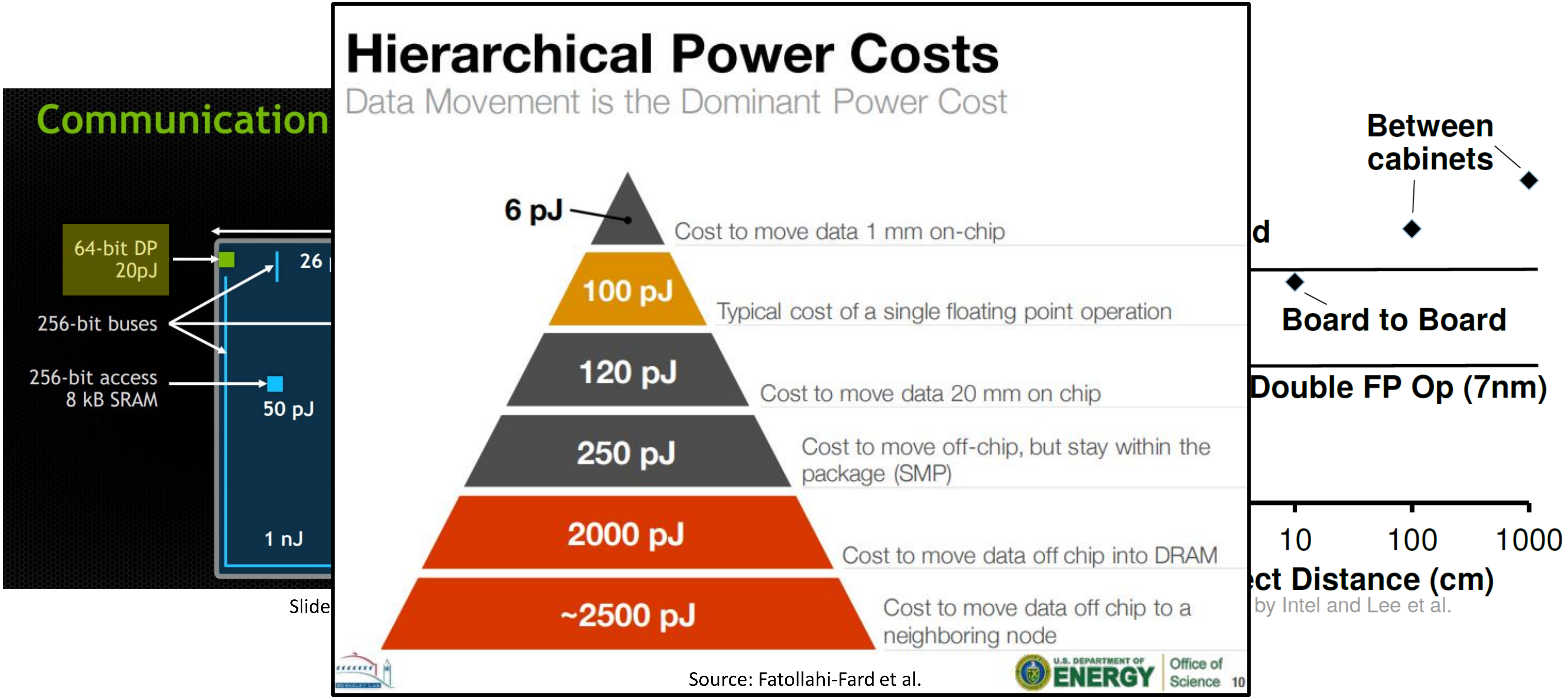
Alexandru Calotoiu, Tal Ben-Nun, Grzegorz Kwasniewski, Johannes de Fine Licht, Timo Schneider, Philipp Schaad, Torsten Hoefler

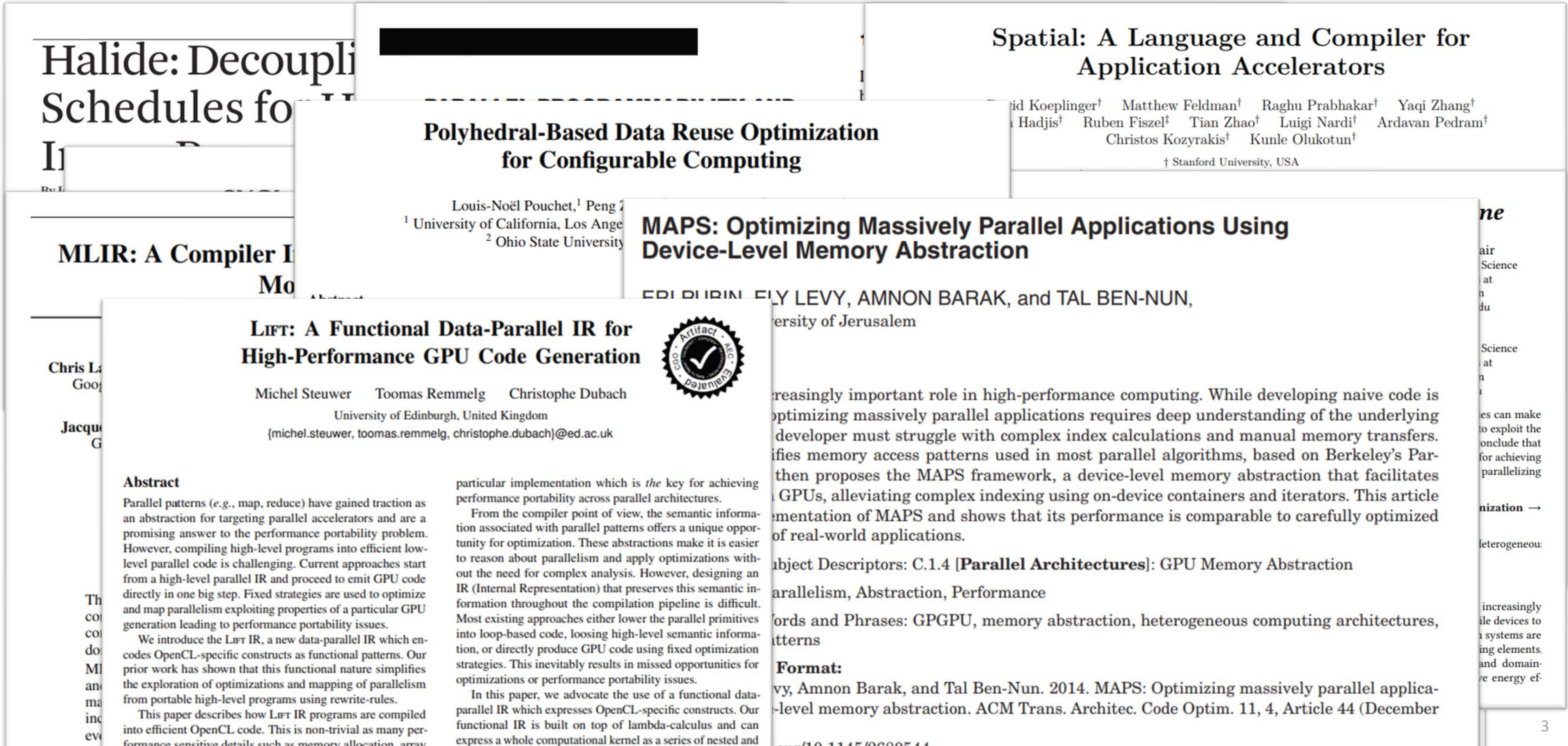
Lifting C Semantics for Dataflow Optimization

Data movement dominates energy costs!



