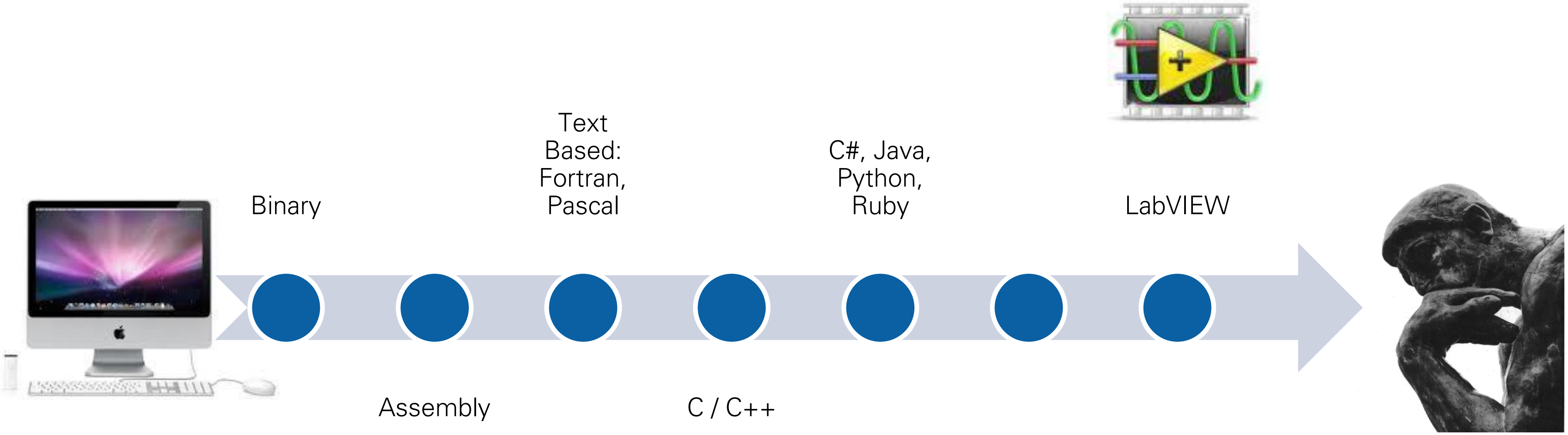
A Graphical Dataflow Programming Approach To High Performance Computing

Somashekaracharya G. Bhaskaracharya
National Instruments
Bangalore

Outline

- Graphical Dataflow Programming
- LabVIEW – Introduction and Demo
- LabVIEW Compiler (under the hood)
- Multicore Programming in LabVIEW
- Polyhedral Compilation of Graphical Dataflow Programs

Evolution of Programming Languages



Binary

Assembly

Text Based: Fortran, Pascal

C / C++

C#, Java, Python, Ruby

LabVIEW

Graphical Dataflow v/s Imperative Programs

Imperative Programming

- Computation specified as sequence of statements
- Each statement changes the program state

```
// s = ut + 0.5a*t*t
double displacement_in_time_t(double time,
                              double initial_velocity,
                              double acceleration) {
   double displacement = initial_velocity * time;
   displacement += 0.5 * acceleration * time * time;
   return displacement;
}
```

Graphical Dataflow v/s Imperative Programs

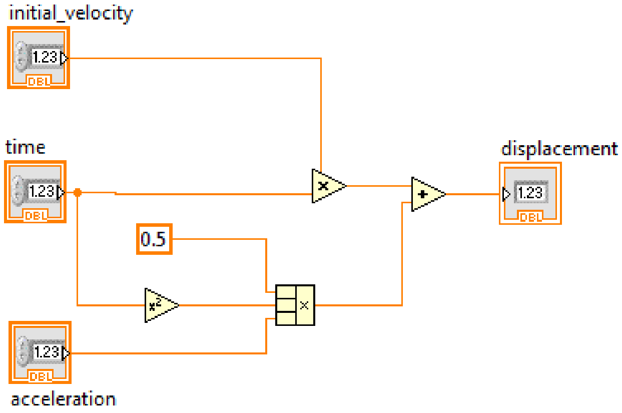
Imperative Programming

- Computation specified as sequence of statements
- Each statement changes the program state

```
// s = ut + 0.5a*t*t
double displacement_in_time_t(double time,
                              double initial_velocity,
                              double acceleration) {
   double displacement = initial_velocity * time;
   displacement += 0.5 * acceleration * time * time;
   return displacement;
}
```

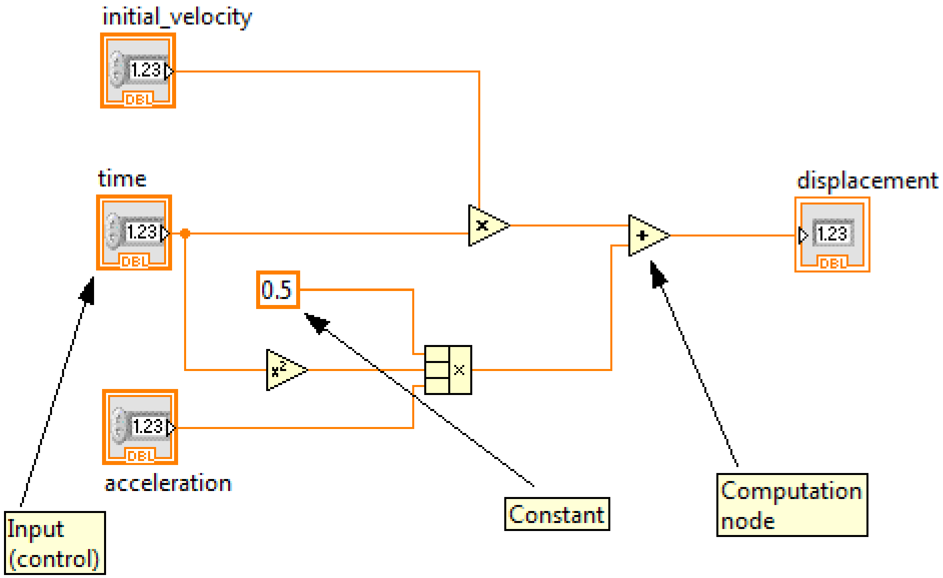
Graphical dataflow programming

- No notion of statements
- No fixed relative execution order
- Referential transparency



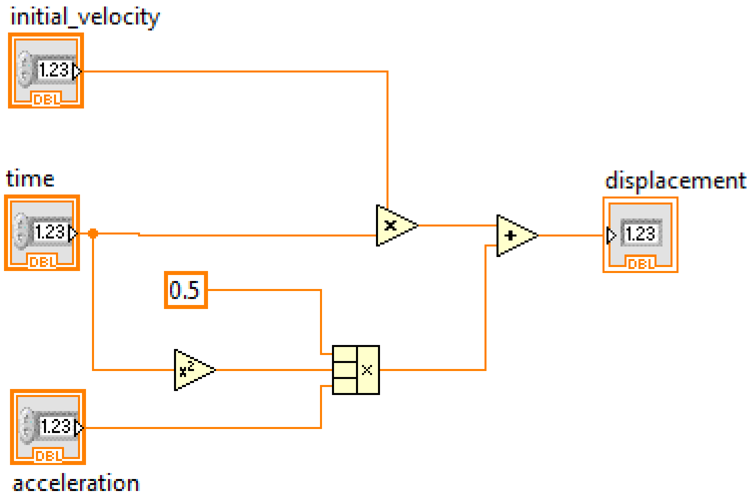
Dataflow Execution Semantics

- Interconnected set of nodes that represent specific computations
- Nodes consume input data to produce output data
- Nodes ready to ***fired*** as soon as data is available on all inputs



Inherent Parallelism Of Dataflow Programs

Partially ordered program specification



Possible orderings of node execution:

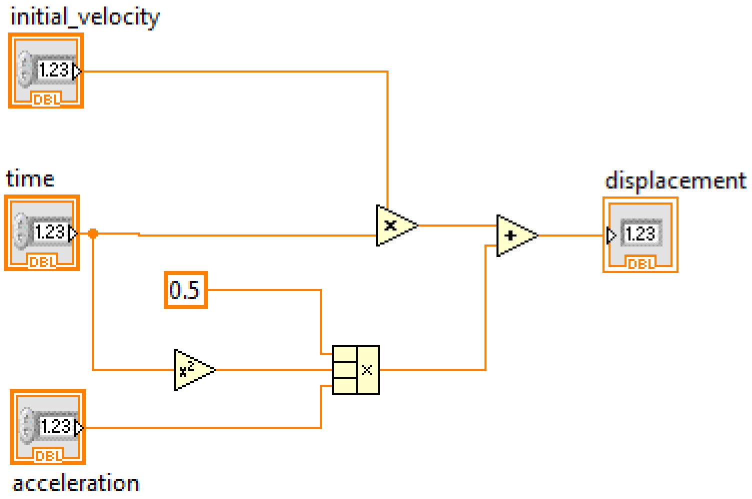
Strictly Sequential

- Multiply < Square < TernaryMultiply < Add
- Square < TernaryMultiply < Multiply < Add
- Square < Multiply < TernaryMultiply < Add

- Sequentiality enforced through data dependences

Inherent Parallelism Of Dataflow Programs

Partially ordered program specification



Possible orderings of node execution:

Strictly Sequential

- Multiply < Square < TernaryMultiply < Add
- Square < TernaryMultiply < Multiply < Add
- Square < Multiply < TernaryMultiply < Add

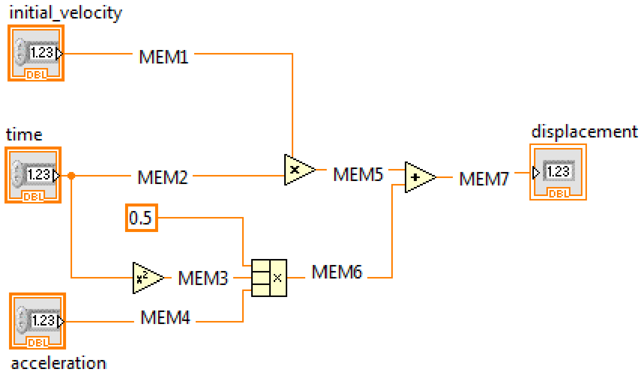
Exploiting inherent parallelism

- (Multiply | | Square) < TernaryMultiply < Add
- (Multiply | | (Square < TernaryMultiply)) < Add
- Square < (Multiply | | TernaryMultiply) < Add

- Sequentiality enforced through data dependences
- Compiler determines the granularity of parallelism

Memory Allocation in Graphical Dataflow

- Valid to substitute expression with its value
  - at any point in program execution

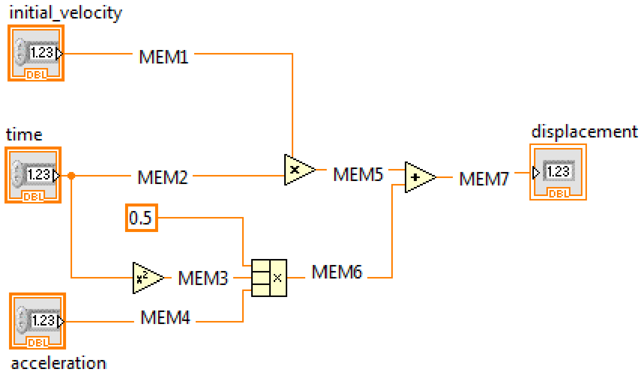


Programmer's perspective of memory allocation

Each new output value in a new memory location

Memory Allocation in Graphical Dataflow

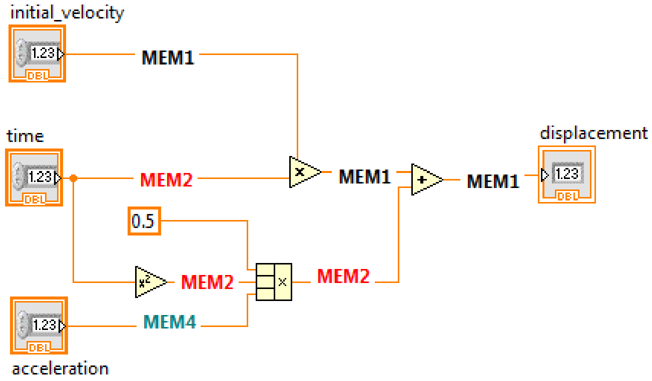
- Valid to substitute expression with its value
  - at any point in program execution



Programmer's perspective of memory allocation

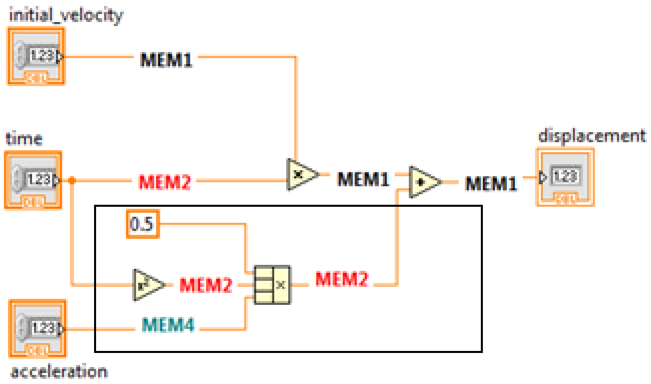
Each new output value in a new memory location

- Copy avoidance strategies to reduce memory overhead
  - Output data is ***inplace*** to input data wherever possible



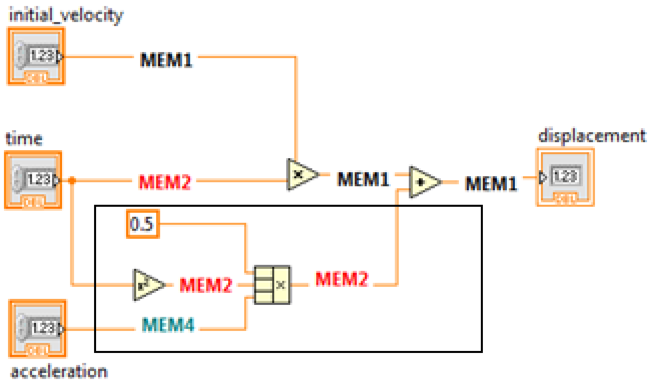
After copy-avoidance, only 3 memory allocations are needed

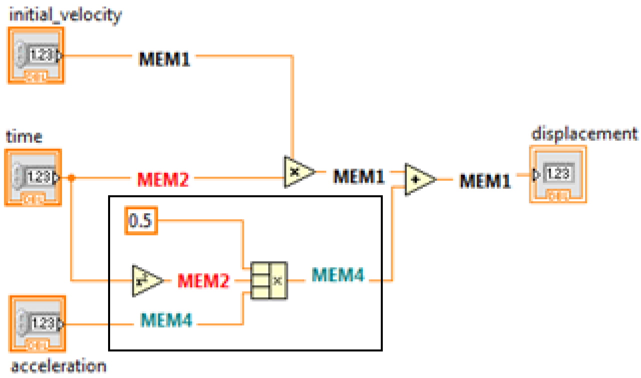
Copy-avoidance and Execution Schedule



- ~~TernaryMultiply < Multiply~~
  - Destructive update of MEM2
  - Pending read of MEM2
- Cannot exploit parallelism

Copy-avoidance and Execution Schedule





- ~~TernaryMultiply < Multiply~~
  - Destructive update of MEM2
  - Pending read of MEM2
- Cannot exploit parallelism

- No destructive update of MEM2
- TernaryMultiply < Multiply
- TernaryMultiply | | Multiply
- TernaryMultiply > Multiply

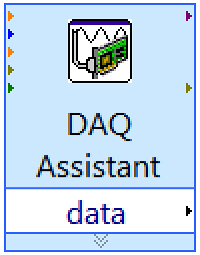
Strong interplay between **copy-avoidance**, **clumping** and **scheduling**

Outline

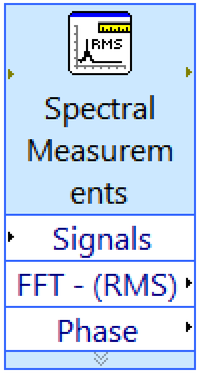
- Graphical Dataflow Programming
- LabVIEW – Introduction and Demo
- LabVIEW Compiler (under the hood)
- Multicore Programming in LabVIEW
- Polyhedral Compilation of Graphical Dataflow Programs

LabVIEW

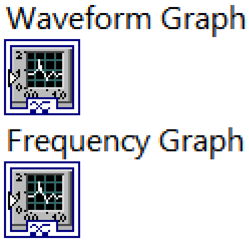
- Platform for graphical dataflow programming
  - Owned by National Instruments
  - G dataflow programming language
  - Editor, compiler, runtime and debugger
  - Supported on Windows, Linux, Mac
  - Power PC, Intel architectures, FPGA



