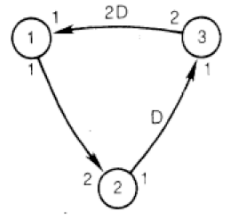




[Lee & Messerschmitt, 1987]

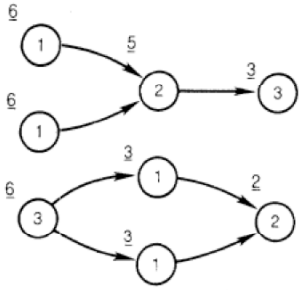
Acyclic Precedence Graph with a Larger *Vectorization Factor*

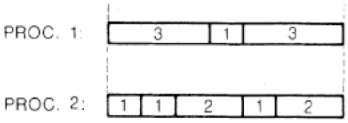
Dataflow graph:



Two processor schedule, showing a makespan of 7, yielding higher throughput:

APG w/ levels:





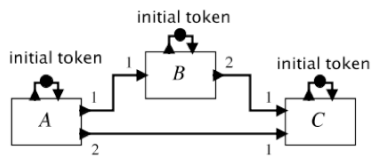
[Lee & Messerschmitt, 1987]

Parallel Scheduling with Unbounded Resources

Self-timed execution: Every actor fires as soon as it has sufficient inputs (even if the previous firing has not finished).

This puts a bound on *throughput*, which we define to be the number of firings of a chosen actor per unit time.

Resource constraints (limited number of processors) can be modeled as extra cycles, as we did here.



Max-Plus Algebra

Operators:

$$a \oplus b = \max(a,b)$$

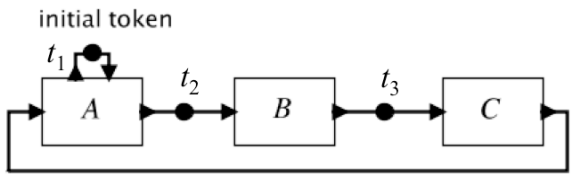
$$a \otimes b = a + B - side$$

Algebra properties:

- associativity:
  $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
  $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
- commutativity:
  $a \oplus b = b \oplus a$
  (note: $\otimes$ is not commutative for matrix ops)
- distributivity:
  $(a \oplus b) \otimes b = (a \otimes c) \oplus (b \otimes c)$

Max-Plus Dynamics

Consider a simple homogeneous-SDF graph:

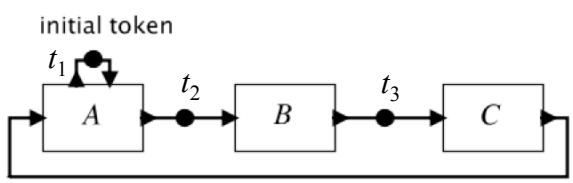


Number each initial token. A complete iteration must regenerate each such token. The earliest time at which it can be regenerated is given by a formula of the form:

$$t_i' = \max(d_{i,1}+t_1, d_{i,2}+t_2, d_{i,3}+t_3)$$

[Geilen and Stuijk, 2010]

Max-Plus Dynamics

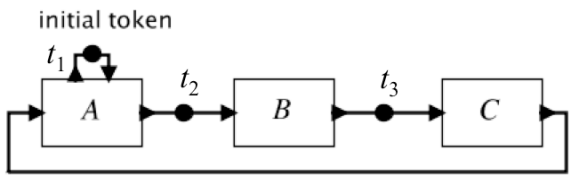


For this example:

$$\begin{aligned} t_1' &= \max(d_A+t_1, -\infty+t_2, (d_C+d_A)+t_3) \\ t_2' &= \max(d_A+t_1, -\infty+t_2, (d_C+d_A)+t_3) \\ t_3' &= \max(-\infty+t_1, d_B+t_2, \infty+t_3) \end{aligned}$$

[Geilen and Stuijk, 2010]

Max-Plus Dynamics

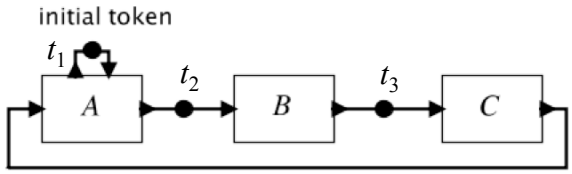


In max-plus algebra:

$$\begin{aligned} t_i' &= \max(d_{i,1}+t_1, d_{i,2}+t_2, d_{i,3}+t_3) \\ &= (d_{i,1} \otimes t_1) \oplus (d_{i,2} \otimes t_2) \oplus (d_{i,3} \otimes t_3) \end{aligned}$$

[Geilen and Stuijk, 2010]

Max-Plus Dynamics



In max-plus algebra:

$$\begin{aligned} \mathbf{t}' &= G\mathbf{t} \\ \begin{bmatrix} t_1' \\ t_2' \\ t_3' \end{bmatrix} &= \begin{bmatrix} d_A & -\infty & (d_C+d_A) \\ d_A & -\infty & (d_C+d_A) \\ -\infty & d_b & -\infty \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \end{aligned}$$

[Geilen and Stuijk, 2010]

References

1. Edward A. Lee and David G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, Vol. C-36, No. 1, pp. 24-35, January, 1987.

2. Baccelli, F., G. Cohen, G. J. Olster and J. P. Quadrat (1992). *Synchronization and Linearity*, An Algebra for Discrete Event Systems. New York, Wiley.

3. Gilbert C. Sih and Edward A. Lee, "Declustering: A New Multiprocessor Scheduling Technique," *IEEE Trans. on Parallel and Distributed Systems,* vol. 4, no. 6, pp. 625-637, June 1993.

4. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs* Kluwer Academic Press, 1996.

5. Geilen, M. and S. Stuijk (2010). Worst-case Performance Analysis of Synchronous Dataflow Scenarios. CODES+ISSS, Scottsdale, Arizona, USA, October 24–29,