Source: Fatollahi-Fard et al.

Efficient data movement is hard!



Efficient data movement is hard!

Multi-target IR: LLVM, HPVM

Polyhedral compilers: Pluto, Polly

General compilers: GCC, Clang, ICC

Efficient data movement is hard!

Indirect accesses

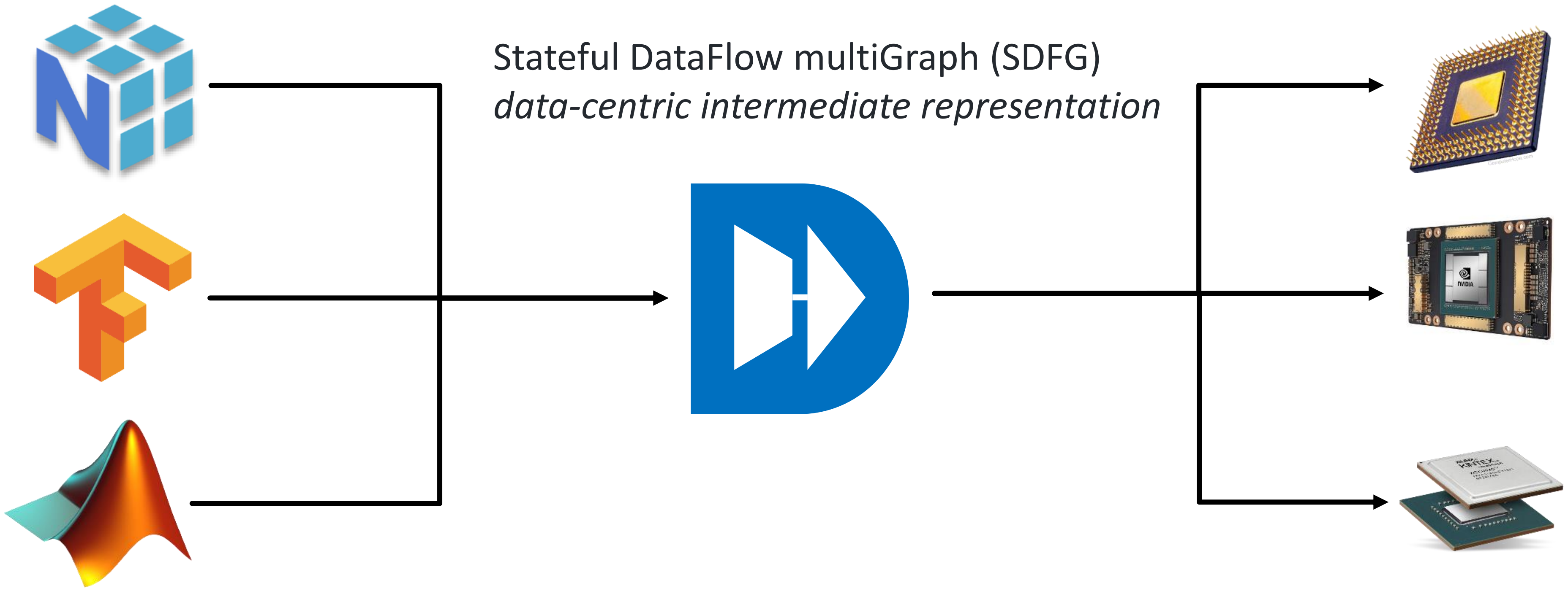
Pointers and aliasing

Polyhedral compilers: Pluto, Polly

Difficult & expensive whole program analysis

General compilers: GCC, Clang, ICC

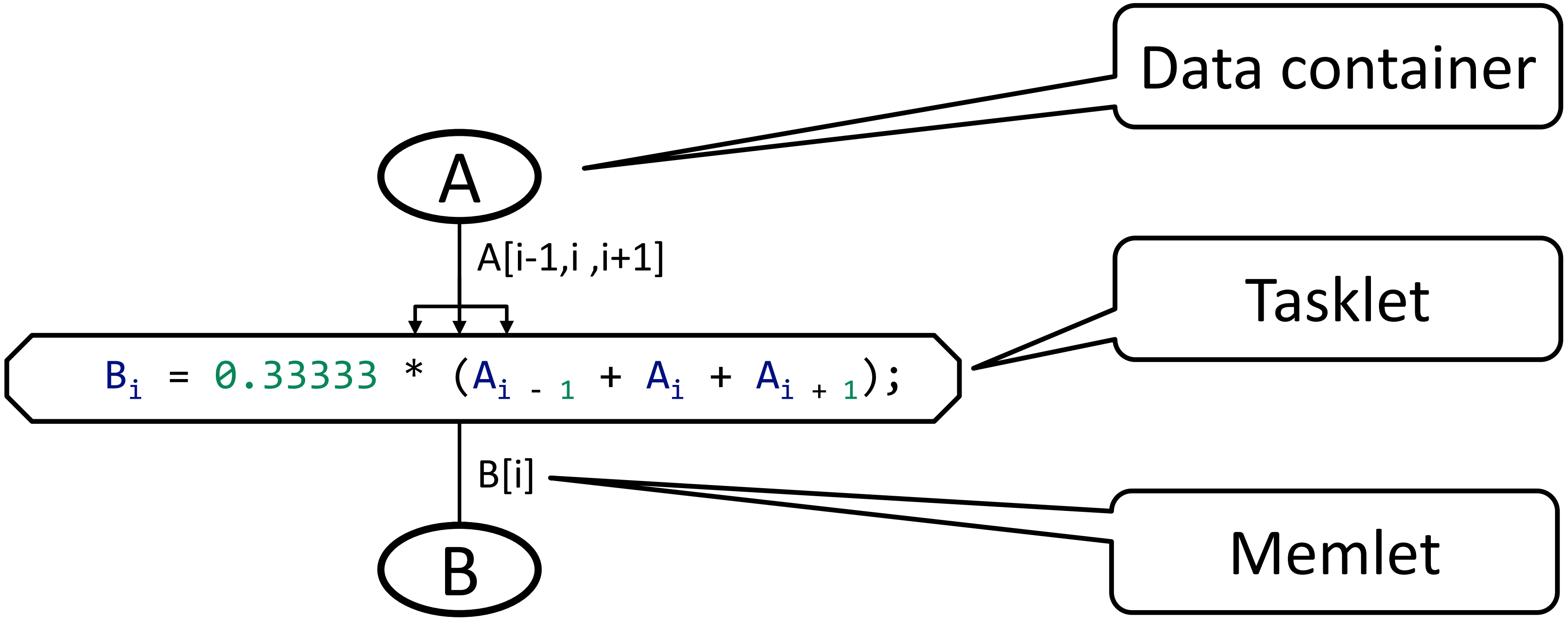
Data Centric Parallel Programming



What is the SDFG IR?