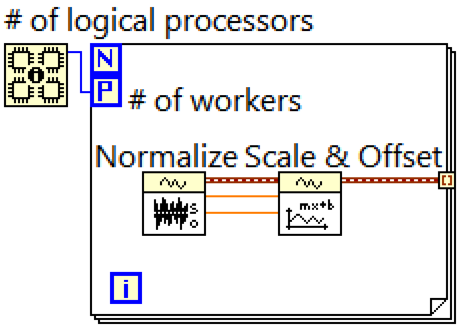
Measurement Control I/O



Deployable Math and Analysis



User Interface

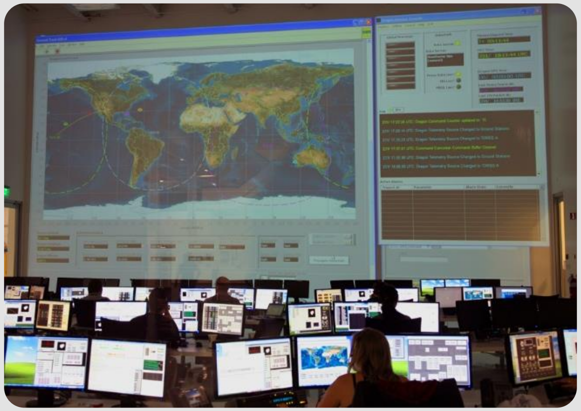


Technology Integration

Scalable: From Kindergarten to Rocket Science

LEGO mindstorms education

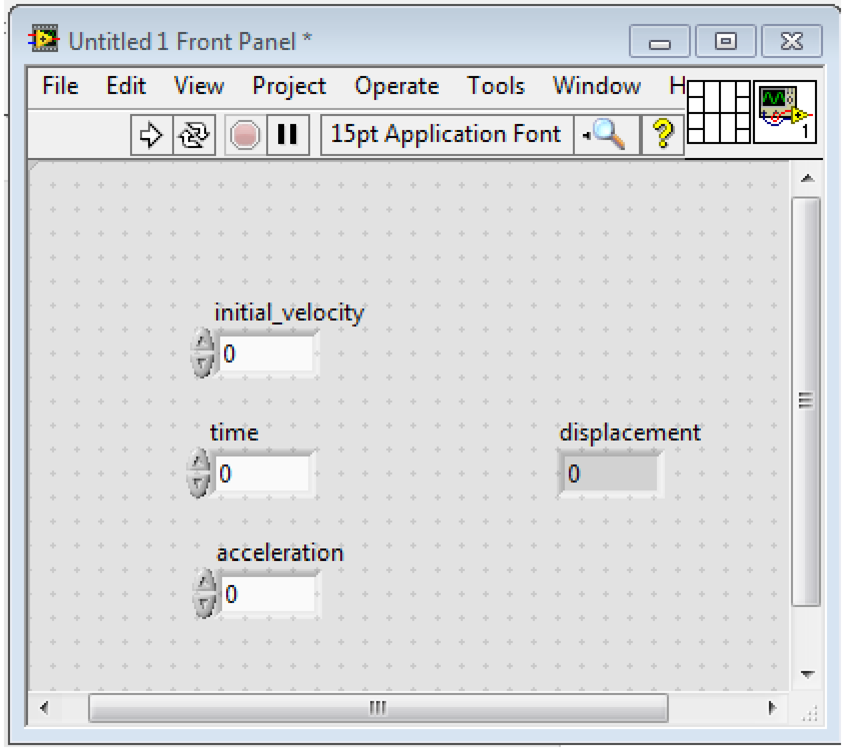
NATIONAL INSTRUMENTS LabVIEW

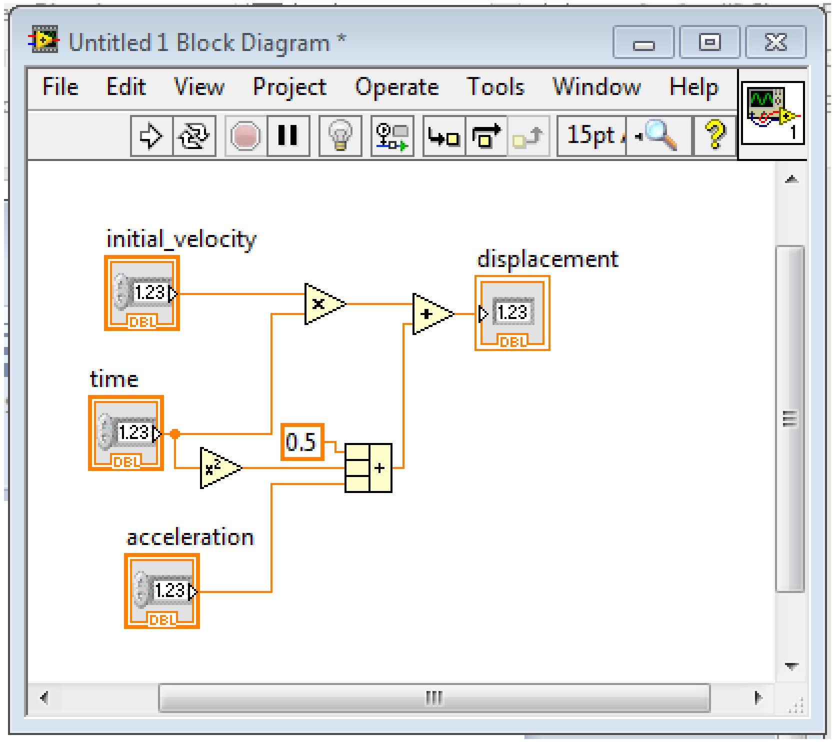


SPACEX

LabVIEW Program

- LabVIEW program
  - Front Panel + Block Diagram

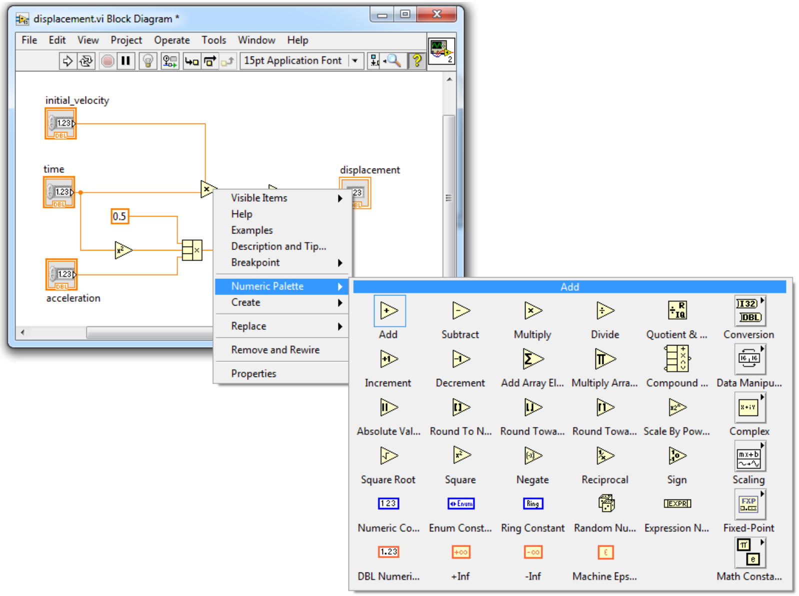


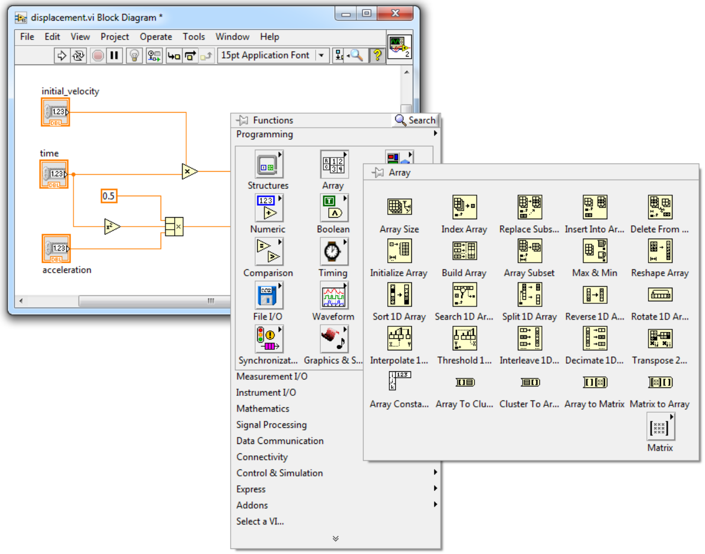


G Programming Language

- Data types
  - Built-in types: integer and floating point types, Boolean, string etc
  - Aggregate types: arrays, clusters, classes

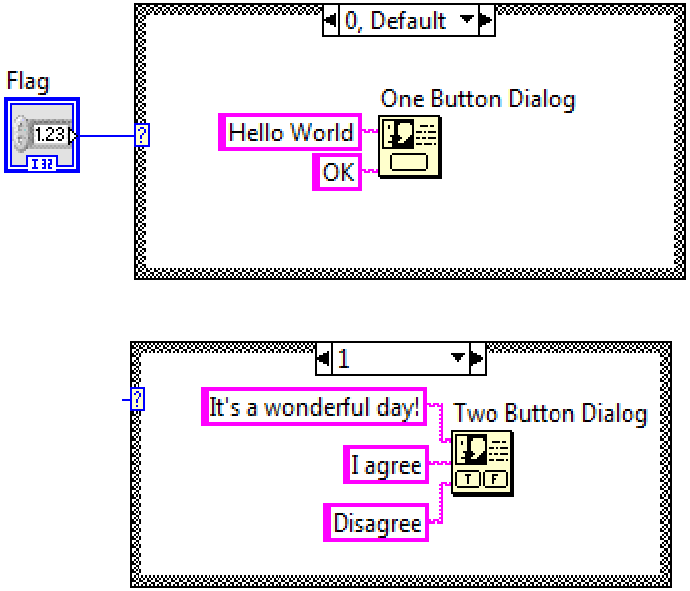
- Data manipulation through built-in collection of primitives
  - Numeric palette (add, multiply, divide, subtract etc)
  - Array palette (Build array, Index array, concatenate array, decimate array etc)





G Programming Language – Control Constructs

- Case Structure

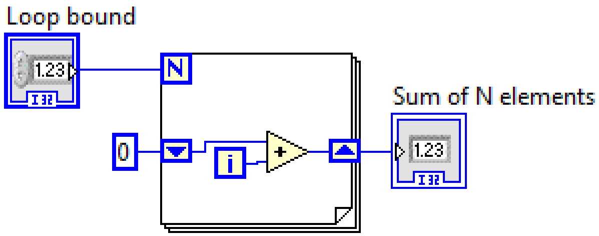


- One or more diagrams (cases)
- Value wired to selector terminal for switching
  - Boolean, string, integer, enumerated type

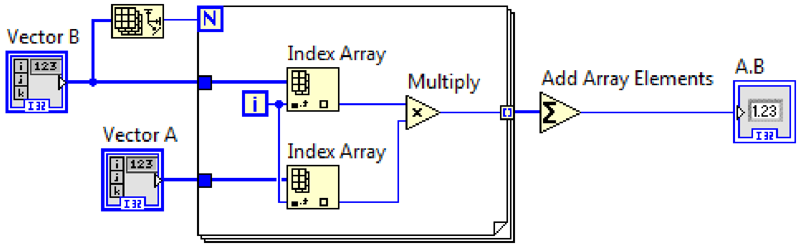
G Programming Language – Control Constructs

Loop structures

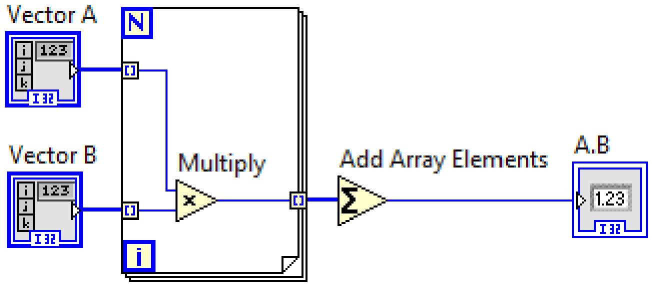
- While loop
- Timed loop
- **For loop**
  - LoopMax and LoopIndex boundary nodes
  - Loop carried data through shift registers
  - Tunnels (with optional indexing)



Shift registers to propagate data across iterations



Unindexed tunnels propagate same data every iteration



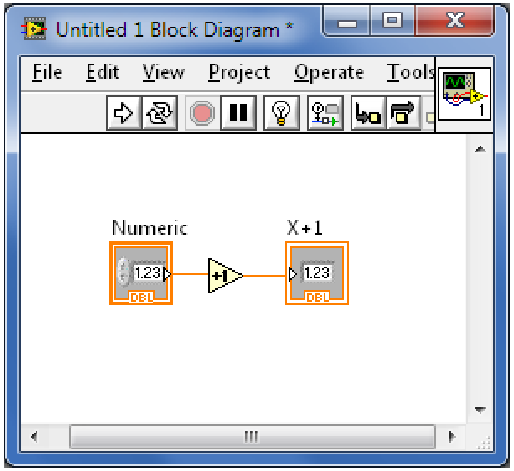
Indexed tunnels

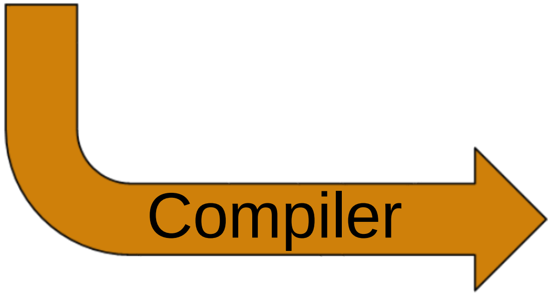
- Array auto-indexing
- Auto- accumulate iteration outputs

Outline

- Graphical Dataflow Programming
- LabVIEW – Introduction and Demo
- LabVIEW Compiler (under the hood)
- Multicore Programming in LabVIEW
- Polyhedral Compilation of Graphical Dataflow Programs

LabVIEW Compiler

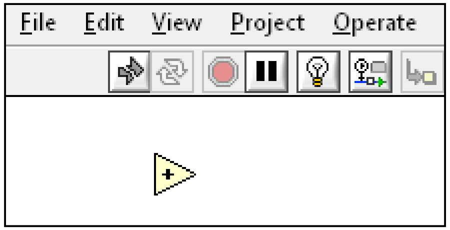


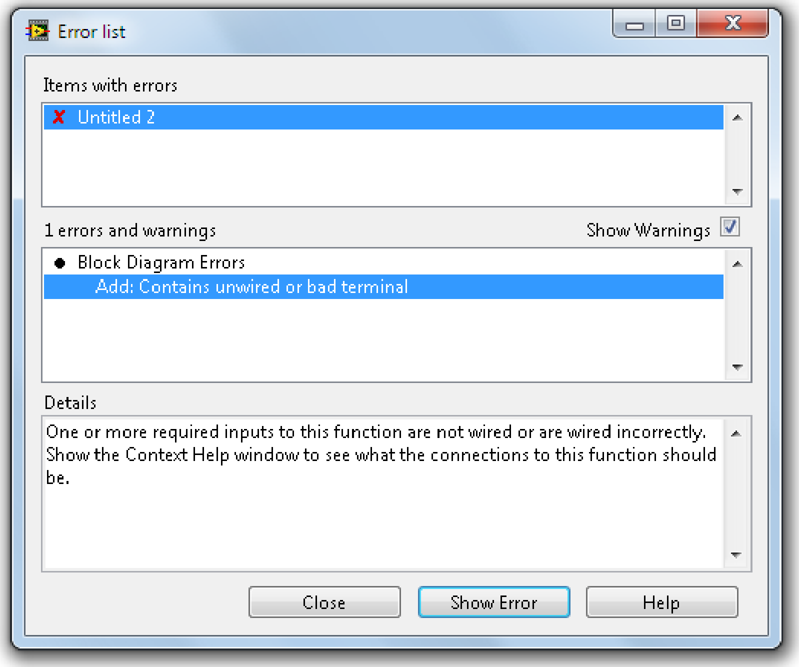

Compiler

```
mov      byte ptr [esi+29h],0
mov      eax,dword ptr [esi+18h]
mov      ebp,dword ptr [esi+14h]
mov      dword ptr [esi+0Ch],eax
cmp      byte ptr [esi+2Ah],1
je       0ABFFE0F
mov      eax,dword ptr [esi+1Ch]
mov      eax,dword ptr [eax+14h]
test     eax,eax
je       0ABFFCEF
cmp      byte ptr [eax+2Ah],1
jne      0ABFFCEF
jmp      0ABFFE0F
mov      ecx,dword ptr [ebp+44h]
xor      eax,eax
mov      edx,1
lock cmpxchg dword ptr [ecx],edx
test     eax,eax
jne      0ABFFCEF
mov      eax,dword ptr [esi+1Ch]
lea      ecx,[ebp+4Ch]
mov      dword ptr [eax+10h],ecx
mov      dword ptr [ebp+68h],eax
mov      dword ptr [ebp+48h],esi
cmp      dword ptr [eax+14h],0
jne      0ABFFD90
mov      dword ptr [eax+14h],esi
mov      byte ptr [ebp+1Eh],1
cmp      dword ptr [esi+30h],2
je       0ABFFE39
mov      byte ptr [ebp+1Bh],1
mov      esi,dword ptr [ebp+360h]
mov      esi,dword ptr [esi]
mov      dword ptr [ebp+37Ch],esi
inc      dword ptr [ebp+37Ch]
mov      esi,dword ptr [ebp+48h]
cmp      byte ptr [esi+3Dh],1
mov      eax,dword ptr [ebp+68h]
je       0ABFFE09
cmp      dword ptr [eax+28h],0
jne      0ABFFE1F
mov      dword ptr [ebp+48h],0
mov      dword ptr [eax+10h],esi
mov      byte ptr [ebp+1Eh],0
mov      ecx,dword ptr [ebp+44h]
mov      dword ptr [ecx],0
cmp      dword ptr [eax+14h],esi
jne      0ABFFE0F
mov      dword ptr [eax+14h],0
cmp      byte ptr [esi+29h],5
jne      0ABFFE0F
mov      dword ptr [esi+29h],2
xor      eax,eax
jmp      0ABFFD13
mov      dword ptr [esi+1Ch],eax
mov      dword ptr [eax+10h],esi
mov      edx,dword ptr [esi+8]
mov      ecx,dword ptr [esi+0Ch]
mov      eax,esi
add      esp,8
pop      esi
mov      ebp,edx
jmp      ecx
add      ebp,3Ch
mov      dword ptr [esp],ebp
call     SubrVIExit (24D6450h)
test     eax,eax
je       0ABFFE02
mov      esi,eax
jmp      0ABFFE0F
mov      byte ptr [ebp+1Bh],0
jmp      0ABFFD90
```

LabVIEW Compiler

- Abstracts the complexities of programming
    - Memory management
    - Thread allocation
    - Language syntax
  - Edit-time semantic analysis
  - Compile on Load/Run/Save





Items with errors

✗ Untitled 2

1 errors and warnings

Show Warnings ☑

● Block Diagram Errors

Add: Contains unwired or bad terminal

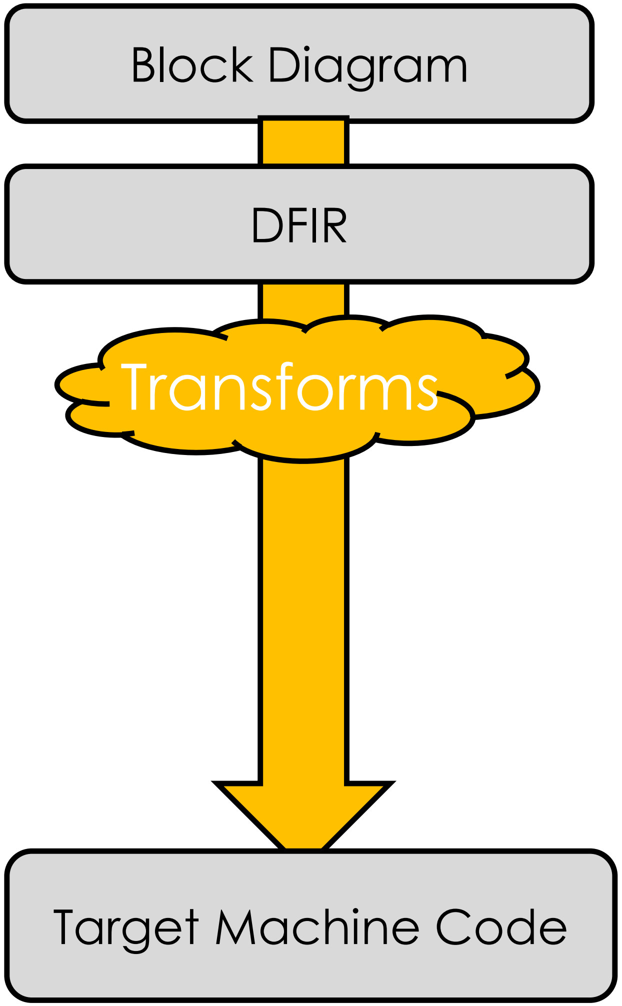
Details

One or more required inputs to this function are not wired or are wired incorrectly. Show the Context Help window to see what the connections to this function should be.

Optimizing the LabVIEW Compiler

DataFlow Intermediate Representation (DFIR)

- High-level graph-based representation
- Preserves execution semantics, dataflow, parallelism, and structure hierarchy
- Developed internally at NI


Block Diagram

DFIR

Transforms

Target Machine Code

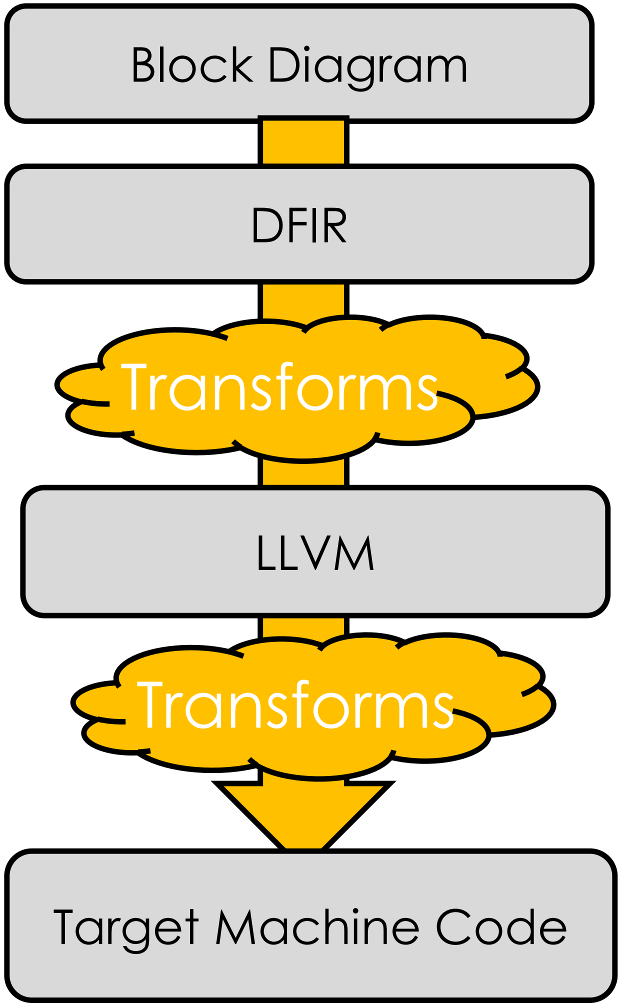
Optimizing the LabVIEW Compiler

DataFlow Intermediate Representation (DFIR)

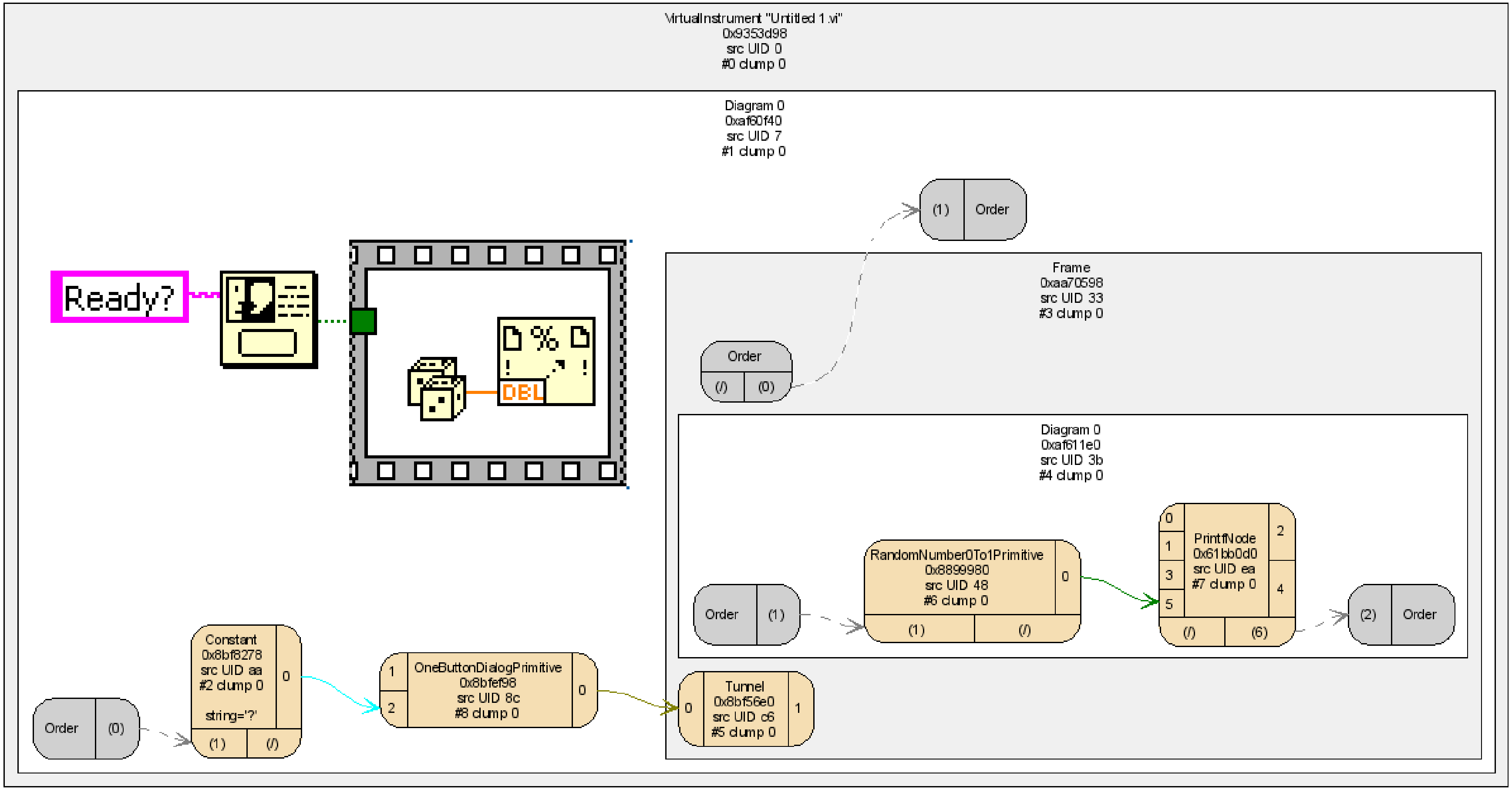
- High-level graph-based representation
- Preserves execution semantics, dataflow, parallelism, and structure hierarchy
- Developed internally at NI

Low-Level Virtual Machine (LLVM)

- Low-level sequential representation
- Knowledge of target machine characteristics
- 3rd party, Open Source

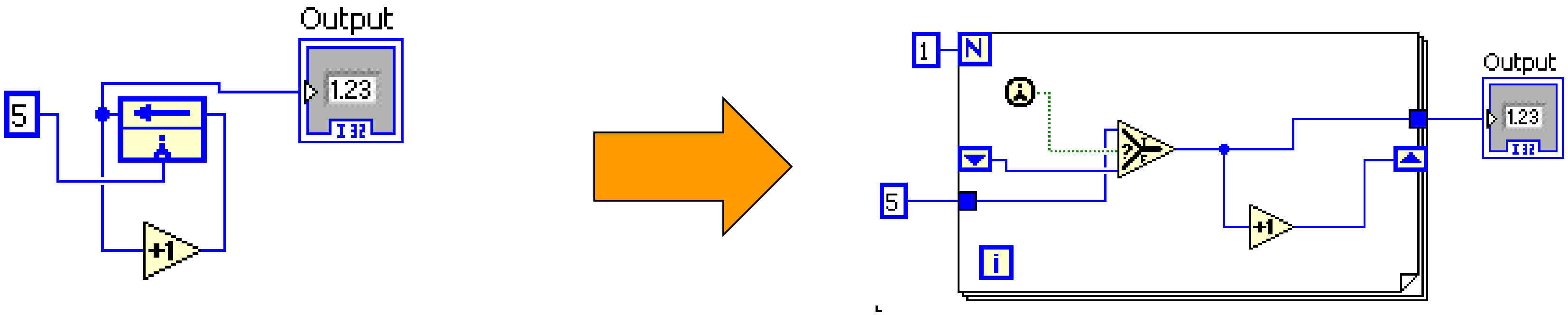


What does DFIR look like?