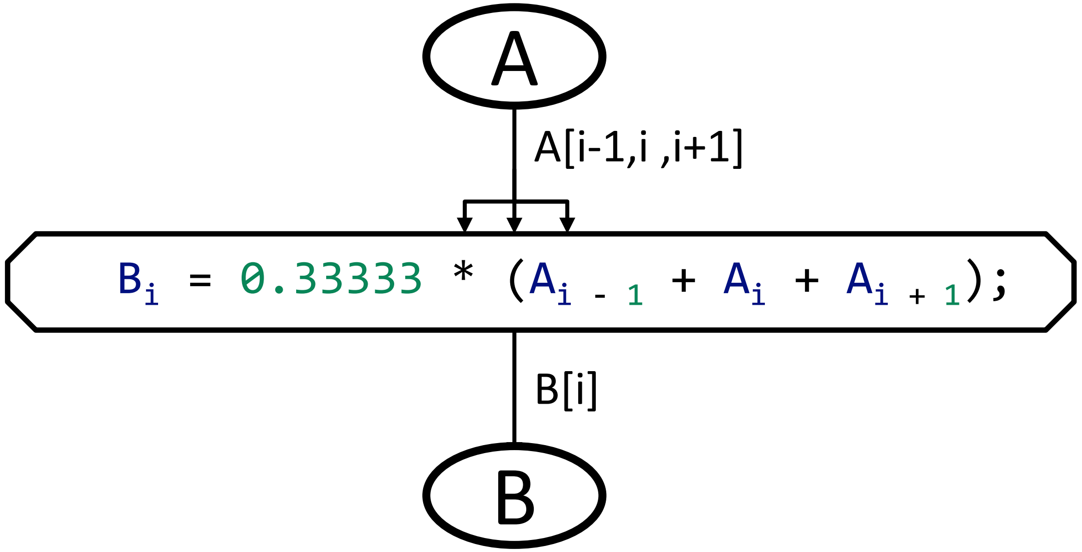
```
B[i] = 0.33333 * (A[i - 1] + A[i] + A[i + 1]);
```

What is the SDFG IR?



What is the SDFG IR?

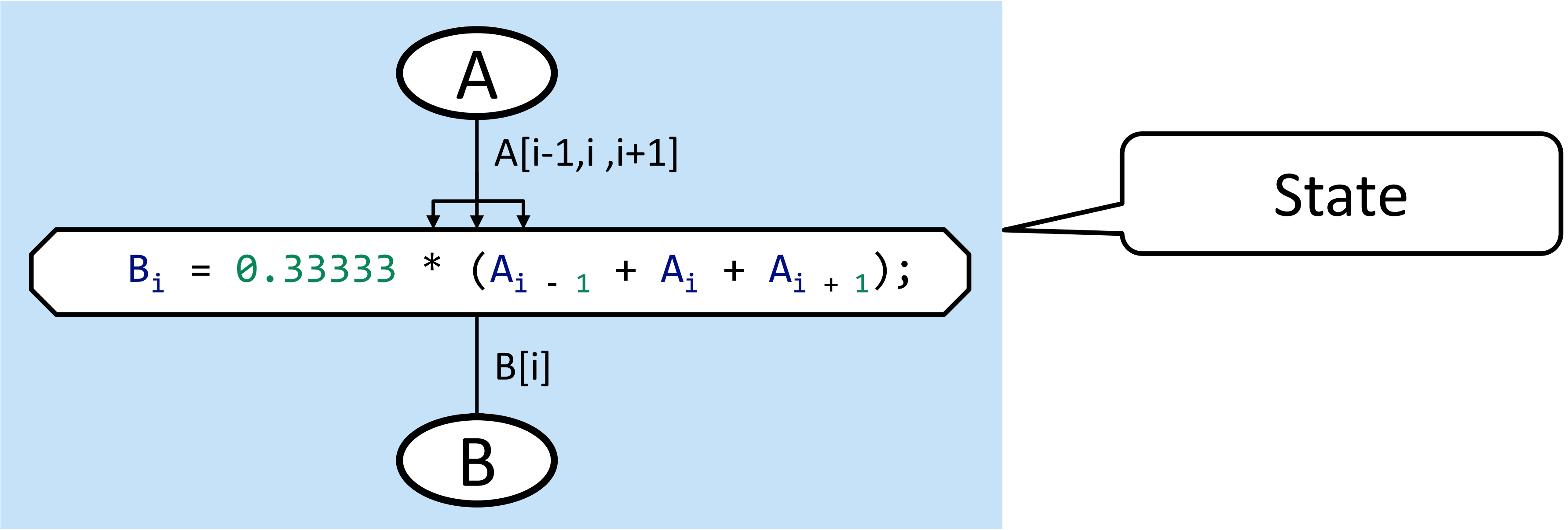
```
for (i = 1; i < n - 1; i++)
{
```




```
}
```

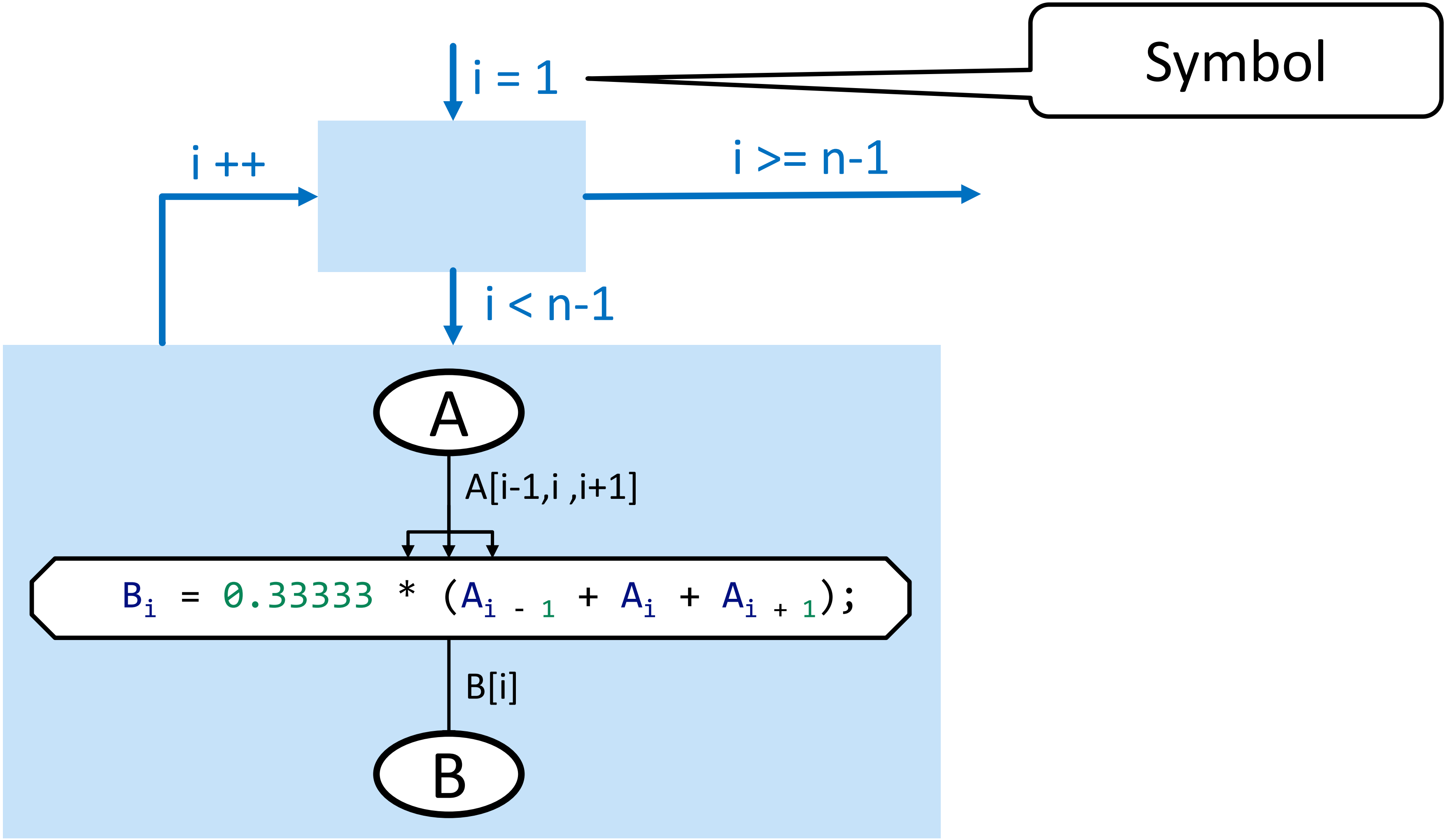
What is the SDFG IR?

```
for (i = 1; i < n - 1; i++)
{
```

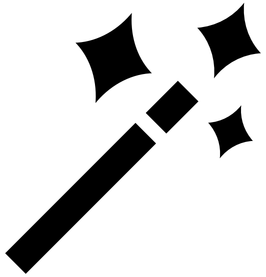



```
}
```

What is the SDFG IR?



Why translate C to DaCe?



- Generate GPU code
- Generate FPGA code
- Improve data movement
  - Apply tiling
  - Reorder loops

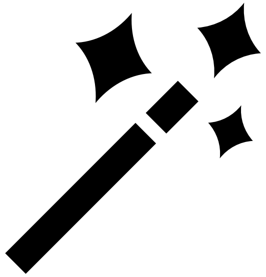
```c
static void kernel_jacobi_1d(int tsteps,
                             int n,
                             double A[2000],
                             double B[2000])
{
  int t, i;

  for (t = 0; t < tsteps; t++)
  {

    for (i = 1; i < n - 1; i++)
    {
      B[i] = 0.33333 * (A[i - 1] + A[i] + A[i + 1]);
    }

    for (i = 1; i < n - 1; i++)
    {
      A[i] = 0.33333 * (B[i - 1] + B[i] + B[i + 1]);
    }
  }
}
```

Why translate C to DaCe?



- Autoparallelization
- Generate GPU code ✓
- Generate FPGA code ✓
- Improve data movement ✓
  - Apply tiling ✓
  - Reorder loops ✓

```c
static void kernel_jacobi_1d(int tsteps,
                             int n,
                             double A[2000],
                             double B[2000])
{
  int t, i;

  for (t = 0; t < tsteps; t++)
  {
    #pragma omp parallel for
    for (i = 1; i < n - 1; i++)
    {
      B[i] = 0.33333 * (A[i - 1] + A[i] + A[i + 1]);
    }
    #pragma omp parallel for
    for (i = 1; i < n - 1; i++)
    {
      A[i] = 0.33333 * (B[i - 1] + B[i] + B[i + 1]);
    }
  }
}
```

C to DaCe: SpMV

1. AST Transformations

```
for (int i = 0; i < N ; i++)

  for (int j = row_ptr[i]; j < row_ptr[i+1]; j++)


     y[i] +=  A[j] * x[col_idx[j]];
```

C to DaCe: SpMV

1. AST Transformations
   - Make basic blocks explicit

```
for (int i = 0; i < N ; i++){

   for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){


      y[i] +=  A[j] * x[col_idx[j]];
   }
}
```

C to DaCe: SpMV

1. AST Transformations
   - Make basic blocks explicit
   - Extract array indices*
   - And many others…

```
for (int i = 0; i < N ; i++){

    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){

        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
    }
}
```

AST Transformations *canonicalize* the program representation and allows the translation to SDFG to make simplifying assumptions.

*All array indices are extracted, including **i**, **i+1**, **j**. They are omitted here for simplicity and space

C to DaCe: SpMV

1. AST Transformations
   - Make basic blocks explicit
   - Extract array indices*
   - And many others…

```c
for (int i = 0; i < N ; i++){

    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){

         int idx =  col_idx[j];
         y[i] +=  A[j] * x[idx];
    }
}
```

Indirect accesses

AST Transformations *canonicalize* the program representation and allows the translation to SDFG to make simplifying assumptions.