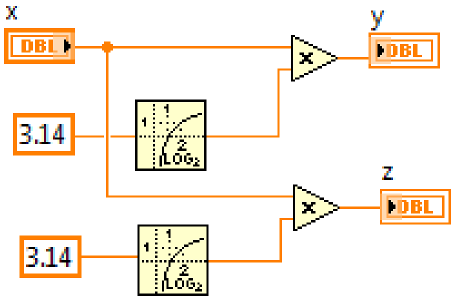
DFIR Decomposition Transforms

- Lowering high-level nodes and constructs
  - equivalent lower-level nodes



Feedback Node Decomposition

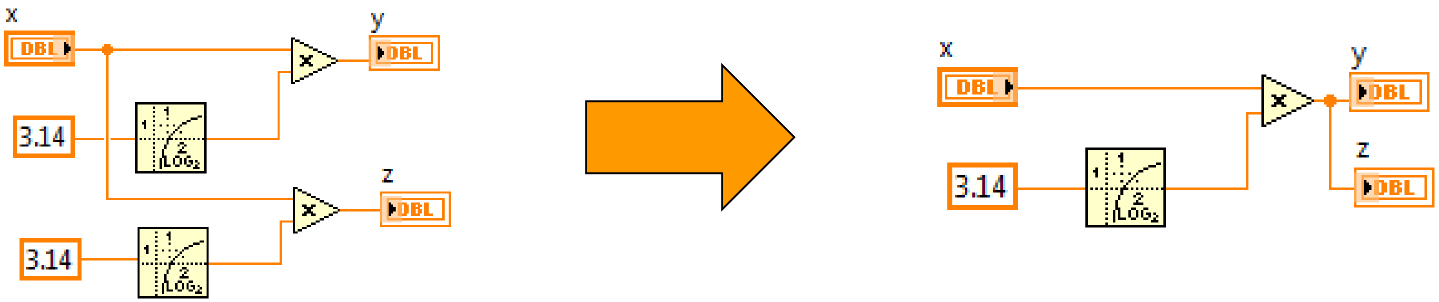
DFIR Optimization Transforms



?

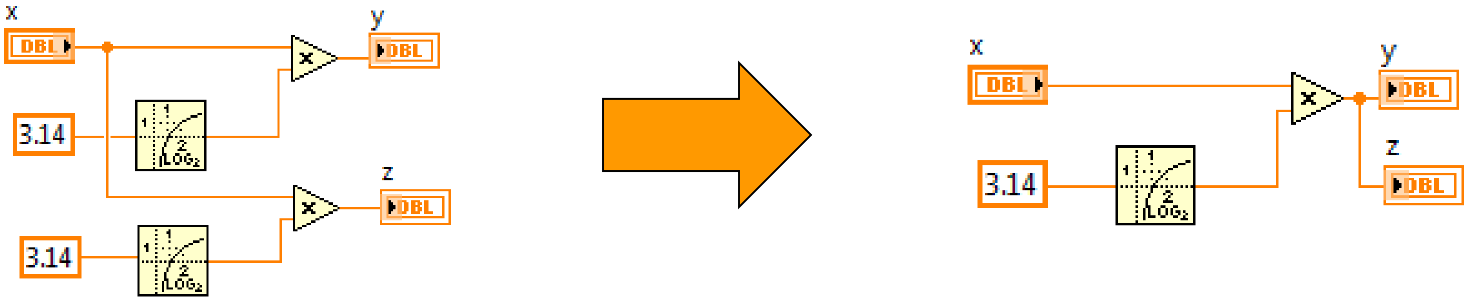
Common Sub-expression Elimination

DFIR Optimization Transforms

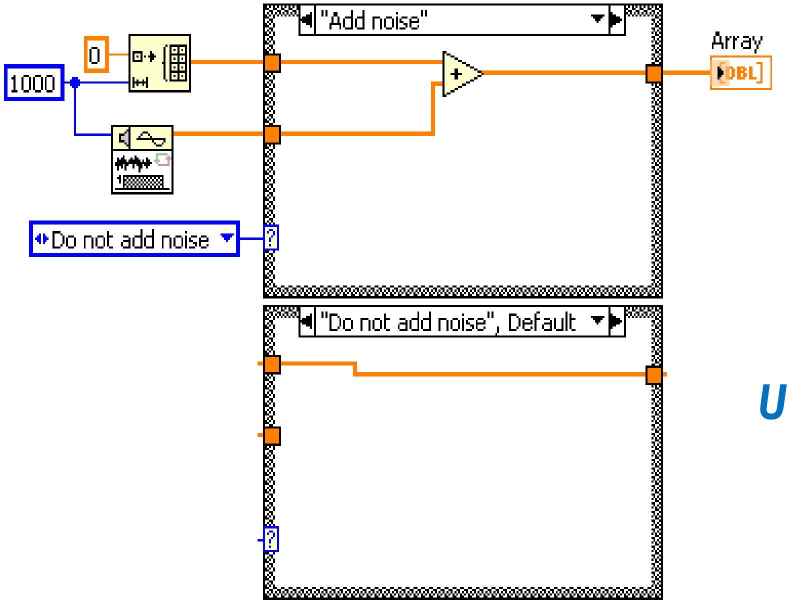


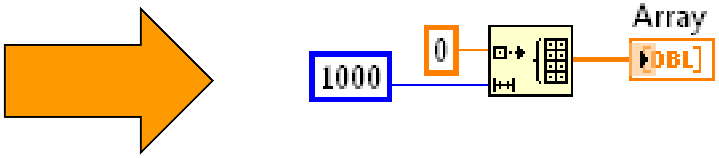
Common Sub-expression Elimination

DFIR Optimization Transforms



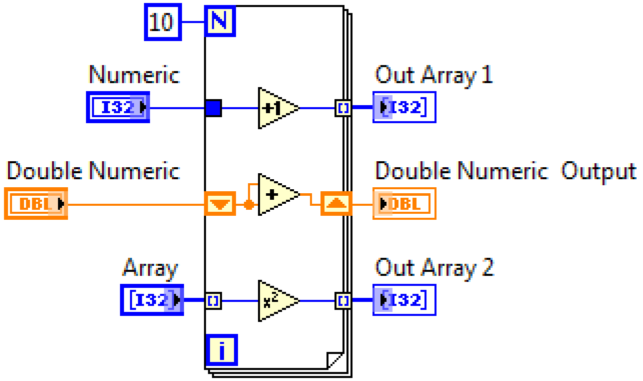
Common Sub-expression Elimination





Unreachable Code Elimination

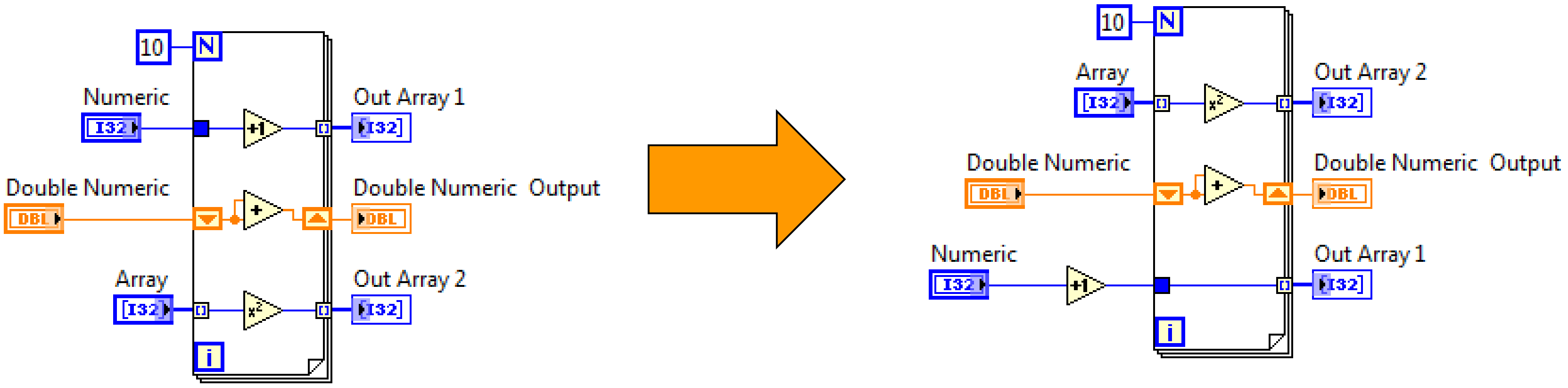
DFIR Optimization Transforms



?

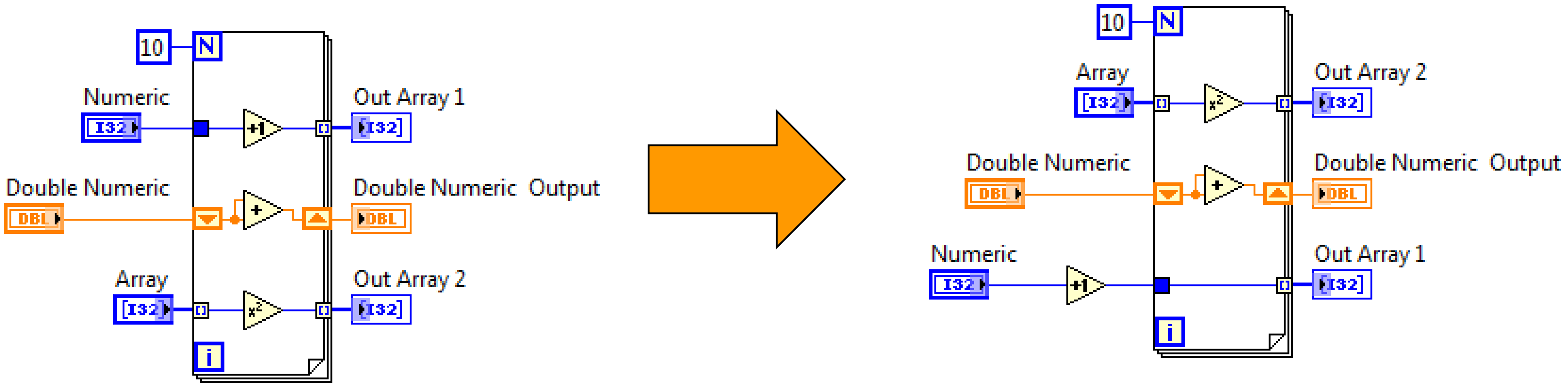
Loop Invariant Code Motion

DFIR Optimization Transforms

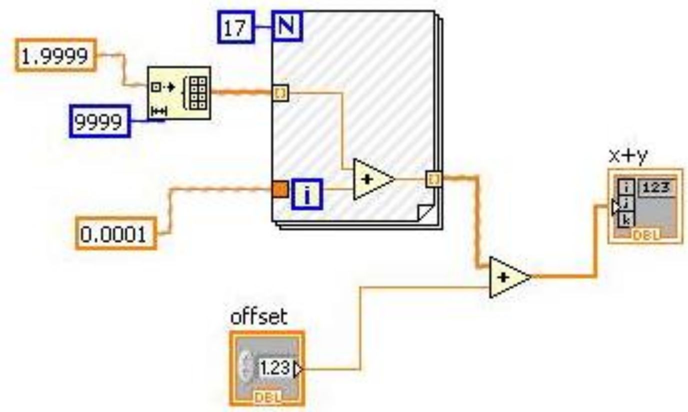


Loop Invariant Code Motion

DFIR Optimization Transforms

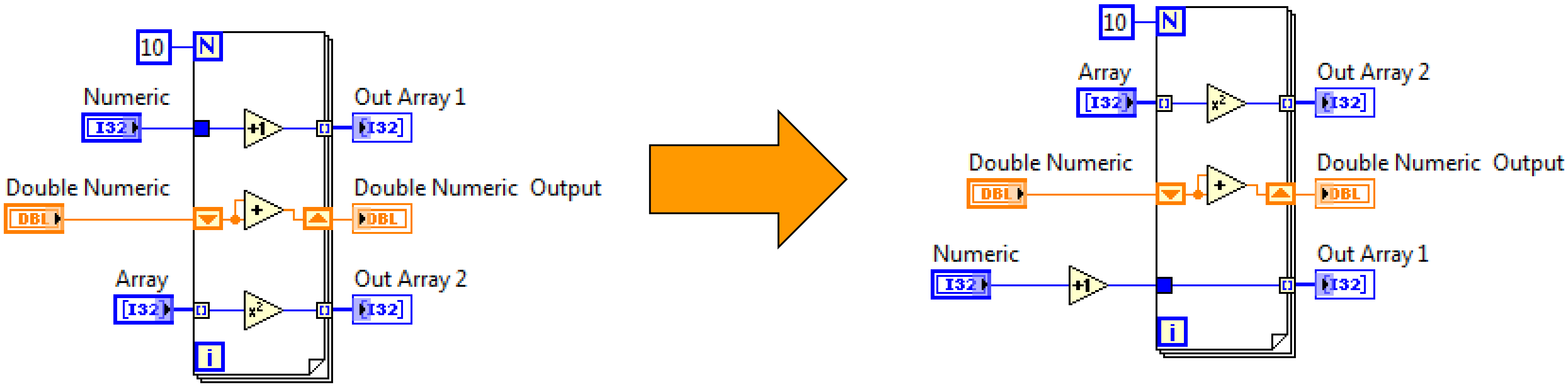


Loop Invariant Code Motion

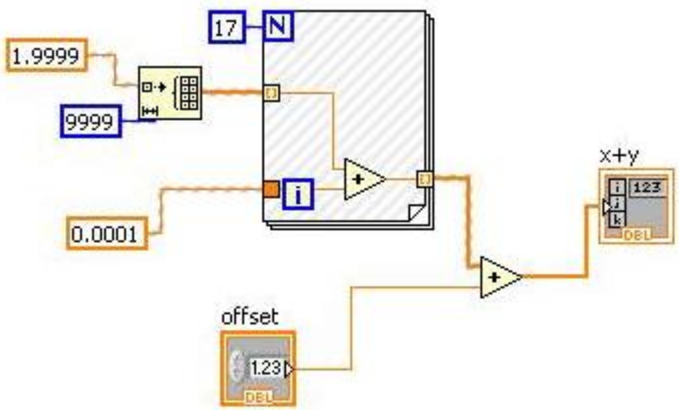


Constant folding

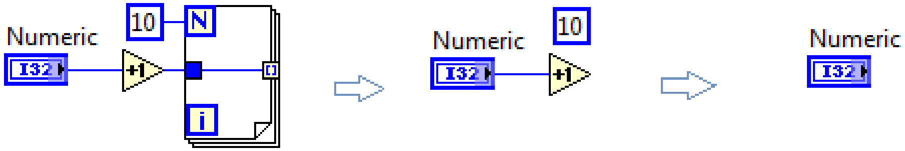
DFIR Optimization Transforms



Loop Invariant Code Motion



Constant folding



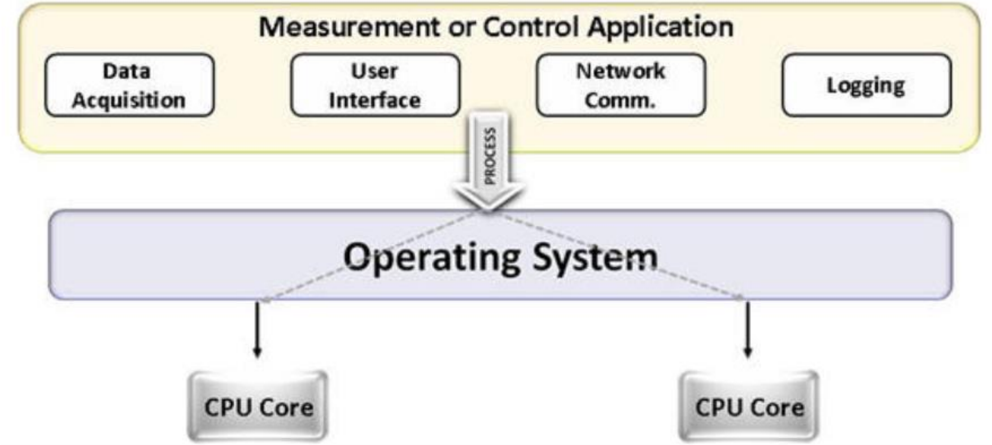
Dead Code Elimination

Outline

- Graphical Dataflow Programming
- LabVIEW – Introduction and Demo
- LabVIEW Compiler (under the hood)
- Multicore Programming in LabVIEW
- Polyhedral Compilation of Graphical Dataflow Programs

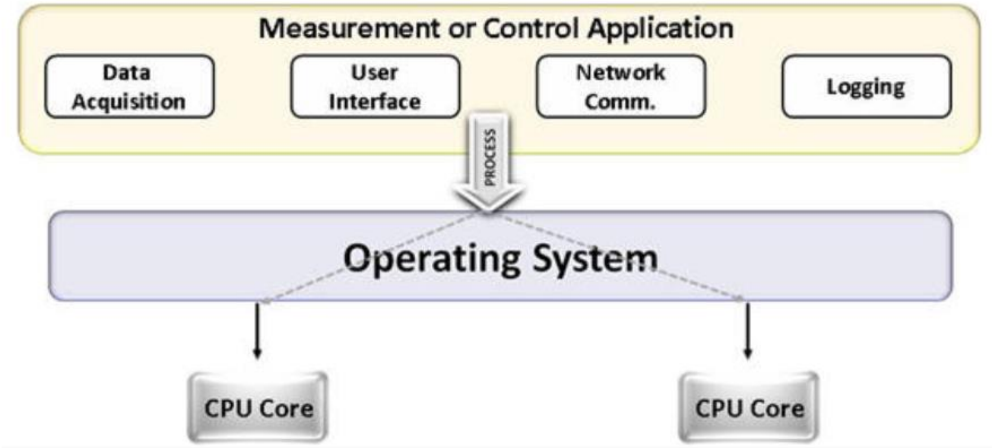
Task Parallelism

- Divide application into independent tasks
  - Tasks mapped to separate processors



Task Parallelism

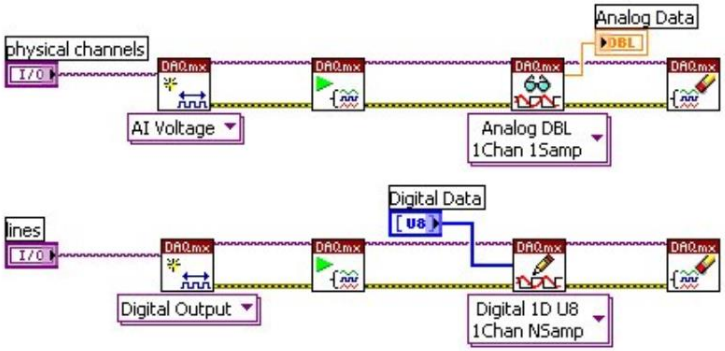
- Divide application into independent tasks
  - Tasks mapped to separate processors



- Traditional text-based languages have sequential syntax
  - Difficult to visualize and organize in parallel form