*All array indices are extracted, including **i**, **i+1**, **j**. They are omitted here for simplicity and space

C to DaCe: SpMV

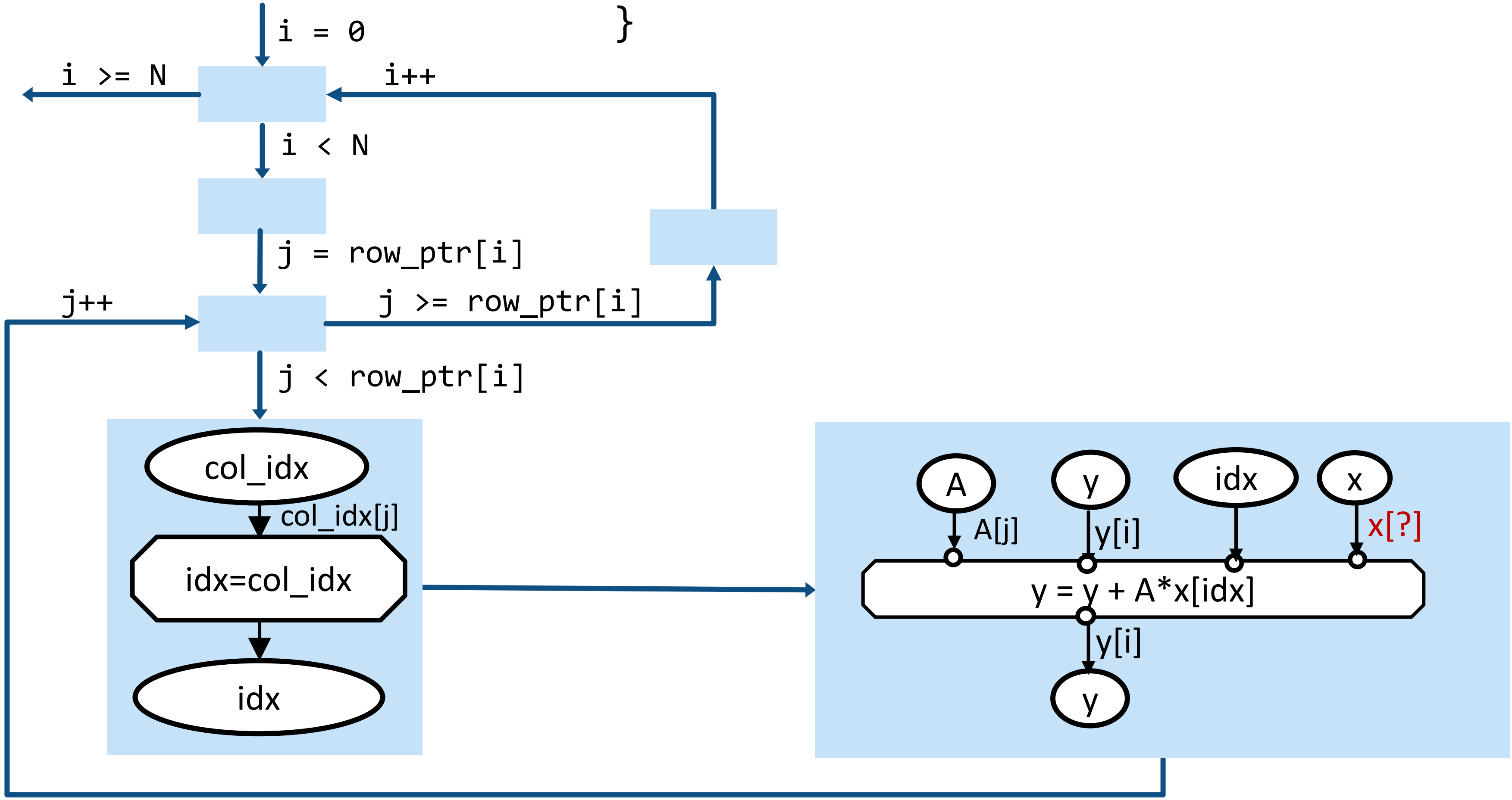
1. AST Transformations
2. Translation from C to SDFG

```
for (int i = 0; i < N ; i++){
   for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
   }
}
```

C to DaCe: SpMV

1. AST Transformations
2. Translation from C to SDFG

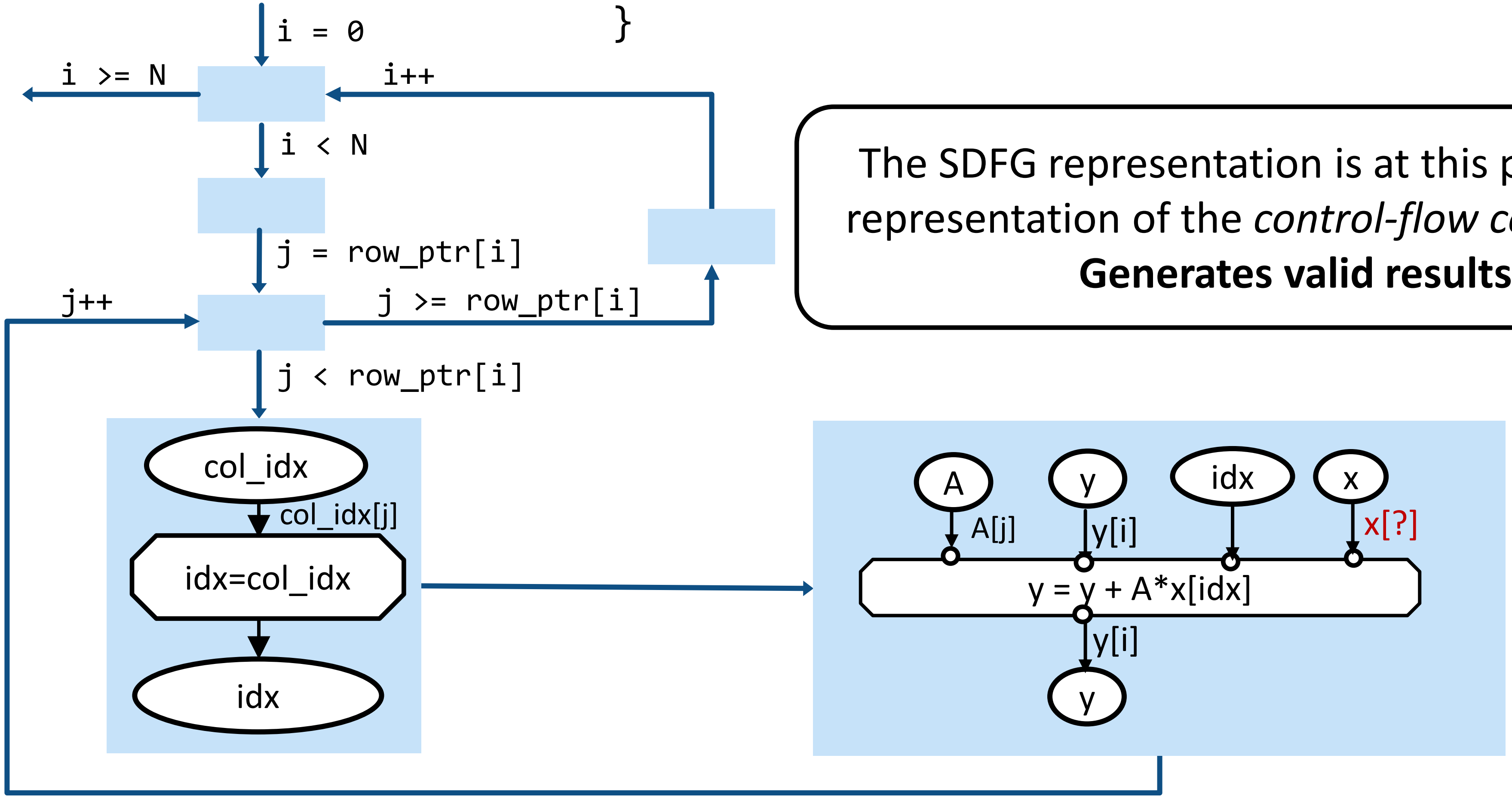
```
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
    }
}
```



C to DaCe: SpMV

```
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
    }
}
```

1. AST Transformations
2. Translation from C to SDFG

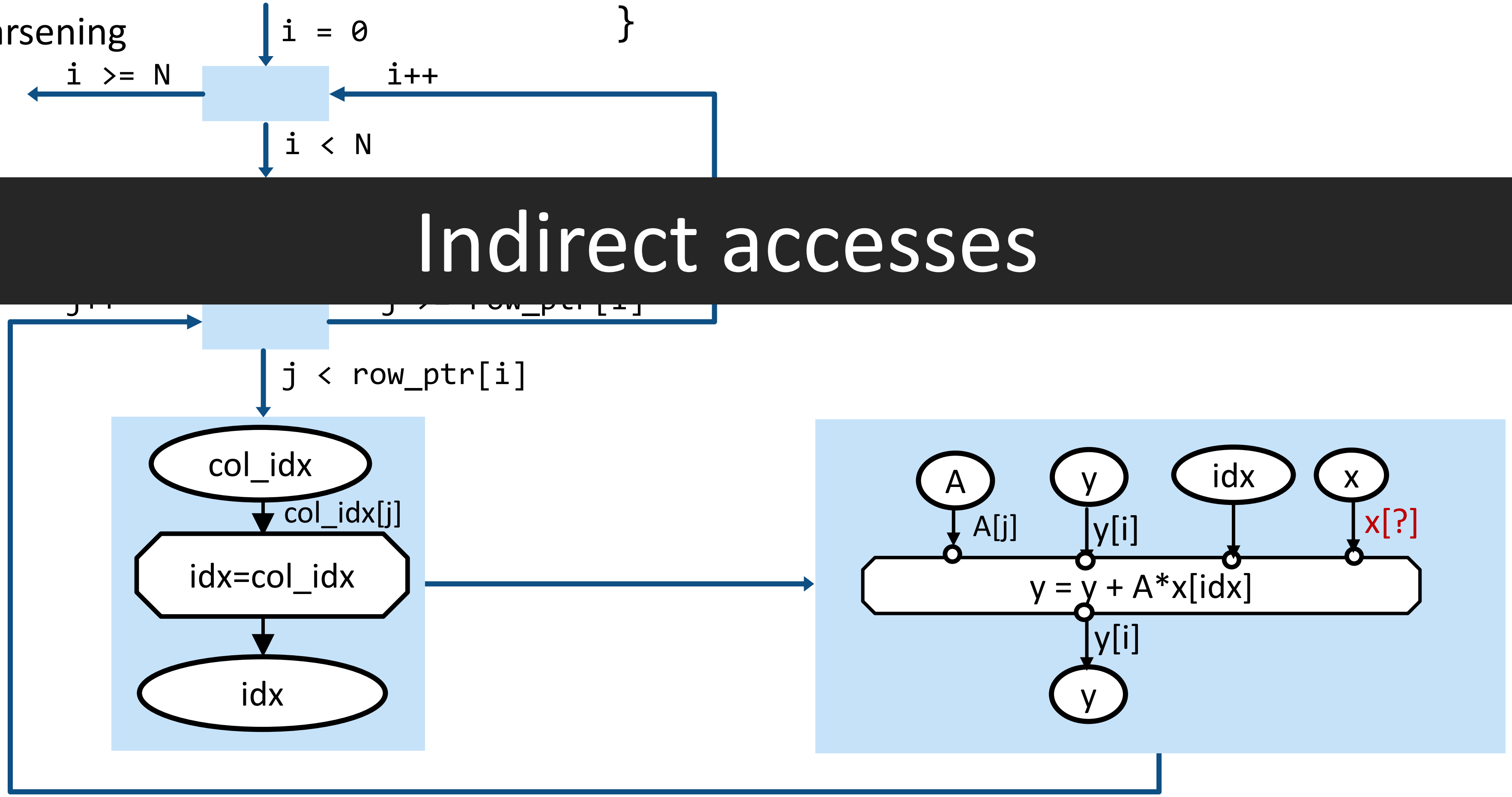


The SDFG representation is at this point a direct representation of the *control-flow centric* C code. **Generates valid results!**

C to DaCe: SpMV

```
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
    }
}
```