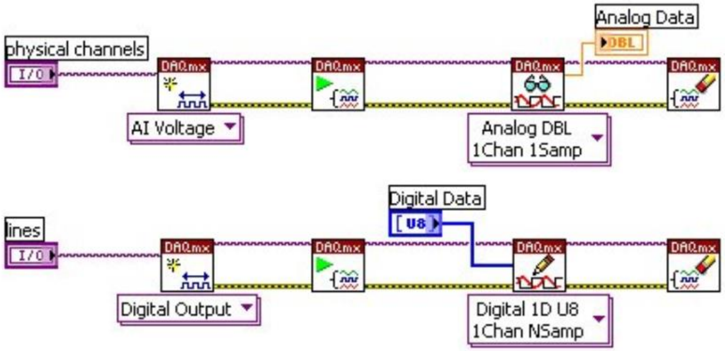
- Parallelism is more evident in graphical dataflow programs
  - Tasks as parallel sections of code on LabVIEW block diagram
  - No need to manage threads or their synchronization

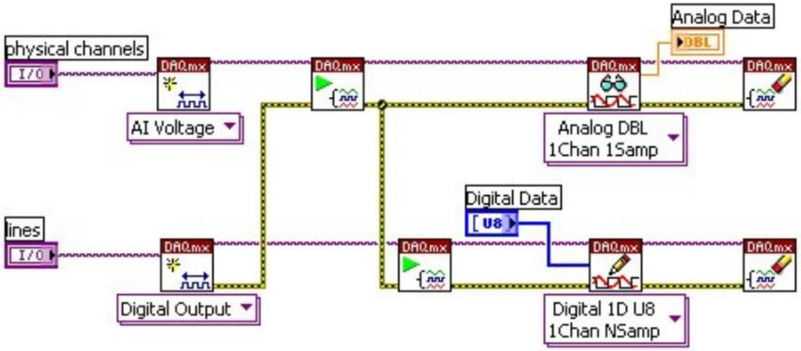
Task Parallelism – An Example



- Independent data acquisition tasks
- Can be executed concurrently on multicore processor

Task Parallelism – An Example With Pitfalls





- Independent data acquisition tasks
- Can be executed concurrently on multicore processor

- Tasks not truly parallel
- Digital task depends on analog task

To maximize task parallelism, avoid unnecessary resource sharing

Multi-threaded LabVIEW Execution Environment

- LabVIEW compiler identifies *clumps*
    - Parallel sections of code on block diagram

Multi-threaded LabVIEW Execution Environment

- LabVIEW compiler identifies *clumps*
  - Parallel sections of code on block diagram

- LabVIEW runtime maintains pool of execution threads
  - Pool size at least as much as number of cores
  - During sequential run, some threads are asleep
  - Idle threads get woken up as degree of parallelism increases

Multi-threaded LabVIEW Execution Environment

- LabVIEW compiler identifies *clumps*
  - Parallel sections of code on block diagram

- LabVIEW runtime maintains pool of execution threads
  - Pool size at least as much as number of cores
  - During sequential run, some threads are asleep
  - Idle threads get woken up as degree of parallelism increases

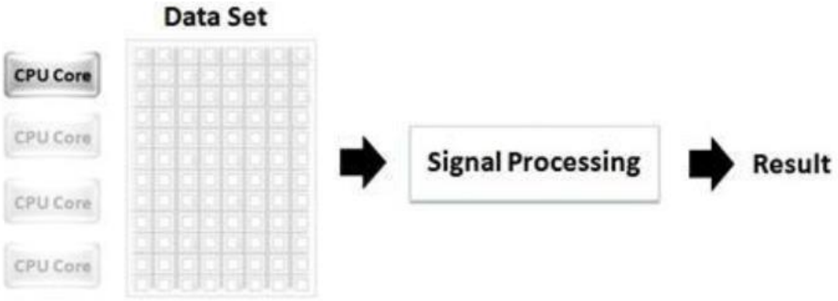
- Thread co-operatively multitasks across clumps
  - Clumps yield periodically to scheduler
  - Waiting clumps get chance to run

Data Parallelism

- Split large dataset into smaller chunks
  - Operate on smaller chunks in parallel
  - Individual results are combined to obtain final result

Data Parallelism

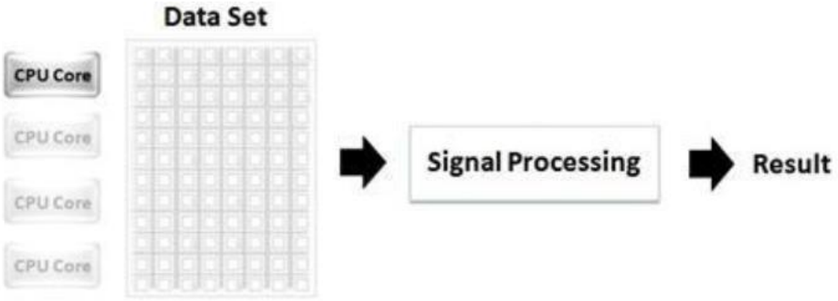
- Split large dataset into smaller chunks
  - Operate on smaller chunks in parallel
  - Individual results are combined to obtain final result



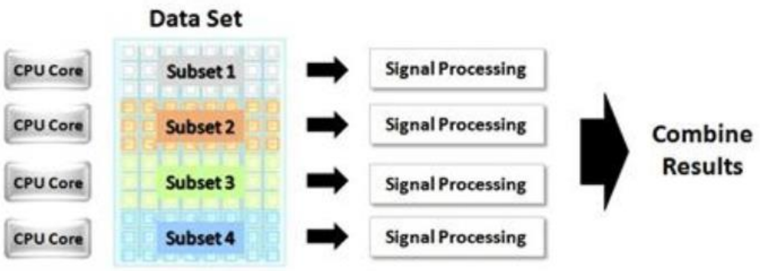
- No data parallelism
- Inefficient use of resources

Data Parallelism

- Split large dataset into smaller chunks
  - Operate on smaller chunks in parallel
  - Individual results are combined to obtain final result

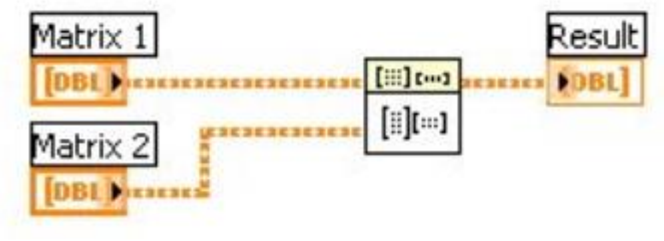


- No data parallelism
- Inefficient use of resources



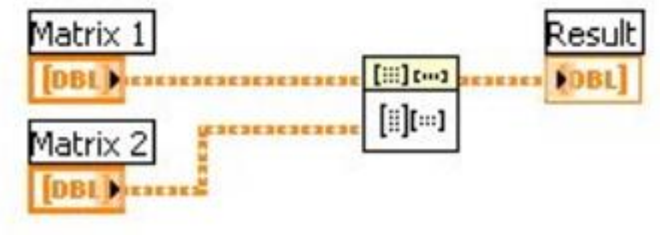
- Large dataset broken up into 4 subsets
- Each core is engaged
- Improved execution speed

Data Parallelism in LabVIEW



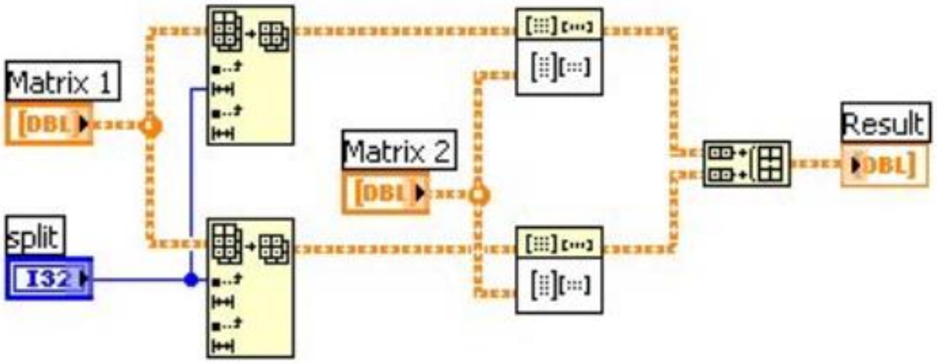
- Standard matmul operation in LabVIEW
- No data parallelism being exploited
- Long execution time for large datasets

Data Parallelism in LabVIEW

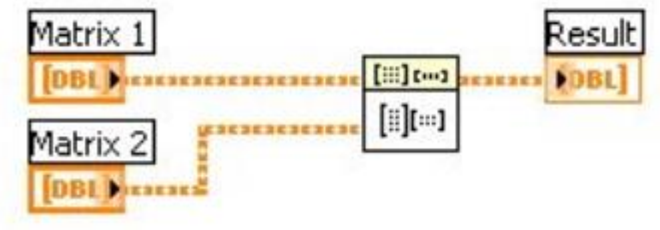


- Standard matmul operation in LabVIEW
- No data parallelism being exploited
- Long execution time for large datasets

- Data parallel matmul
- Matrix1 divided into two halves
- Concurrent matmul with each half
- Individual results combined

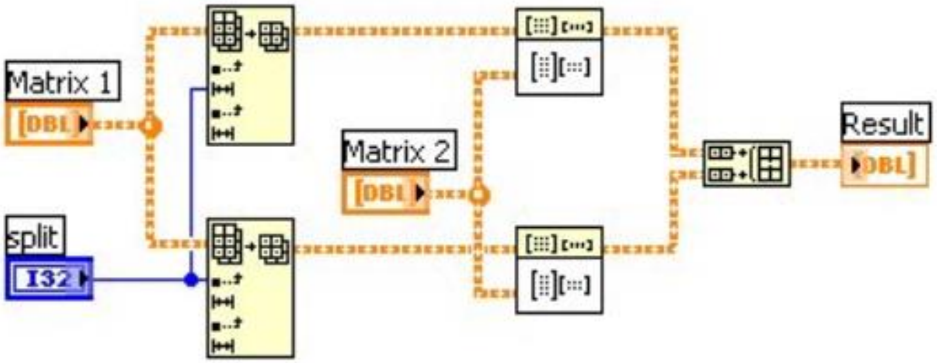


Data Parallelism in LabVIEW



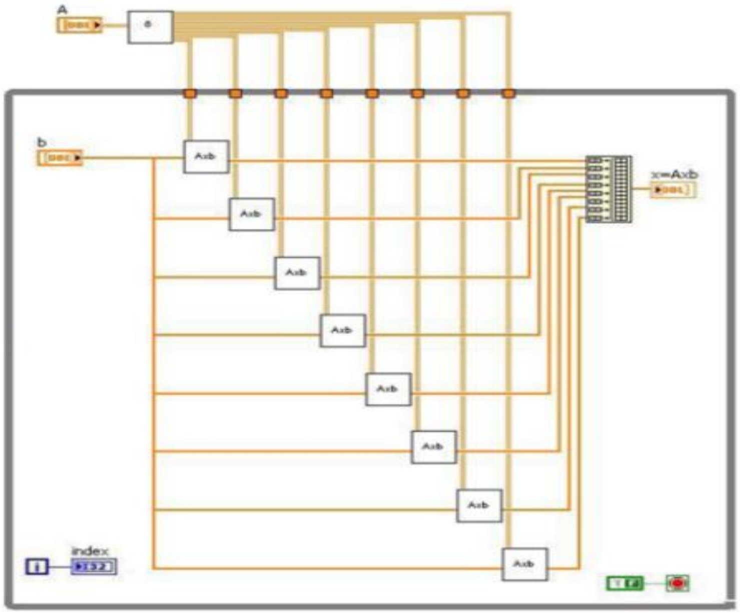
- Standard matmul operation in LabVIEW
- No data parallelism being exploited
- Long execution time for large datasets

- Data parallel matmul
- Matrix1 divided into two halves
- Concurrent matmul with each half
- Individual results combined



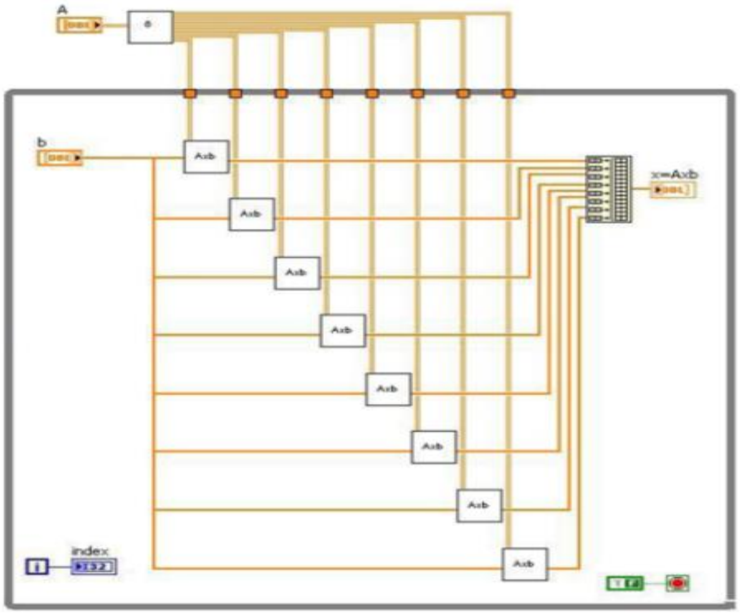
	Execution Time on Single Core Processor	Execution Time on Dual Core Processor
Matrix Multiplication without Data Parallelism	1.195 seconds	1.159 seconds
Matrix Multiplication with Data Parallelism	1.224 seconds	0.629 seconds

Data Parallelism in the Real World



- Matrix-vector in real-time HPC application e.g. control system
- Sensor measurements as vector input on per-loop basis
- Matrix-vector result to control actuators
- Matrix-vector computation on 8 cores

Data Parallelism in the Real World



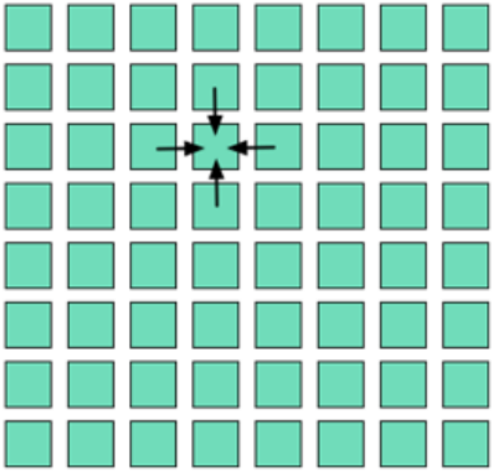
- Matrix-vector in real-time HPC application e.g. control system
- Sensor measurements as vector input on per-loop basis
- Matrix-vector result to control actuators
- Matrix-vector computation on 8 cores

LabVIEW program for ***plasma control in ASDEX tokamak***

- Germany’s most advanced nuclear fusion platform
- Compute-intensive matrix operations on oct-core server
- Real-time constraint of maintaining a 1ms control loop

“in first design stage...with LabVIEW, we obtained a ***20X processing speedup on an octal core processor machine over a single-core processor****, while reaching our 1 ms control loop requirement”* -- Louis Giannone, lead researcher

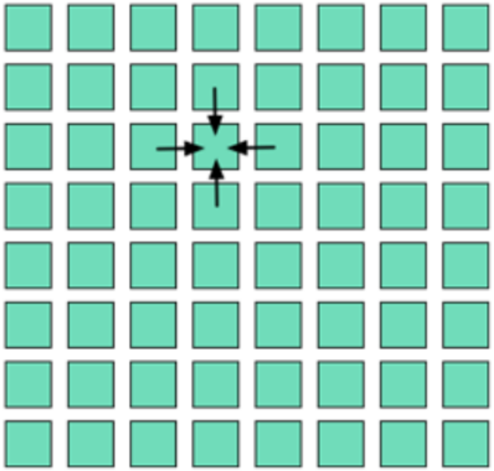
Structured Grids



Near-neighbor dependences in time-iterated stencil computations

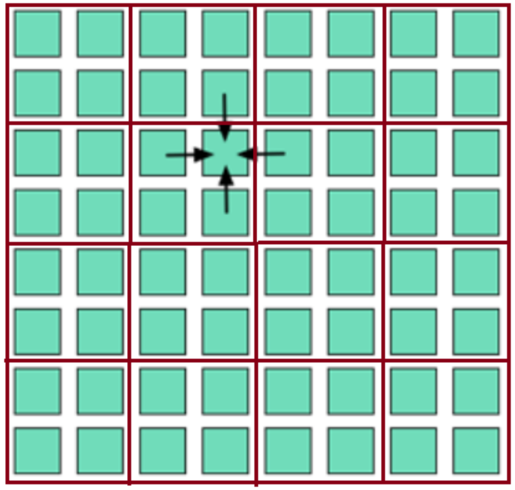
```
for(t = 1; t < T; ++t)
   for(i = 1; i < N; ++i)
       for(j = 1; j < N; ++j)
          grid[t][i][j] = f(grid[t-1][i-1][j],
                            grid[t-1][i+1][j],
                            grid[t-1][i][j-1],
                            grid[t-1][i][j+1]);
```

Structured Grids



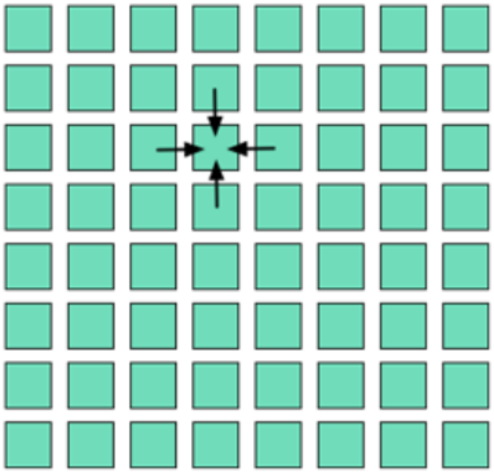
Near-neighbor dependences in time-iterated stencil computations

```
for(t = 1; t < T; ++t)
   for(i = 1; i < N; ++i)
       for(j = 1; j < N; ++j)
           grid[t][i][j] = f(grid[t-1][i-1][j],
                             grid[t-1][i+1][j],
                             grid[t-1][i][j-1],
                             grid[t-1][i][j+1]);
```



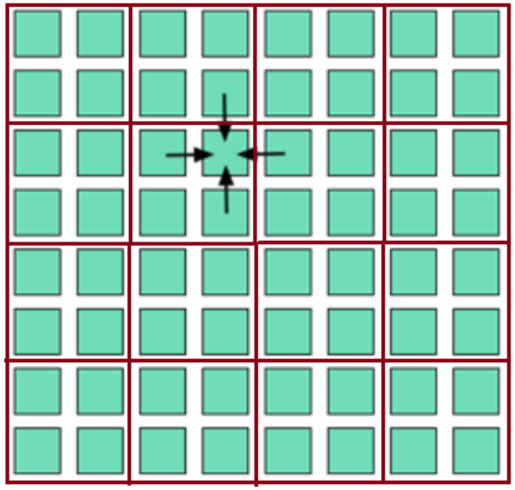
- Split into sub-grids
- Compute them independently

Structured Grids



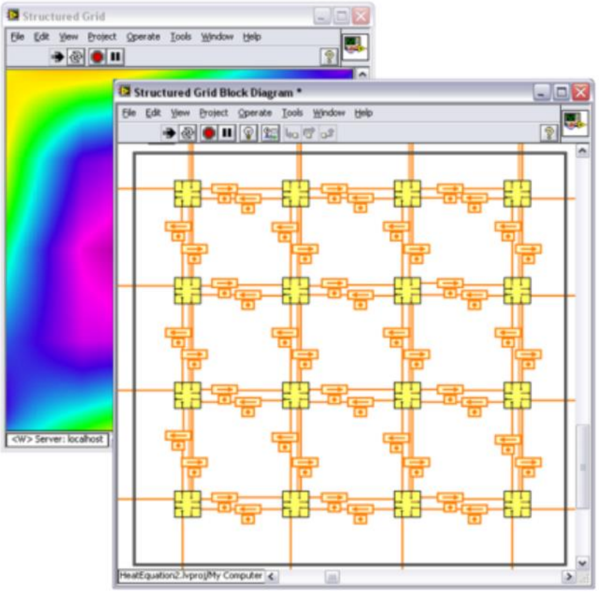
Near-neighbor dependences in time-iterated stencil computations

```
for(t = 1; t < T; ++t)
   for(i = 1; i < N; ++i)
       for(j = 1; j < N; ++j)
           grid[t][i][j] = f(grid[t-1][i-1][j],
                             grid[t-1][i+1][j],
                             grid[t-1][i][j-1],
                             grid[t-1][i][j+1]);
```



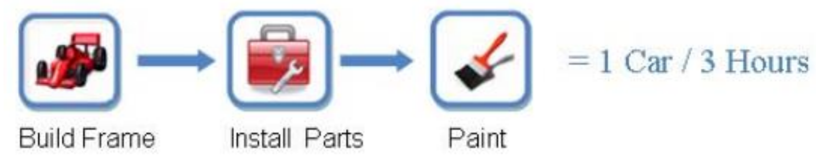
- Split into sub-grids
- Compute them independently

- Each icon mapped to separate core
- Feedback nodes represent data exchange


Structured Grid
File Edit View Project Operate Tools Window Help
<W> Server: localhost
Structured Grid Block Diagram *
File Edit View Project Operate Tools Window Help
HeatEquation2.lvproj/My Computer

Pipelining

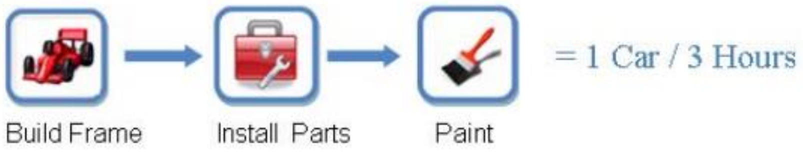
- Divide inherently serial task into concrete stages
- Execute stages in assembly-line fashion