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening



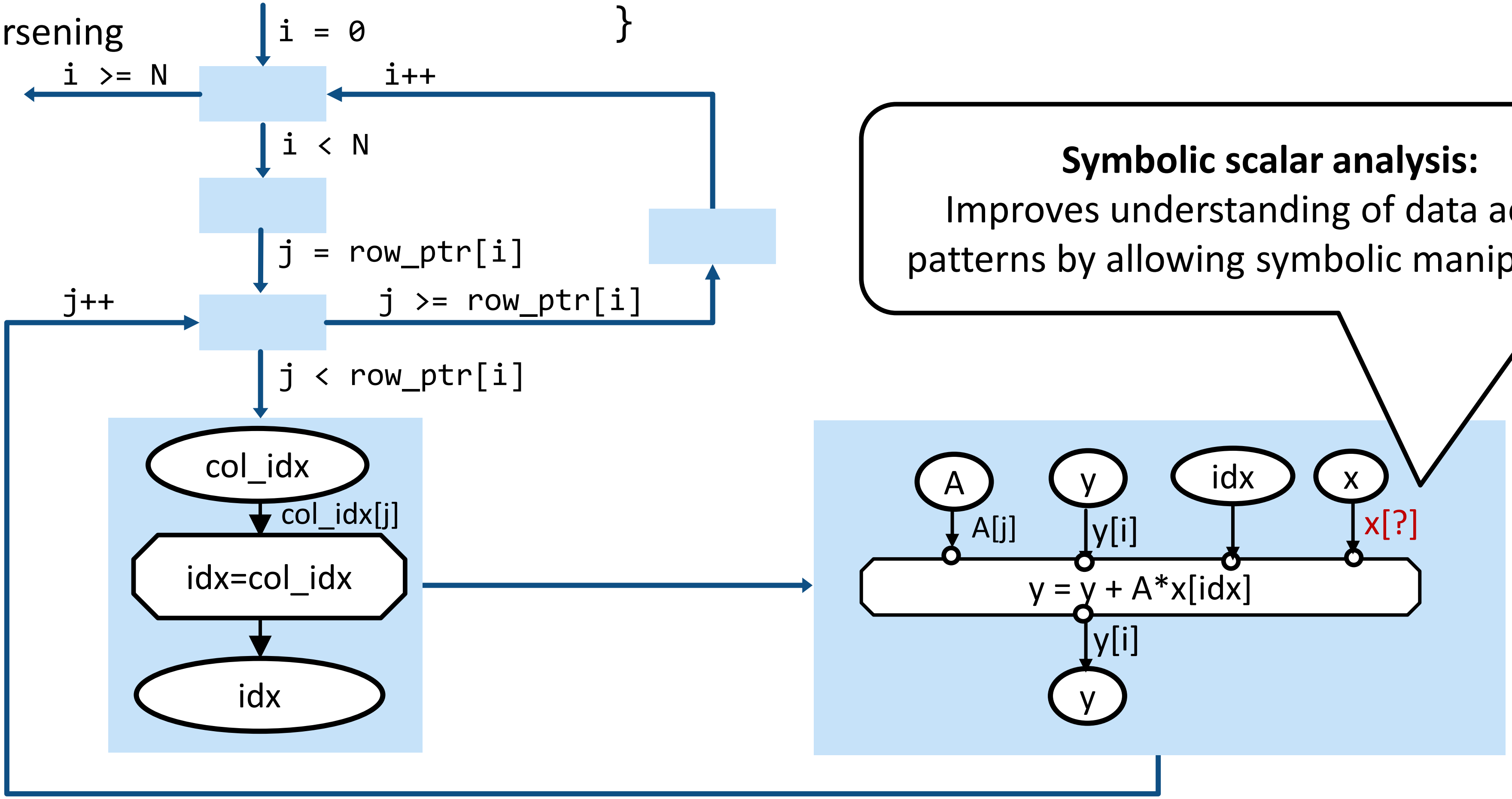
Indirect accesses

C to DaCe: SpMV

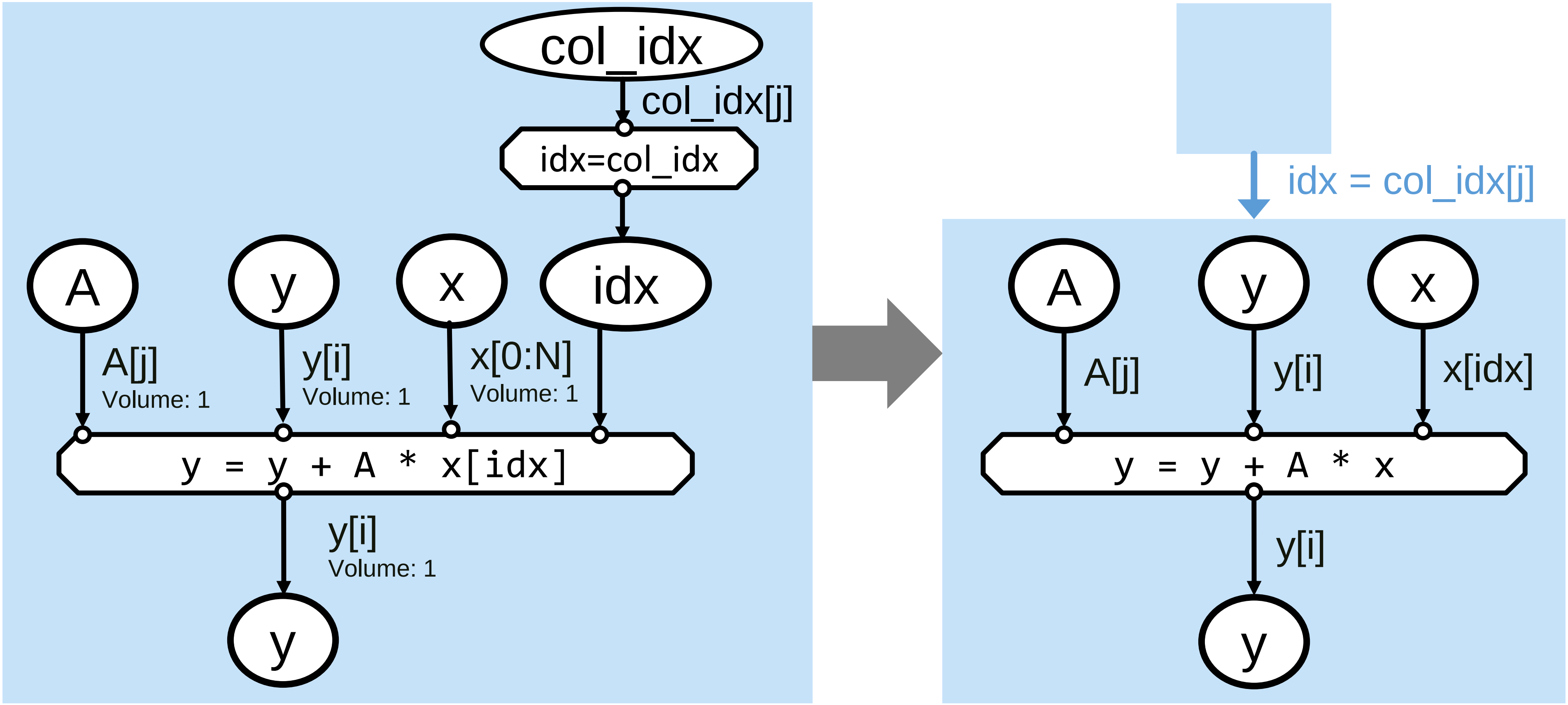
```
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
    }
}
```