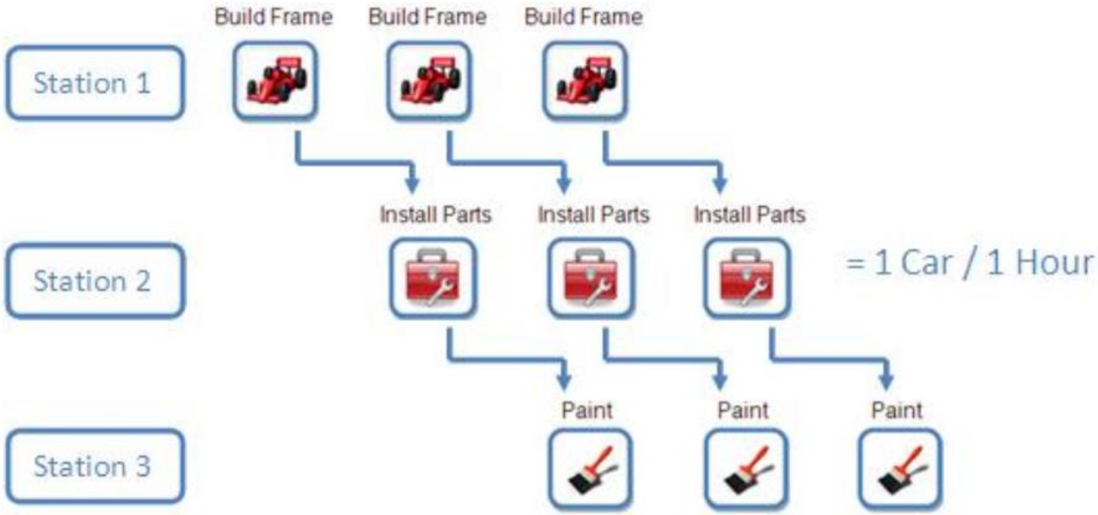
- **No pipelining**
- **Poor throughput**

Pipelining

- Divide inherently serial task into concrete stages
- Execute stages in assembly-line fashion

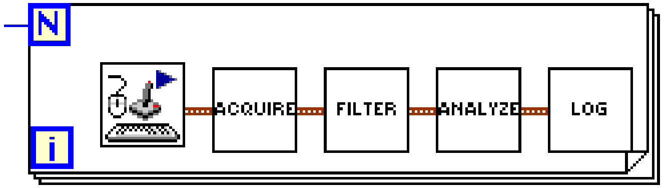


- **No pipelining**
- **Poor throughput**



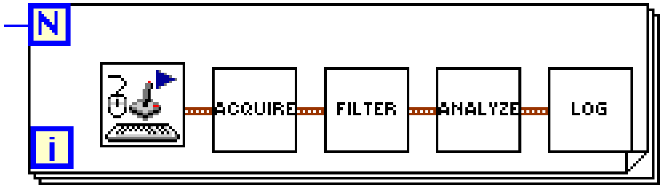
- **Pipelined execution**
- **Improved throughput**

Pipelining in LabVIEW

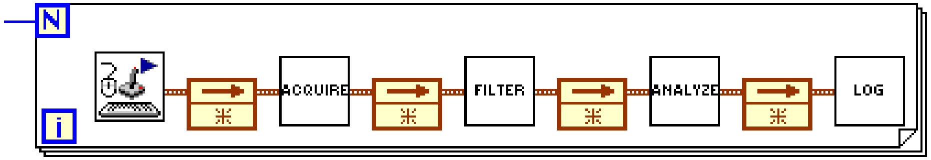


- Sequential task in a loop, with 4 stages
- Typical of streaming applications
  - FFTs manipulated one step at a time

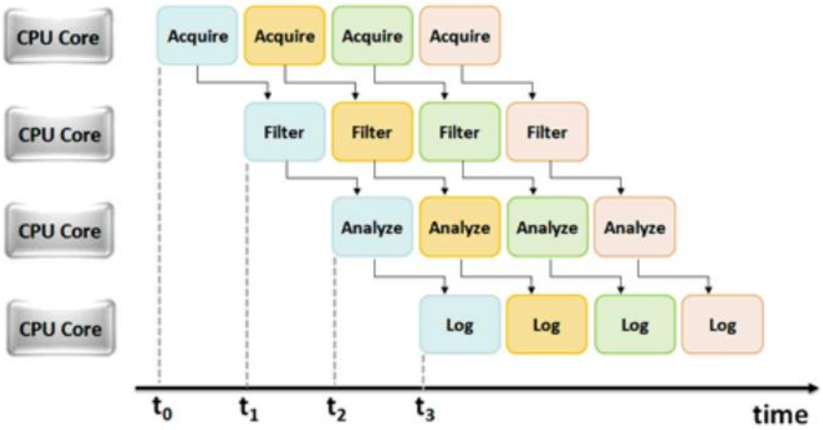
Pipelining in LabVIEW



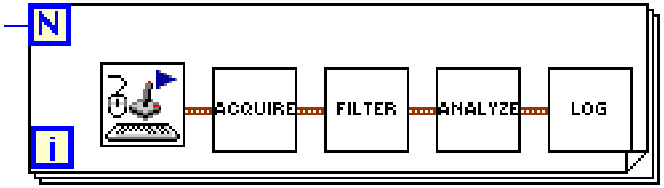
- Sequential task in a loop, with 4 stages
- Typical of streaming applications
  - FFTs manipulated one step at a time



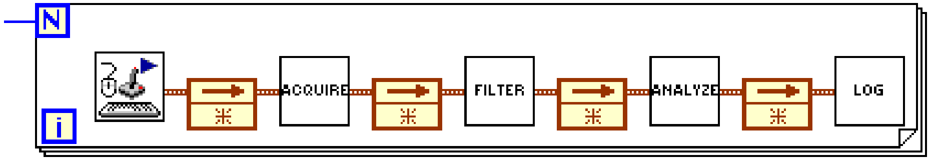
- Feedback nodes to separate pipeline stages



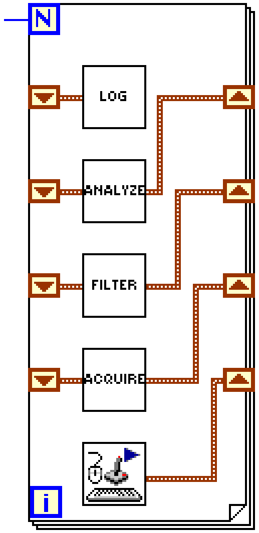
Pipelining in LabVIEW



- Sequential task in a loop, with 4 stages
- Typical of streaming applications
    - FFTs manipulated one step at a time

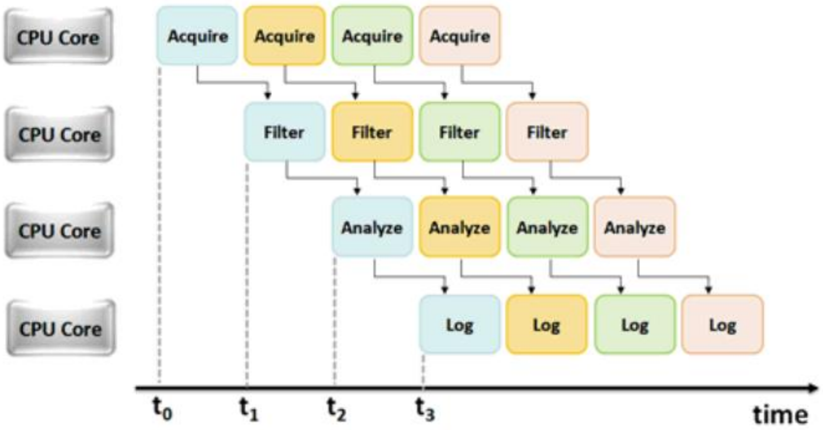


- Feedback nodes to separate pipeline stages

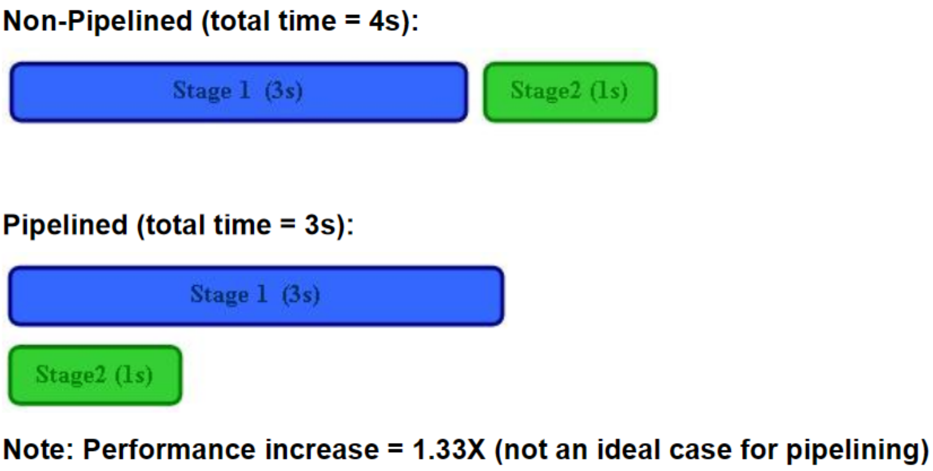


- Pipelined execution through shift registers

- Each stage can be mapped to a separate core



Pipelining – Important Concerns



Non-Pipelined (total time = 4s):

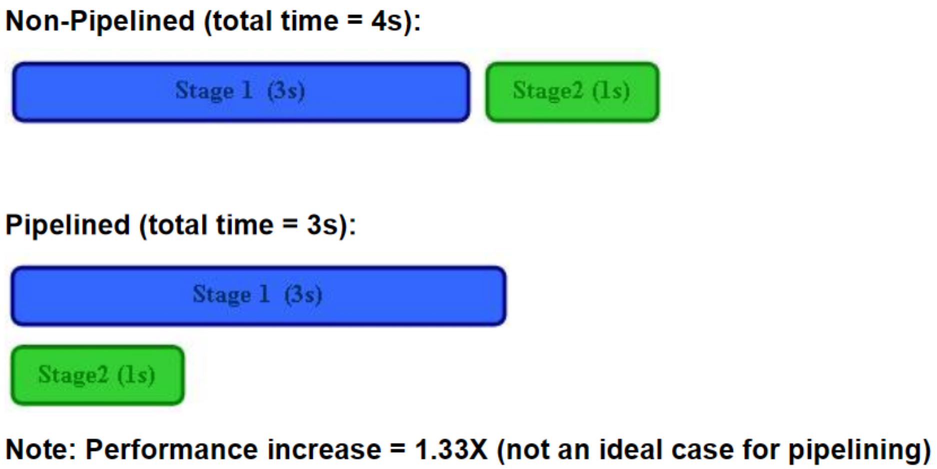
Pipelined (total time = 3s):

Note: Performance increase = 1.33X (not an ideal case for pipelining)

Pipeline stages must be well-balanced

LabVIEW built-in timing primitives for benchmarking

Pipelining – Important Concerns



Non-Pipelined (total time = 4s):

Pipelined (total time = 3s):

Note: Performance increase = 1.33X (not an ideal case for pipelining)

Pipeline stages must be well-balanced

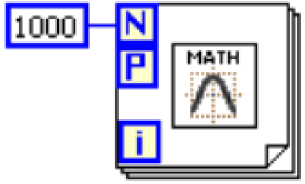
LabVIEW built-in timing primitives for benchmarking

Avoid large data transfer between stages, across cores →

- Cores may not share cache
- Data size could exceed cache size

Parallel For Loop for Iteration Parallelism

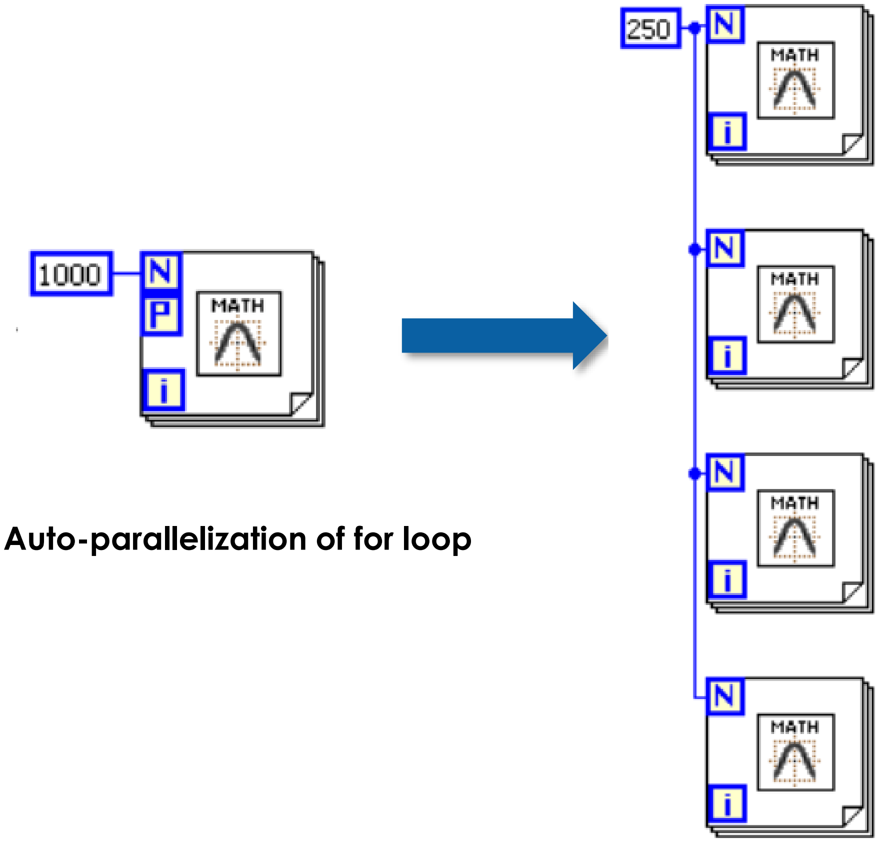
- Concurrent execution iterations of a for loop in multiple threads
  - Greater CPU utilization



Auto-parallelization of for loop

Parallel For Loop for Iteration Parallelism

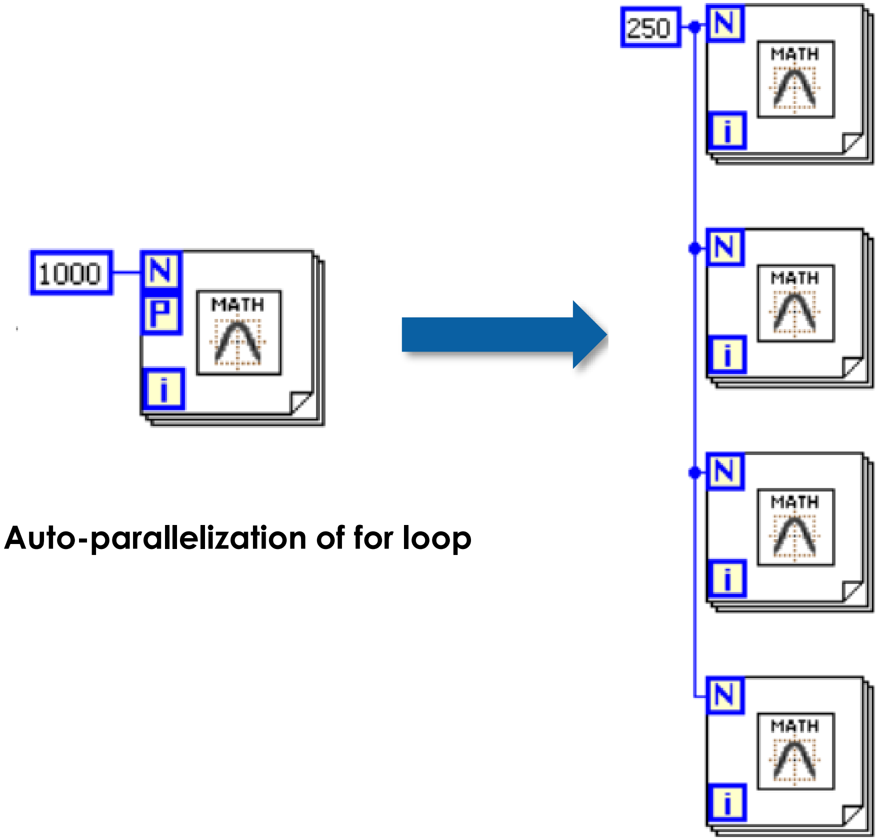
- Concurrent execution iterations of a for loop in multiple threads
  - Greater CPU utilization



Auto-parallelization of for loop

Parallel For Loop for Iteration Parallelism

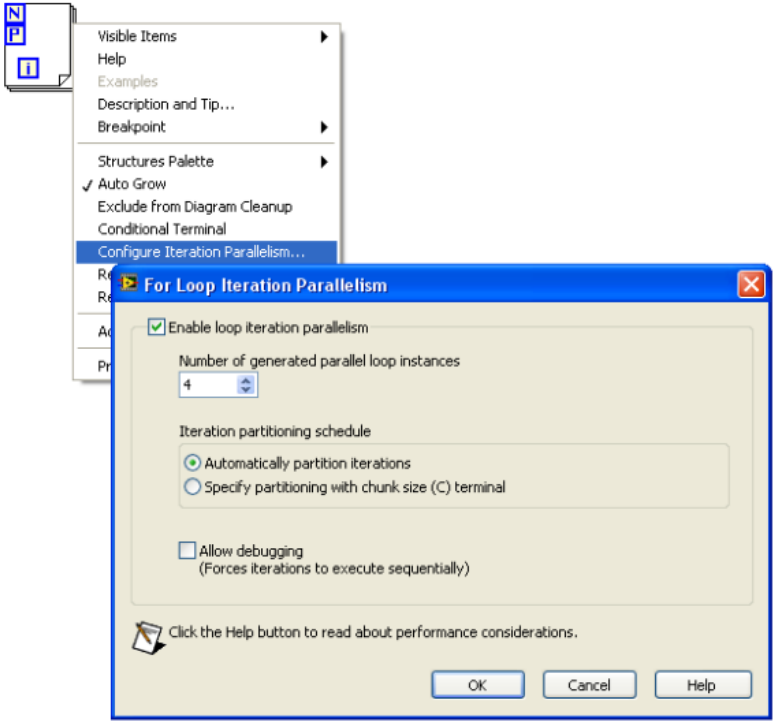
- Concurrent execution iterations of a for loop in multiple threads
  - Greater CPU utilization



Auto-parallelization of for loop

- Compiler generate multiple parallel loop instances
- Each parallel loop instance represents independently schedulable clump

Configuring Iteration Parallelism



For Loop Iteration Parallelism

☑ Enable loop iteration parallelism

Number of generated parallel loop instances: 4

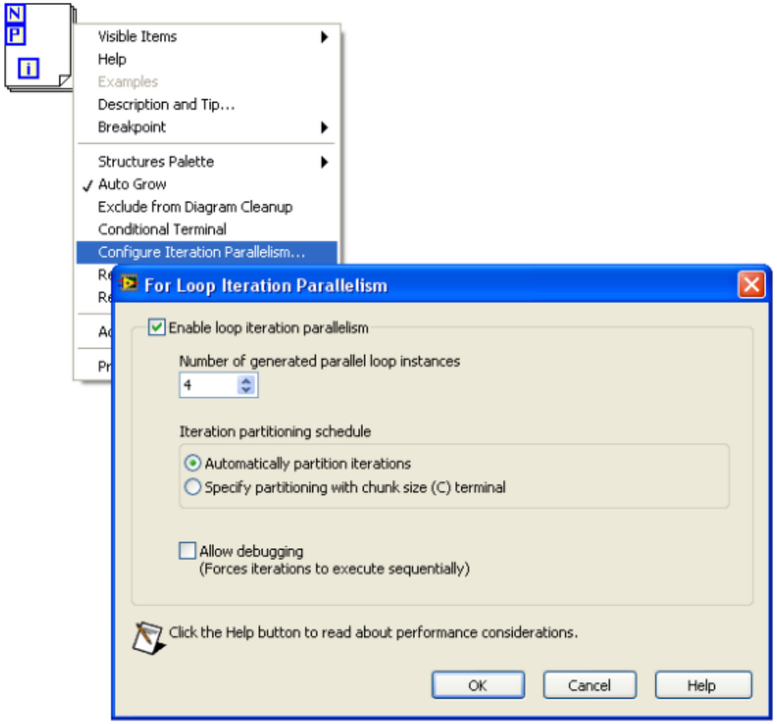
Iteration partitioning schedule

● Automatically partition iterations
○ Specify partitioning with chunk size (C) terminal

□ Allow debugging
(Forces iterations to execute sequentially)

Click the Help button to read about performance considerations.

Configuring Iteration Parallelism



For Loop Iteration Parallelism

☑ Enable loop iteration parallelism

Number of generated parallel loop instances: 4

Iteration partitioning schedule

● Automatically partition iterations
○ Specify partitioning with chunk size (C) terminal

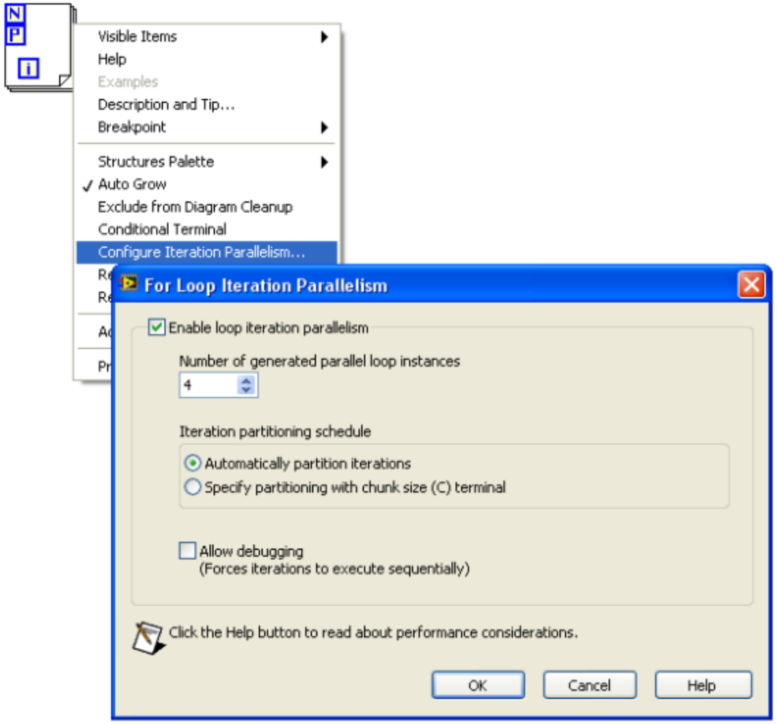
□ Allow debugging
(Forces iterations to execute sequentially)

Click the Help button to read about performance considerations.

Automatic iteration partitioning

- Initial chunks of iterations are large (reduces scheduling overhead)
- Chunk size gradually decreases (better load balancing)

Configuring Iteration Parallelism



For Loop Iteration Parallelism

☑ Enable loop iteration parallelism

Number of generated parallel loop instances: 4

Iteration partitioning schedule

● Automatically partition iterations

○ Specify partitioning with chunk size (C) terminal

□ Allow debugging
(Forces iterations to execute sequentially)

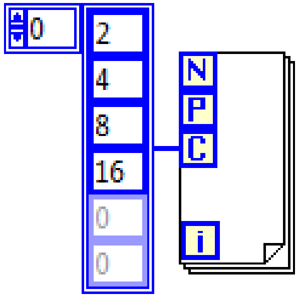
Click the Help button to read about performance considerations.

Automatic iteration partitioning

- Initial chunks of iterations are large (reduces scheduling overhead)
- Chunk size gradually decreases (better load balancing)

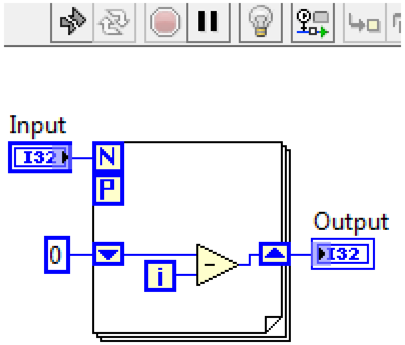
Customized iteration partitioning

- Wire in chunk size or array of chunk sizes to the **C** terminal



Iteration Parallelism – When to Use?

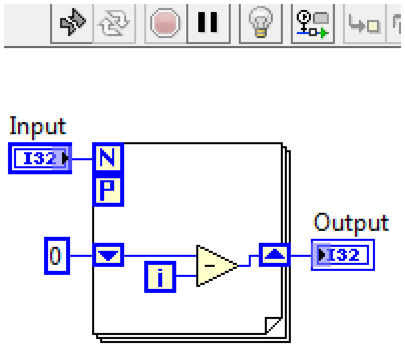
Loop must produce same result regardless of order of execution of iterations



Data carried across iterations through shift registers

Iteration Parallelism – When to Use?

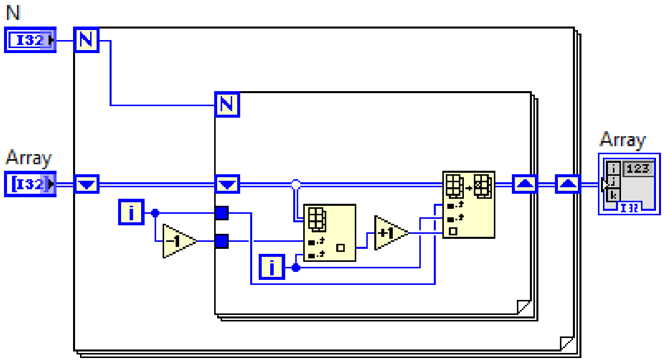
Loop must produce same result regardless of order of execution of iterations



Data carried across iterations through shift registers

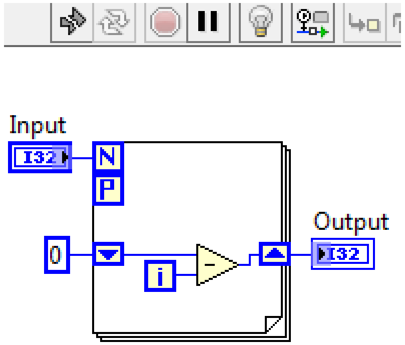
```
for (int i = 1; i < N; ++i)
      for (int j = 1; j < N; ++j)
            a[i][j] = a[i-1][j] + 1;
```

Can any loop be parallelized here?



Iteration Parallelism – When to Use?

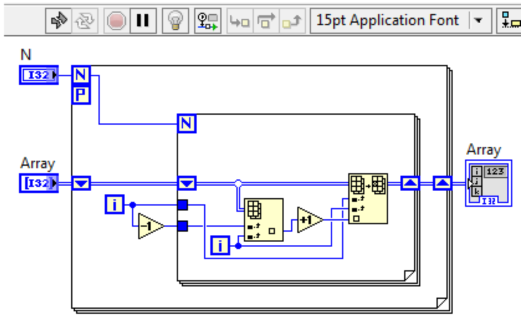
Loop must produce same result regardless of order of execution of iterations

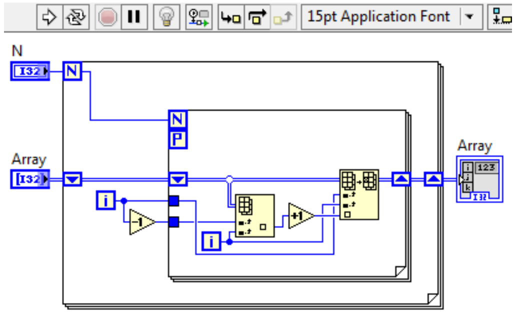


Data carried across iterations through shift registers

```
for (int i = 1; i < N; ++i)
      for (int j = 1; j < N; ++i)
            a[i][j] = a[i-1][j] + 1;
```

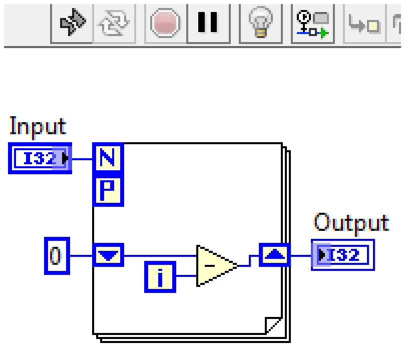
Can any loop be parallelized here?





Iteration Parallelism – When to Use?

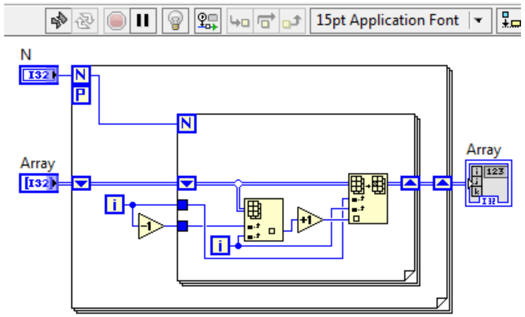
Loop must produce same result regardless of order of execution of iterations

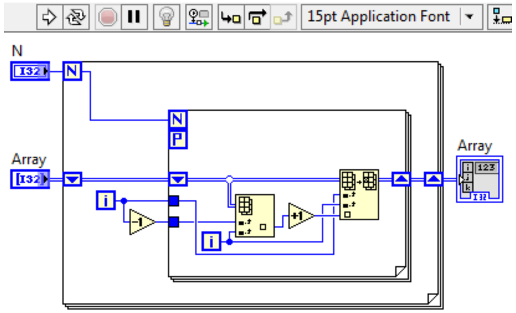


Data carried across iterations through shift registers

```
for (int i = 1; i < N; ++i)
      for (int j = 1; j < N; ++i)
            a[i][j] = a[i-1][j] + 1;

Can any loop be parallelized here?
```





LabVIEW automatically does
cross-iteration dependence analysis

- VI breaks if dependences are violated

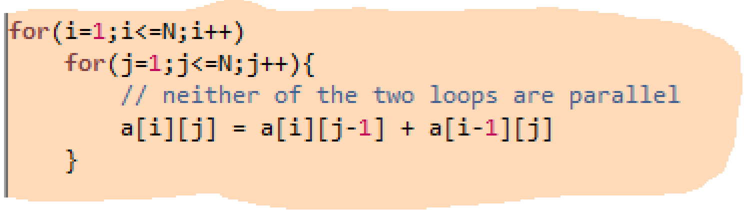
One iteration should not depend on results of another

- Writing A[i-1] in iteration i-1
- Reading A[i-1] in iteration (i)

Outline

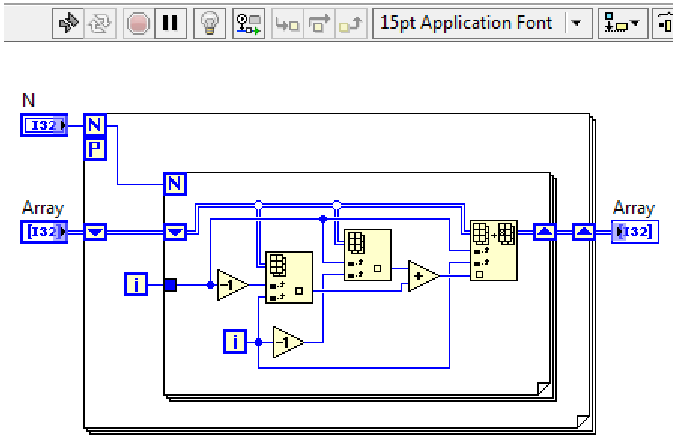
- Graphical Dataflow Programming
- LabVIEW – Introduction and Demo
- LabVIEW Compiler (under the hood)
- Multicore Programming in LabVIEW
- Polyhedral Compilation of Graphical Dataflow Programs

Parallel For Loop Limitations



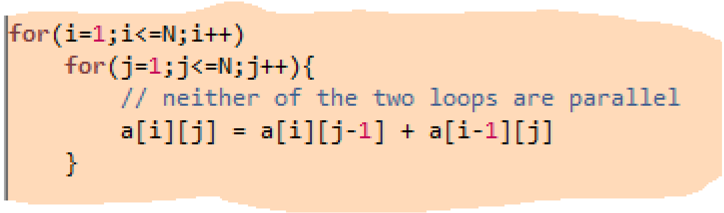
```
for(i=1;i<=N;i++)
    for(j=1;j<=N;j++){
        // neither of the two loops are parallel
        a[i][j] = a[i][j-1] + a[i-1][j]
    }
```

None of these loops can be parallelized

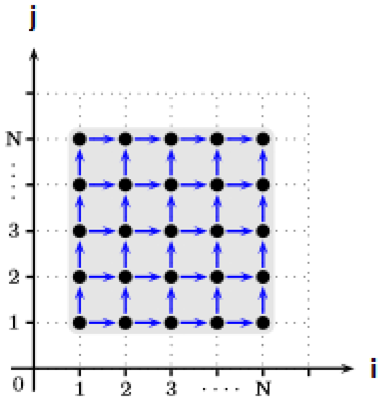


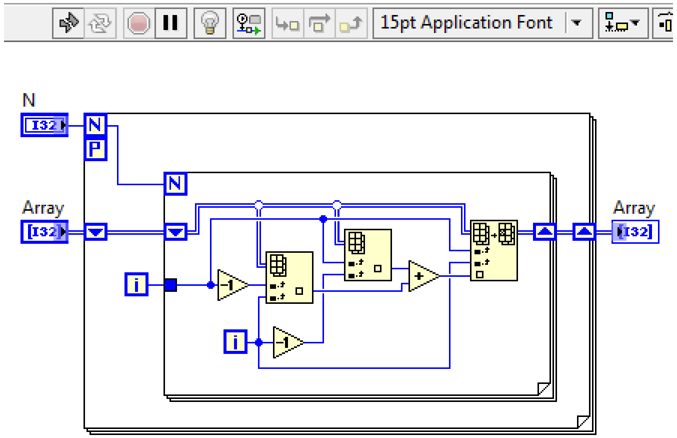
Loop-nest is inner parallel

Parallel For Loop Limitations



```
for(i=1;i<=N;i++)
    for(j=1;j<=N;j++){
        // neither of the two loops are parallel
        a[i][j] = a[i][j-1] + a[i-1][j]
    }
```

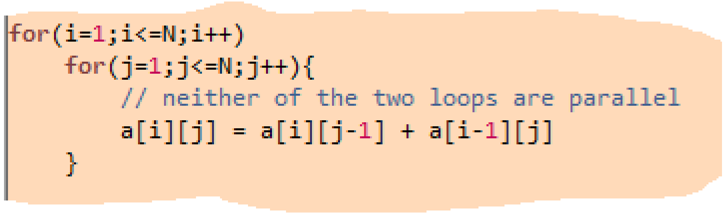




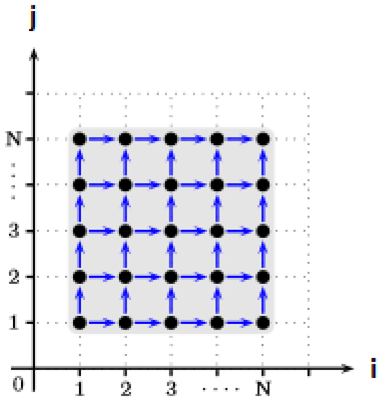
None of these loops can be parallelized

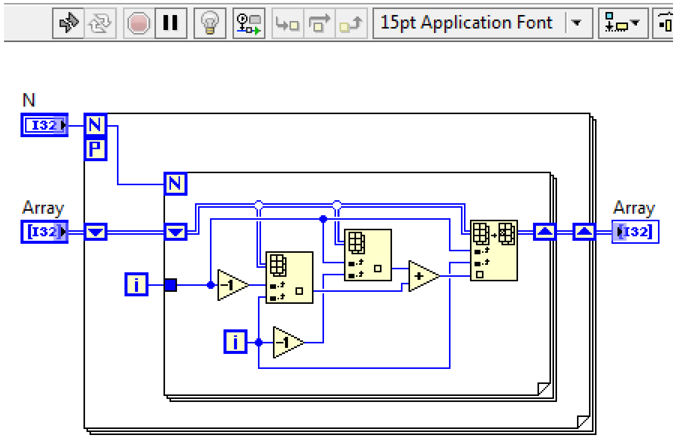
Loop-nest is inner parallel

Parallel For Loop Limitations

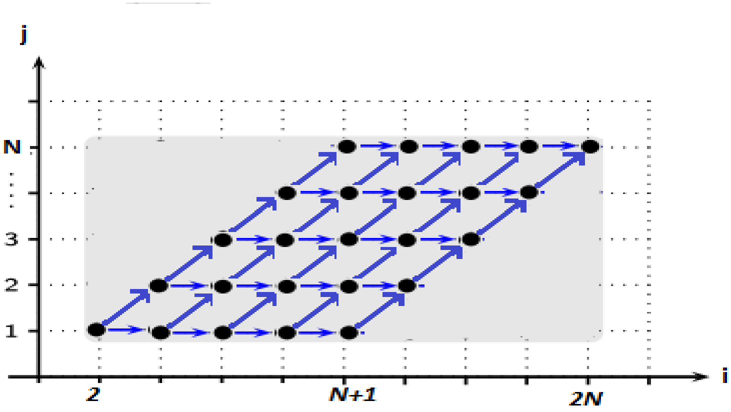


```
for(i=1;i<=N;i++)
    for(j=1;j<=N;j++){
        // neither of the two loops are parallel
        a[i][j] = a[i][j-1] + a[i-1][j]
    }
```



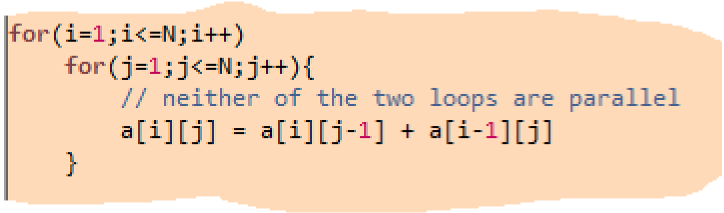


None of these loops can be parallelized



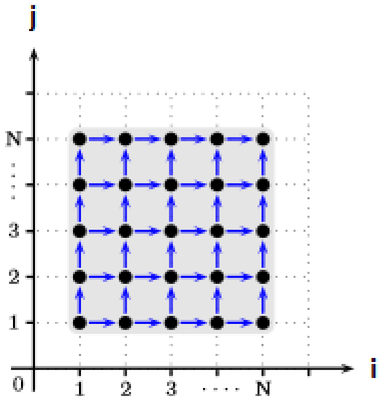
Loop-nest is inner parallel

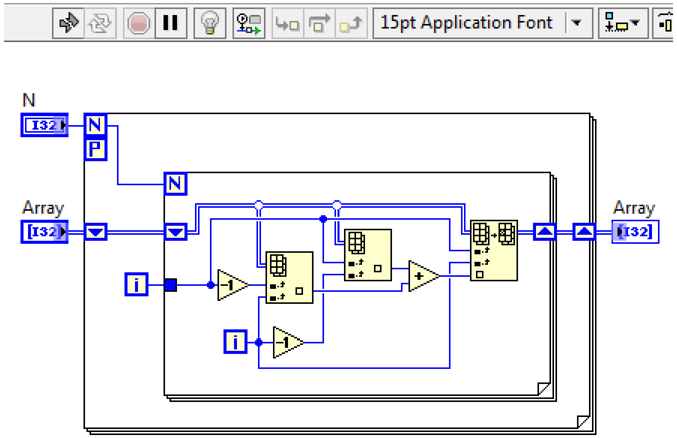
Parallel For Loop Limitations



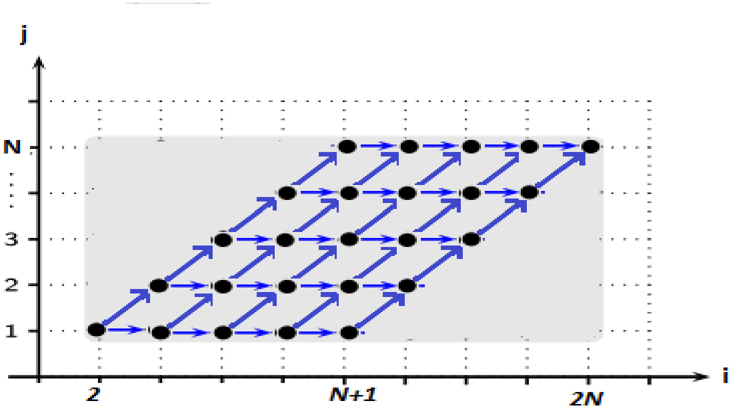
```
for(i=1;i<=N;i++)
    for(j=1;j<=N;j++){
        // neither of the two loops are parallel
        a[i][j] = a[i][j-1] + a[i-1][j]
    }
```

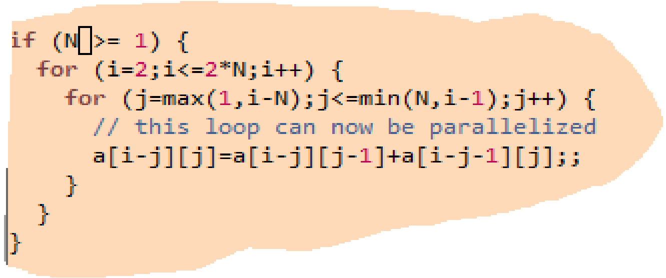
None of these loops can be parallelized





Loop skewing exposes the hidden parallelism



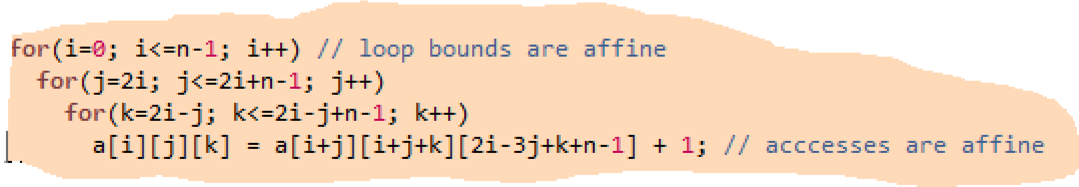


```
if (N>= 1) {
  for (i=2;i<=2*N;i++) {
    for (j=max(1,i-N);j<=min(N,i-1);j++) {
      // this loop can now be parallelized
      a[i-j][j]=a[i-j][j-1]+a[i-j-1][j];;
    }
  }
}
```

Loop-nest is inner parallel

Polyhedral Model - A Short Overview

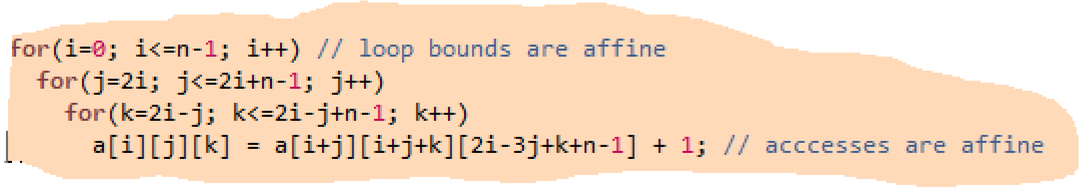
- Abstract mathematical representation
    - Convenient to reason about complex program transformations
- Static Control Parts (SCoP), typically affine loop-nests
    - e.g. stencil computations, linear algebra kernels



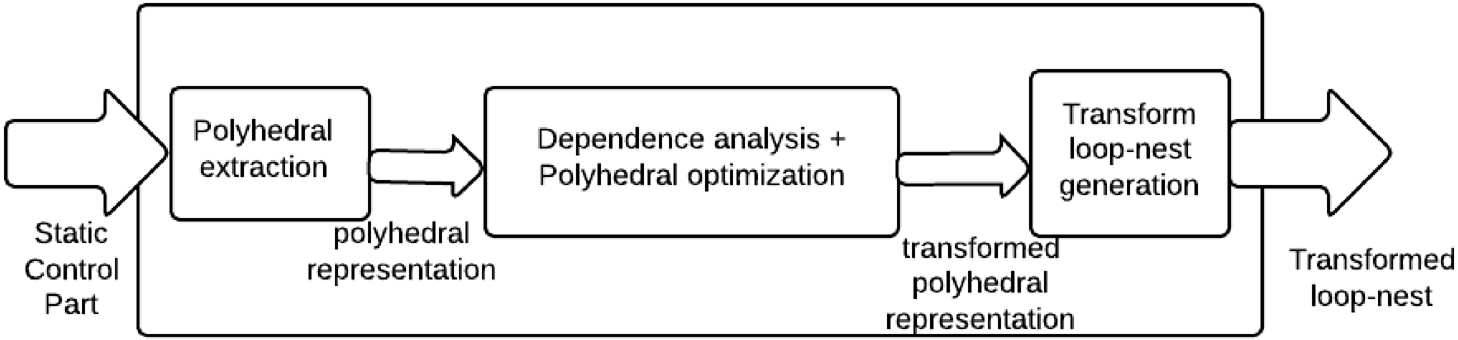
```
for(i=0; i<=n-1; i++) // loop bounds are affine
  for(j=2i; j<=2i+n-1; j++)
    for(k=2i-j; k<=2i-j+n-1; k++)
      a[i][j][k] = a[i+j][i+j+k][2i-3j+k+n-1] + 1; // acccesses are affine
```

Polyhedral Model - A Short Overview

- Abstract mathematical representation
  - Convenient to reason about complex program transformations
- Static Control Parts (SCoP), typically affine loop-nests
  - e.g. stencil computations, linear algebra kernels



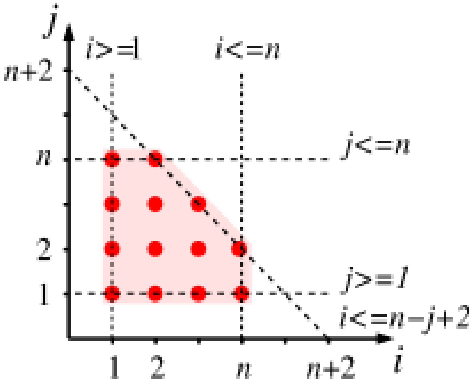
```
for(i=0; i<=n-1; i++) // loop bounds are affine
  for(j=2i; j<=2i+n-1; j++)
    for(k=2i-j; k<=2i-j+n-1; k++)
      a[i][j][k] = a[i+j][i+j+k][2i-3j+k+n-1] + 1; // acccesses are affine
```


Static Control Part
Polyhedral extraction
polyhedral representation
Dependence analysis + Polyhedral optimization
transformed polyhedral representation
Transform loop-nest generation
Transformed loop-nest

Polyhedral Model - A Short Overview

- Dynamic instances of a statement
  - Integer points inside a polyhedron
  - Iteration domain as conjunction of affine inequalities involving surrounding loop iterators and global parameters

```
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
if(i<=n-j+2)
S1;
```



Iteration domain of S1

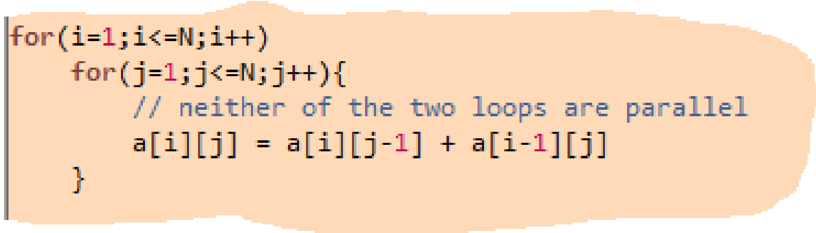
$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 1 \\ -n \\ 1 \\ -n \\ -n-2 \end{pmatrix}$$

Iteration domain of S1

Figure. Polyhedral representation of a loop-nest in geometrical and linear algebraic form

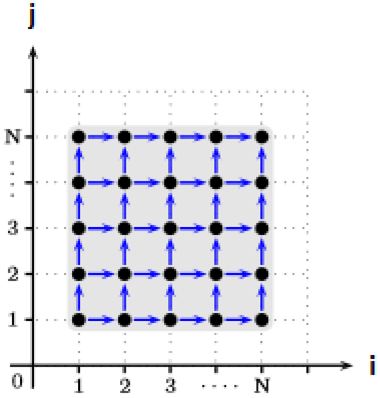
Polyhedral model - a brief overview

- A multi-dimensional affine schedule
  - Specifies order in which the integer points need to be scanned
  - Maps each integer point to multi-dimensional logical timestamp (think…hours, minutes, seconds)



```
for(i=1;i<=N;i++)
    for(j=1;j<=N;j++){
        // neither of the two loops are parallel
        a[i][j] = a[i][j-1] + a[i-1][j]
    }
```

Schedule of the statement instances is given by ***theta(i, j) = (i, j)***

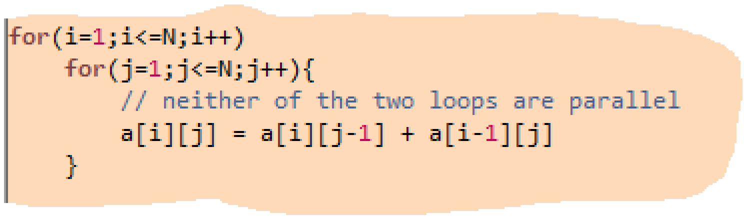


Polyhedral model - a brief overview

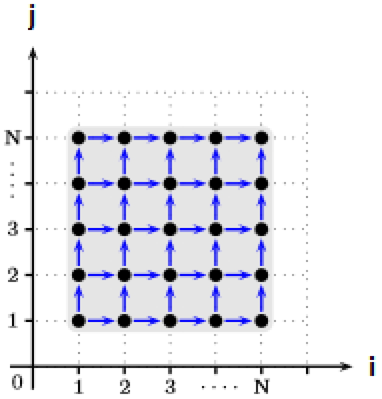
- Array access information also encoded, must be affine
- Polyhedral optimizer/parallelizer
  - Analyzes the dependences
  - Pick schedule without violating dependences using a cost model
  - PLuTo: minimize dependence distances in transformed space
    - Optimizes parallelism and locality simultaneously

Polyhedral model - a brief overview

- Array access information also encoded, must be affine
- Polyhedral optimizer/parallelizer
  - Analyzes the dependences
  - Pick schedule without violating dependences using a cost model
  - PLuTo: minimize dependence distances in transformed space
    - Optimizes parallelism and locality simultaneously



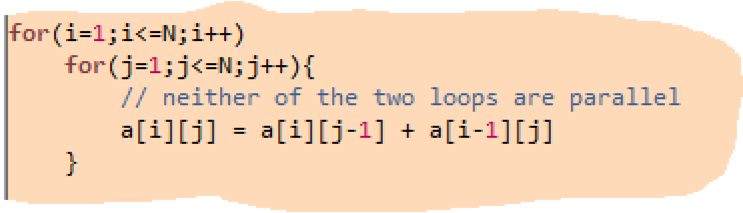
```
for(i=1;i<=N;i++)
    for(j=1;j<=N;j++){
        // neither of the two loops are parallel
        a[i][j] = a[i][j-1] + a[i-1][j]
    }
```



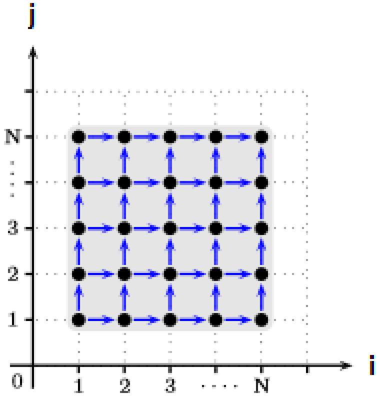
Schedule of the statement instances is given by ***theta(i, j) = (i, j)***

Polyhedral model - a brief overview

- Array access information also encoded, must be affine
- Polyhedral optimizer/parallelizer
  - Analyzes the dependences
  - Pick schedule without violating dependences using a cost model
  - PLuTo: minimize dependence distances in transformed space
    - Optimizes parallelism and locality simultaneously

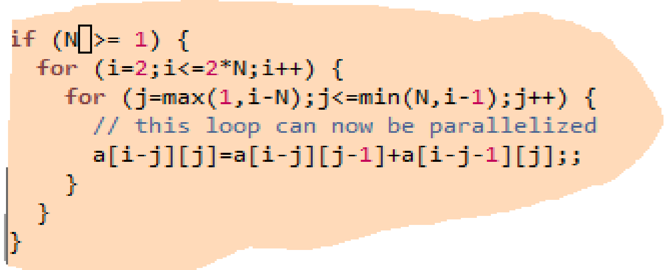


```
for(i=1;i<=N;i++)
    for(j=1;j<=N;j++){
        // neither of the two loops are parallel
        a[i][j] = a[i][j-1] + a[i-1][j]
    }
```

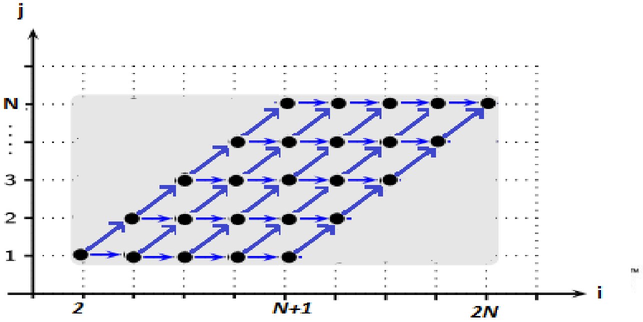


Schedule of the statement instances is given by
theta(i, j) = (i, j)

New schedule is
theta(i, j) = (i+j, j)



```
if (N□>= 1) {
  for (i=2;i<=2*N;i++) {
    for (j=max(1,i-N);j<=min(N,i-1);j++) {
      // this loop can now be parallelized
      a[i-j][j]=a[i-j][j-1]+a[i-j-1][j];;
    }
  }
}
```



Polyhedral compilation - some related work

Polyhedral compilation of imperative programs

- Extract polyhedral representation e.g. Clan (Cedric Bastoul et al)
- Polyhedral transformation - PLuTo (Uday Bondhugula et al)
- Generated transformed code e.g. CLooG (Cedric Bastoul et al)
- Polyhedral compilation in production compilers e.g. IBM-XL, RSTREAM

Polyhedral compilation of graphical dataflow programs?

- Polyhedral extraction from dataflow programs
- Synthesizing dataflow programs from polyhedral representation

Extracting Polyhedral Representation

- Identifying statement analogues
- Relating array accesses to a particular array allocation
- Execution schedule depends on the actual inplaceness strategy

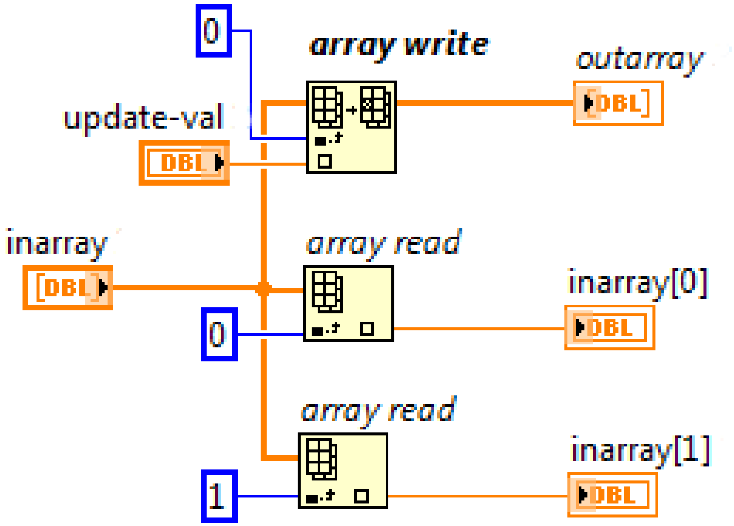
Static Control Dataflow Diagram (SCoD)

- Canonical form of dataflow program
- Inplaceness patterns that facilitate polyhedral extraction
  - no new memory allocation for array data inside the SCoD

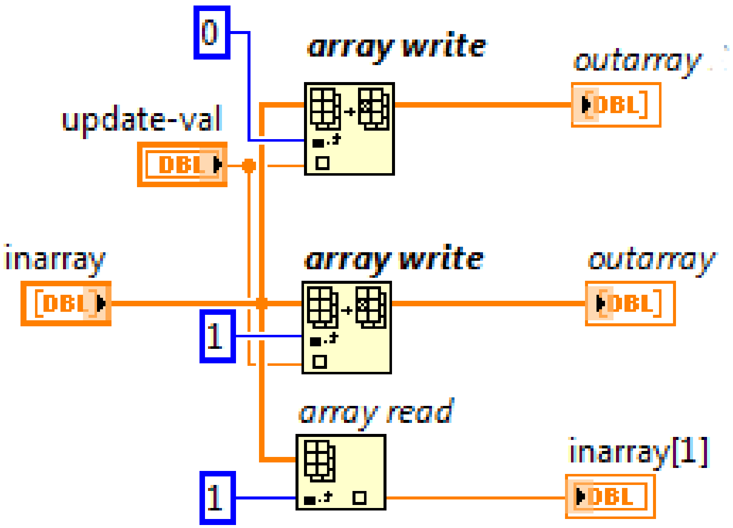
- Similarities with SCoP
  - All computations nodes are functional
  - Maximal dataflow diagram with countable loop constructs
  - Loop bounds and conditional depend on parameters that are invariant for the diagram

SCoD – Destructive Updates

- At most one destructive update of array data



Atmost one destructive update of the incoming array? *Yes*



Atmost one destructive update of the incoming array? ***No***

Compute-dags as Statement Analogues

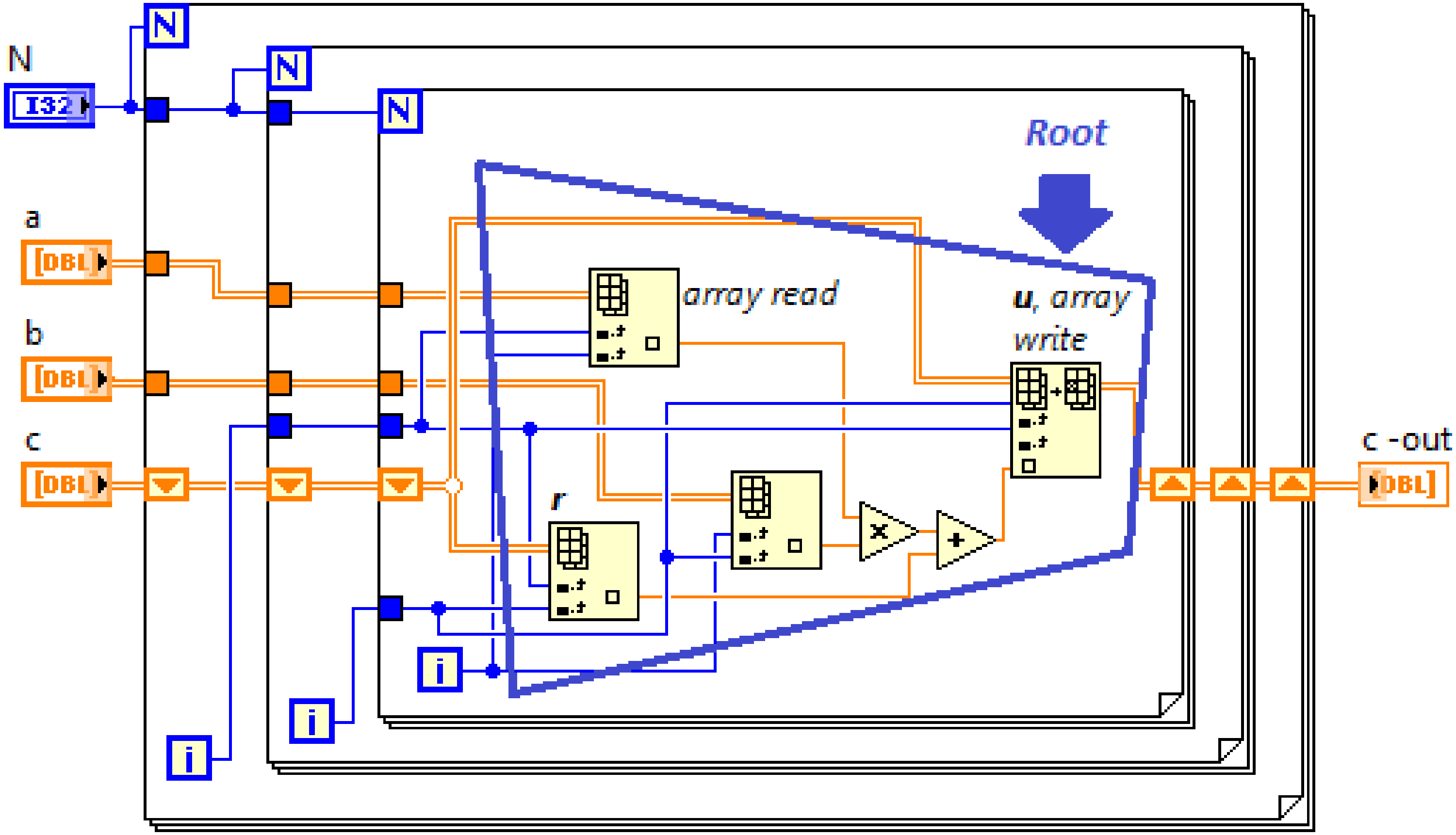
- Schedule of nodes exists such that no array copy is needed
  - hint: schedule all array reads ahead of the array write

- SCoD as sequence of computations that over-write incoming array data

- Compute-dags can be identified to serve as statement analogues

Compute-dags as Statement Analogues

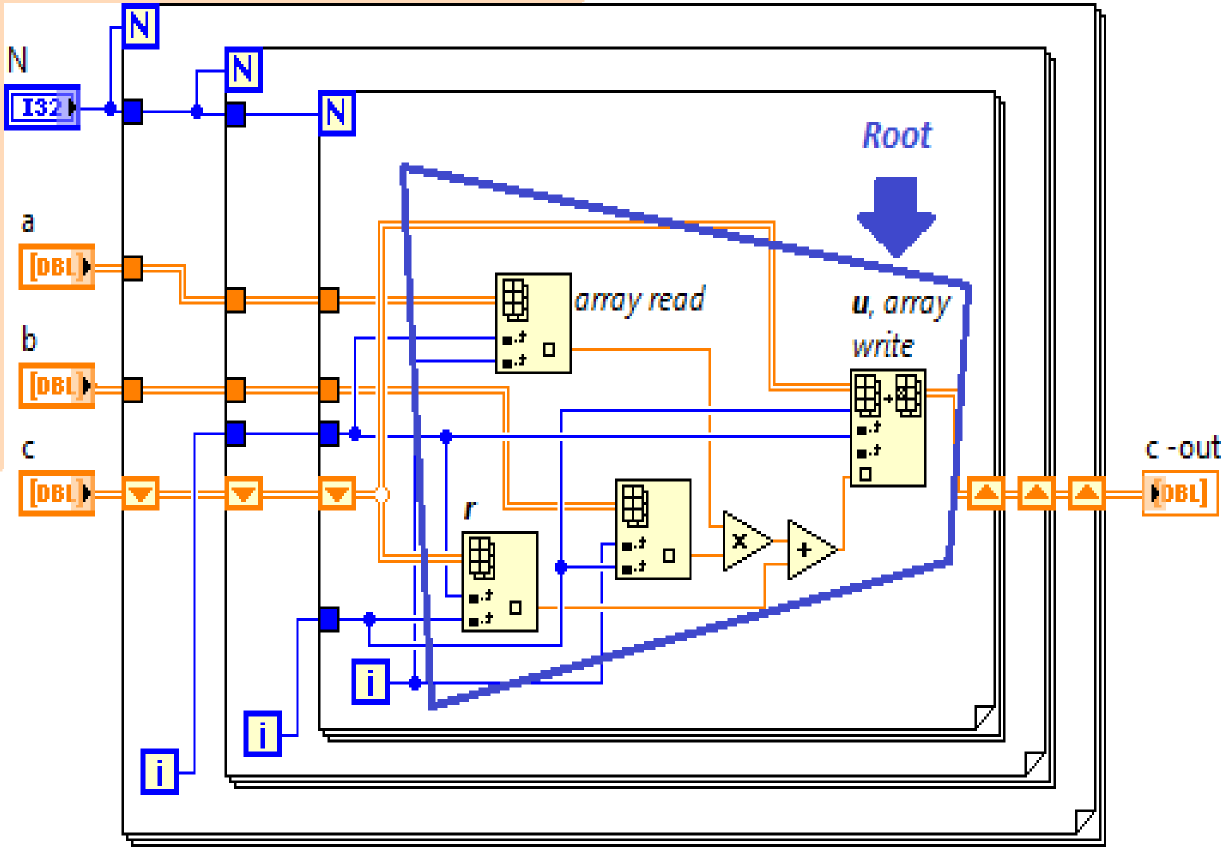
- A path exists from all nodes in the compute-dag to the root



Iteration Domain of Statement Analogues

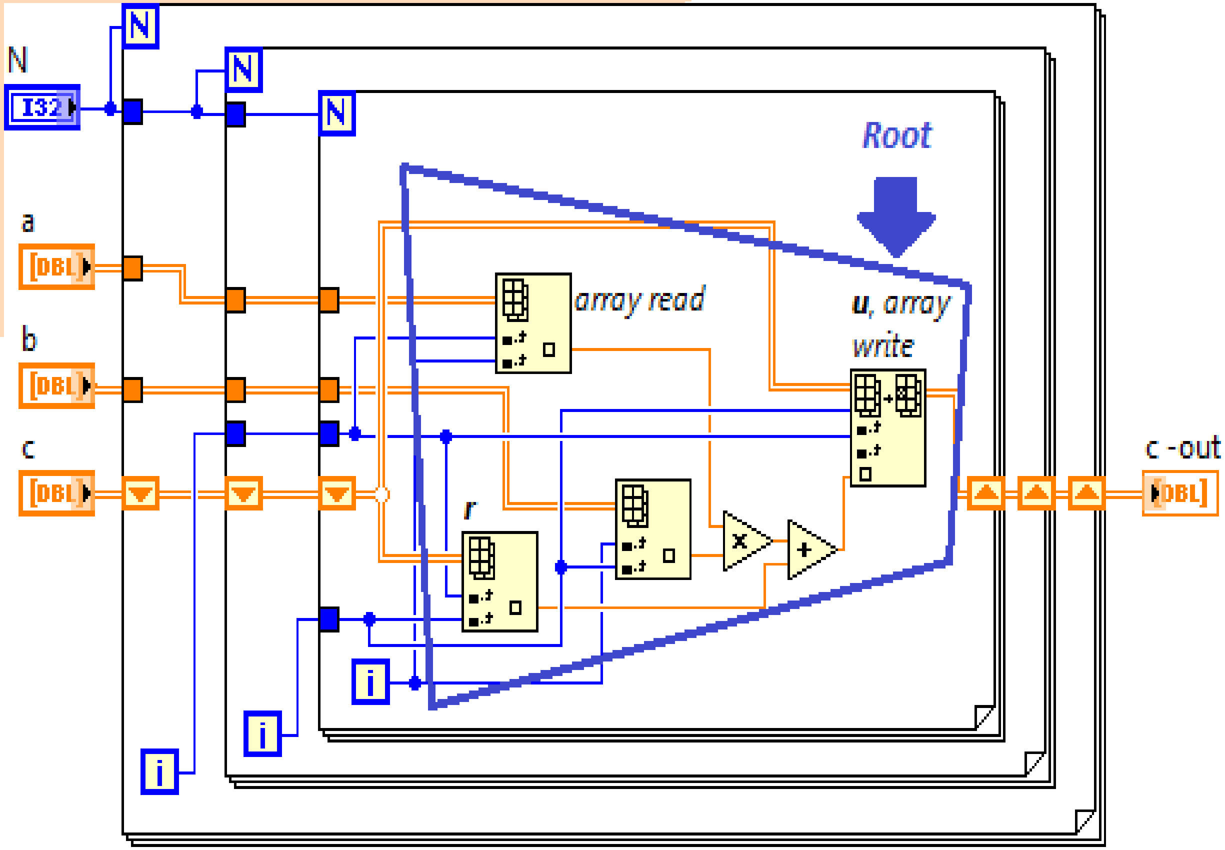
```
# Number of statements
1

# =================================================== Statement 0
# --------------------------------------------------- Domain
# Iterator Domain is provided
1
# Iterator Domain
# In matrix form
# There are 3 surrounding loops
#  -t2 + N0 -1 >= 0
#  t2 >= 0
#  -t1 + N0 -1 >= 0
#  t1 >= 0
#  -t0 + N0 -1 >= 0
#  t0 >= 0
6 6
1 1 0 0 0 0
1 -1 0 0 1 -1
1 0 1 0 0 0
1 0 -1 0 1 -1
1 0 0 1 0 0
1 0 0 -1 1 -1
```



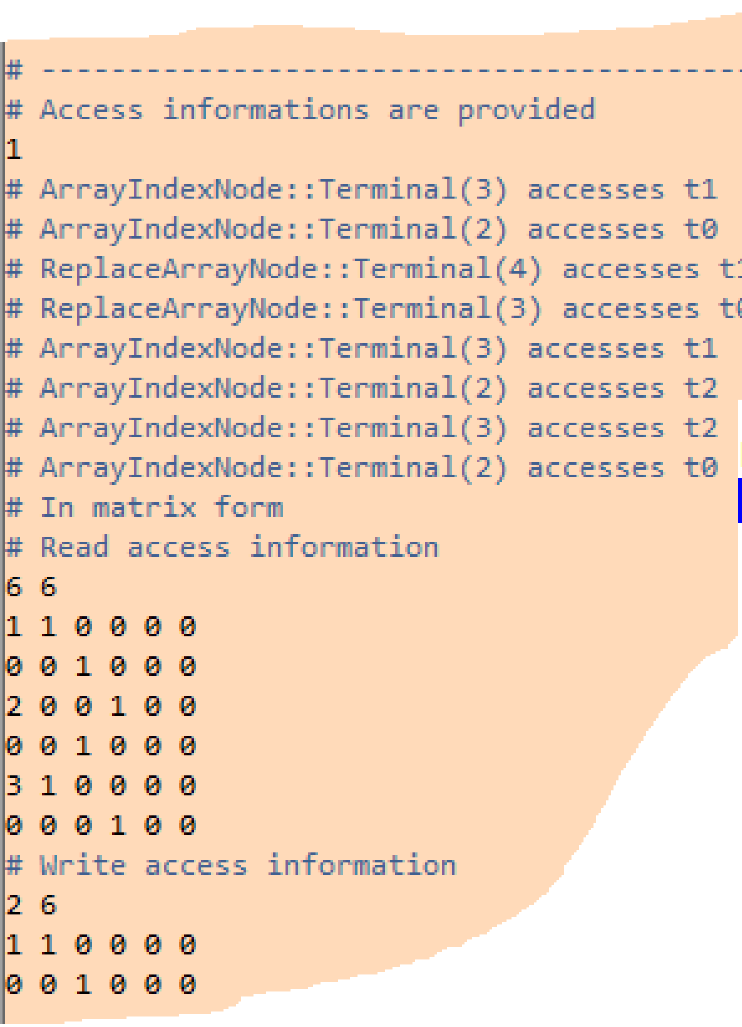
Determining Schedule of Statement Analogues

```
# ------------------------------------------ Scattering
# Scattering function is provided
1
# Scattering function
# ReplaceArrayNode is scheduled at 0, t0, 0, t1, 0, t2, 0,
# In matrix form
7 6
0 0 0 0 0 0
0 1 0 0 0 0
0 0 0 0 0 0
0 0 1 0 0 0
0 0 0 0 0 0
0 0 0 1 0 0
0 0 0 0 0 0
```

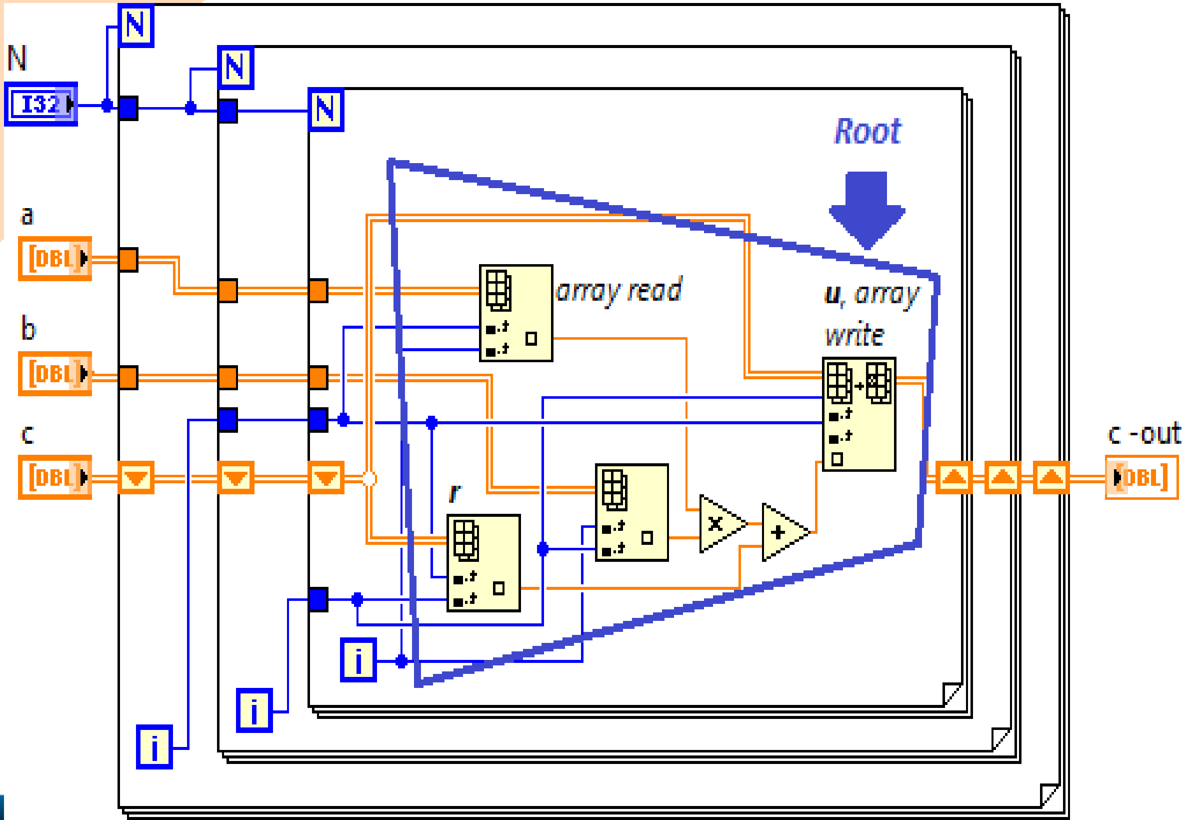


ni.com

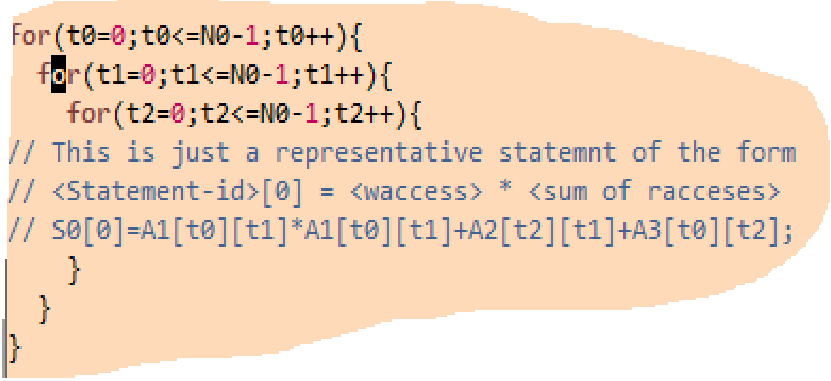
Analyzing Accesses of Statement Analogues



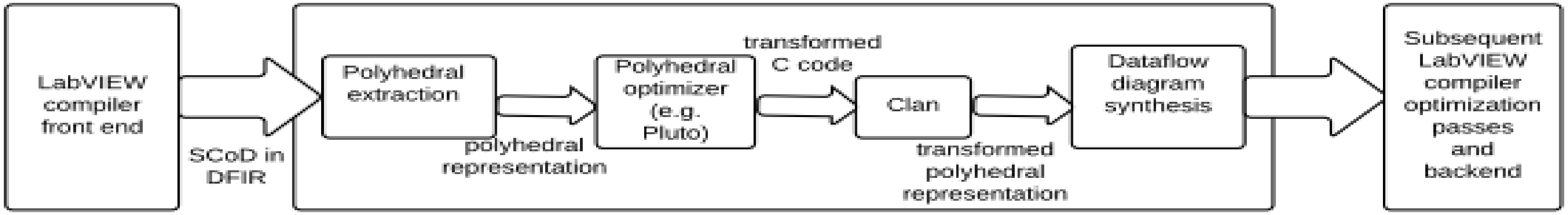
```
# ---------------------------------------------------- Access
# Access informations are provided
1
# ArrayIndexNode::Terminal(3) accesses t1
# ArrayIndexNode::Terminal(2) accesses t0
# ReplaceArrayNode::Terminal(4) accesses t1
# ReplaceArrayNode::Terminal(3) accesses t0
# ArrayIndexNode::Terminal(3) accesses t1
# ArrayIndexNode::Terminal(2) accesses t2
# ArrayIndexNode::Terminal(3) accesses t2
# ArrayIndexNode::Terminal(2) accesses t0
# In matrix form
# Read access information
6 6
1 1 0 0 0 0
0 0 1 0 0 0
2 0 0 1 0 0
0 0 1 0 0 0
3 1 0 0 0 0
0 0 0 1 0 0
# Write access information
2 6
1 1 0 0 0 0
0 0 1 0 0 0
```



The PolyGLoT framework

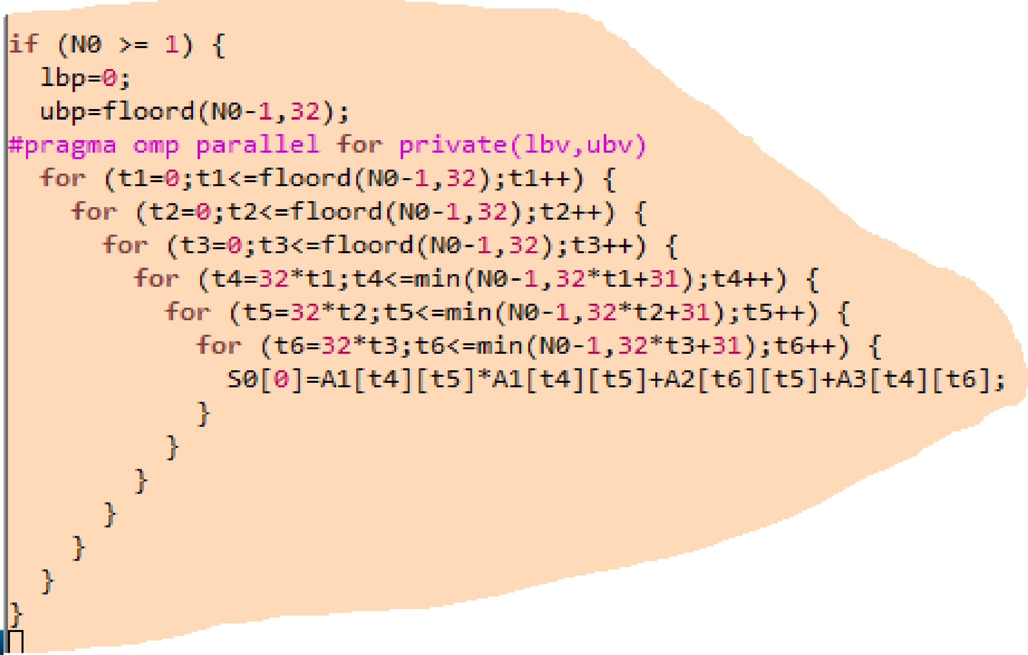


```
for(t0=0;t0<=N0-1;t0++){
  for(t1=0;t1<=N0-1;t1++){
    for(t2=0;t2<=N0-1;t2++){
// This is just a representative statemnt of the form
// <Statement-id>[0] = <waccess> * <sum of racceses>
// S0[0]=A1[t0][t1]*A1[t0][t1]+A2[t2][t1]+A3[t0][t2];
    }
  }
}
```



LabVIEW compiler front end
SCoD in DFIR
Polyhedral extraction
polyhedral representation
Polyhedral optimizer (e.g. Pluto)
transformed C code
Clan
transformed polyhedral representation
Dataflow diagram synthesis
Subsequent LabVIEW compiler optimization passes and backend

A high-level overview of PolyGLoT



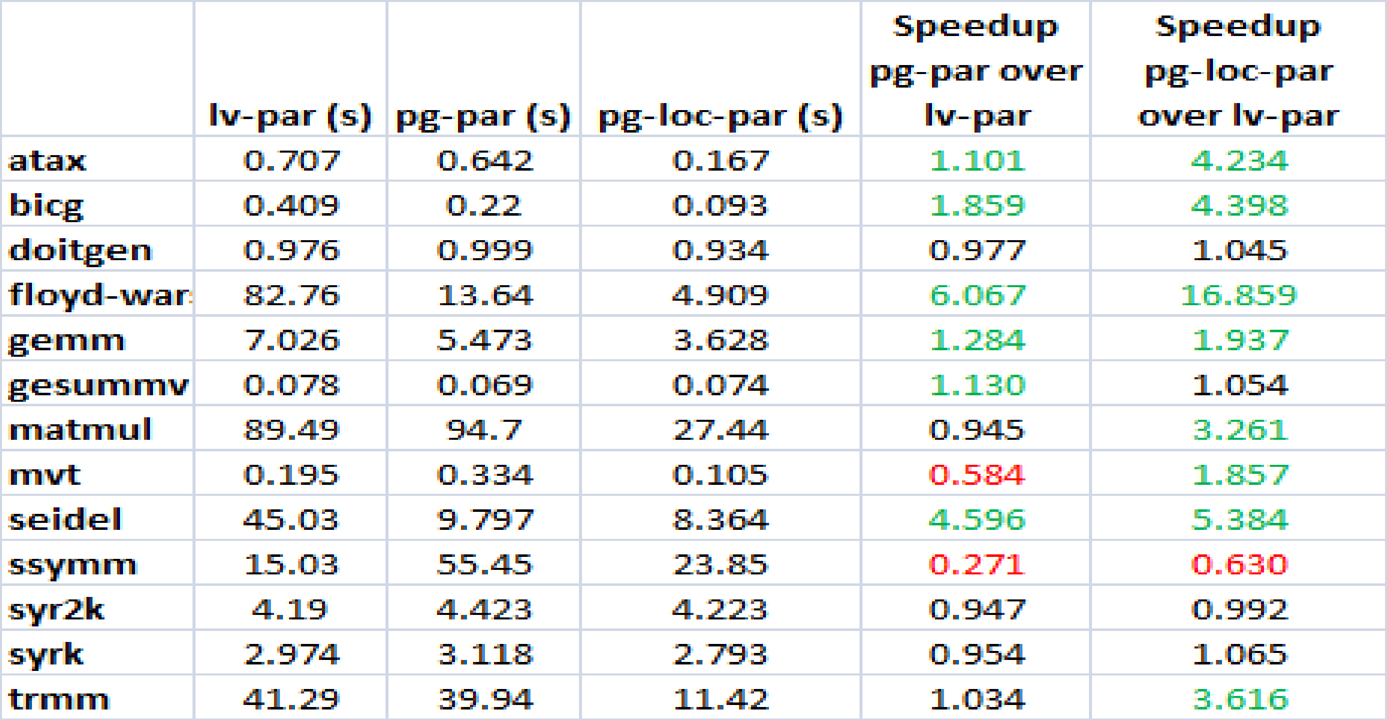
```
if (N0 >= 1) {
  lbp=0;
  ubp=floord(N0-1,32);
#pragma omp parallel for private(lbv,ubv)
  for (t1=0;t1<=floord(N0-1,32);t1++) {
    for (t2=0;t2<=floord(N0-1,32);t2++) {
      for (t3=0;t3<=floord(N0-1,32);t3++) {
        for (t4=32*t1;t4<=min(N0-1,32*t1+31);t4++) {
          for (t5=32*t2;t5<=min(N0-1,32*t2+31);t5++) {
            for (t6=32*t3;t6<=min(N0-1,32*t3+31);t6++) {
              S0[0]=A1[t4][t5]*A1[t4][t5]+A2[t6][t5]+A3[t4][t6];
            }
          }
        }
      }
    }
  }
}
```

Experimental evaluation

- Implemented benchmarks in Polybench suite in LabVIEW
- PolyGLoT as a separate transform pass in LV desktop compiler
  - uses Pluto as the polyhedral optimizer (locality transformations + parallelization)
- Dual-socket Intel(R) Xeon(R) CPU E5606 (2.13GHz) machine with 8 cores, 24GB RAM, 8MB L3 cache

Experimental evaluation

- **lv-parallel** - LabVIEW production compiler, with parallelization
- **pg-par** - LabVIEW compiler with PolyGLoT enabled for auto-parallelization
- **pg-loc-par** - LabVIEW compiler with PolyGLoT enabled for auto-parallelization + locality optimization
- mean speed-up of ***2.30×*** with **pg-loc-par** over **lv-parallel**



	lv-par (s)	pg-par (s)	pg-loc-par (s)	Speedup pg-par over lv-par	Speedup pg-loc-par over lv-par
atax	0.707	0.642	0.167	1.101	4.234
bicg	0.409	0.22	0.093	1.859	4.398
doitgen	0.976	0.999	0.934	0.977	1.045
floyd-war	82.76	13.64	4.909	6.067	16.859
gemm	7.026	5.473	3.628	1.284	1.937
gesummv	0.078	0.069	0.074	1.130	1.054
matmul	89.49	94.7	27.44	0.945	3.261
mvt	0.195	0.334	0.105	0.584	1.857
seidel	45.03	9.797	8.364	4.596	5.384
ssymm	15.03	55.45	23.85	0.271	0.630
syr2k	4.19	4.423	4.223	0.947	0.992
syrk	2.974	3.118	2.793	0.954	1.065
trmm	41.29	39.94	11.42	1.034	3.616

Summary

- Graphical dataflow programming
  - Simple, intuitive and accessible to novice programmers
  - Well-suited for exploiting and expressing parallelism
  - Used by scientists and engineers in various domains

- Optimizing and parallelizing LabVIEW compiler
  - Clumps of independently schedulable sections of code
  - Task parallelism, data parallelism, pipelining etc

- Parallel for loop for cross-iteration parallelism

- Polyhedral model for complex program transformations

Thanks!

Questions?