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening

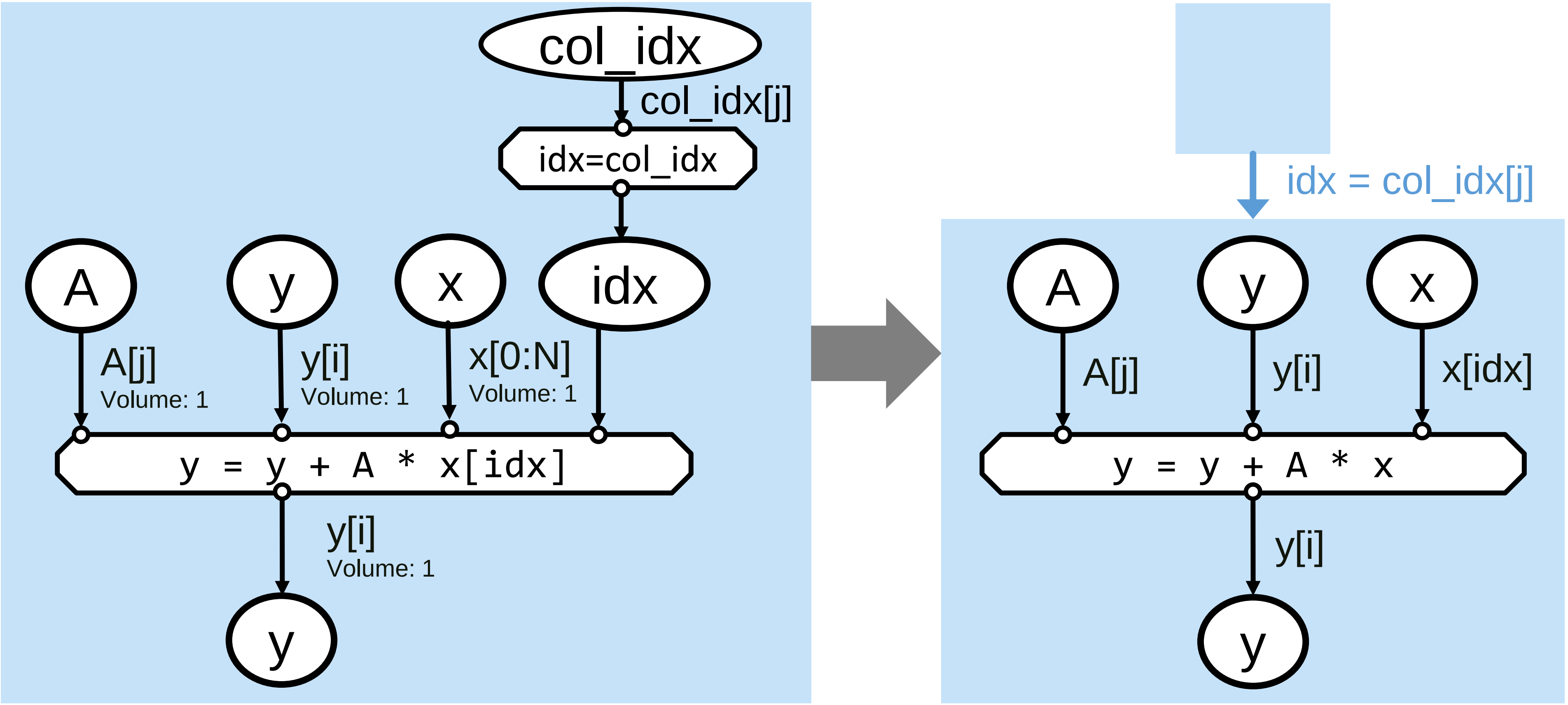
Symbolic scalar analysis:
Improves understanding of data access patterns by allowing symbolic manipulation



Symbolic scalar analysis



Symbolic scalar analysis

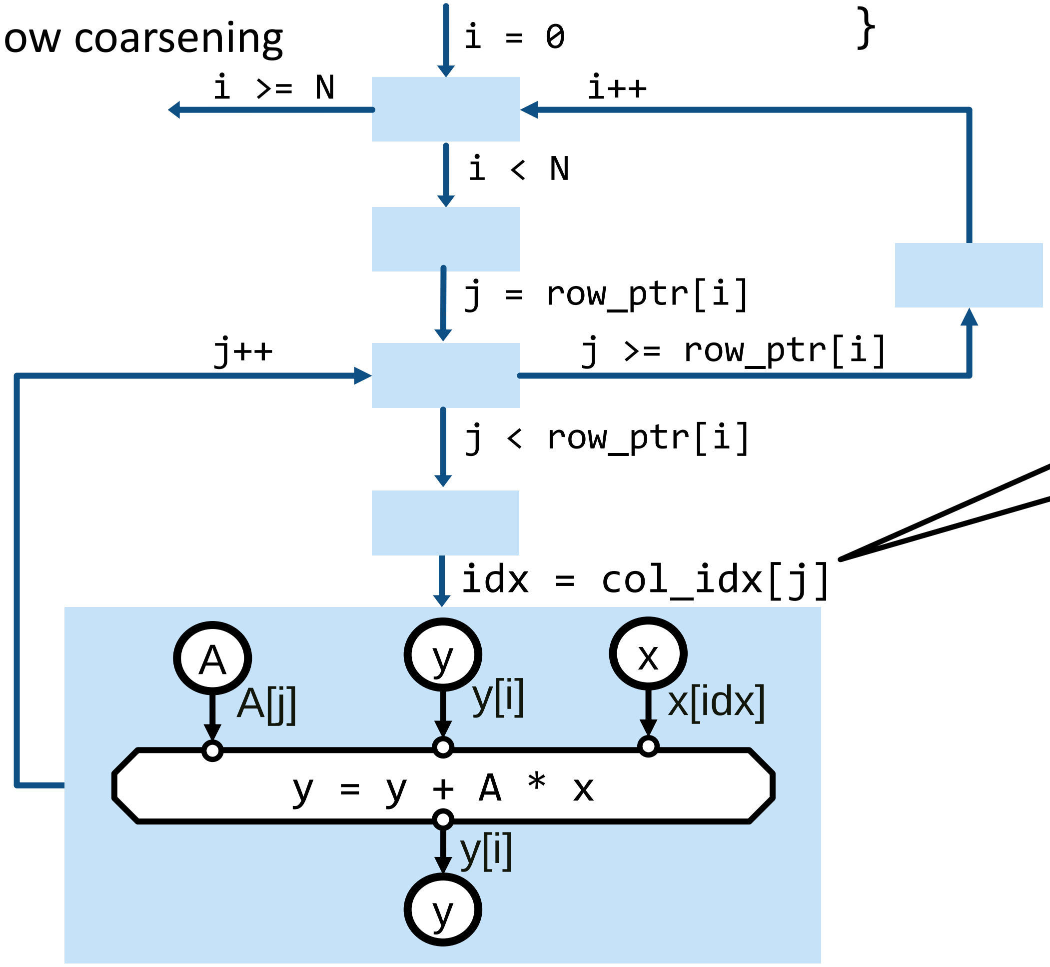


Indirect accesses

C to DaCe: SpMV

```
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
    }
}
```

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening

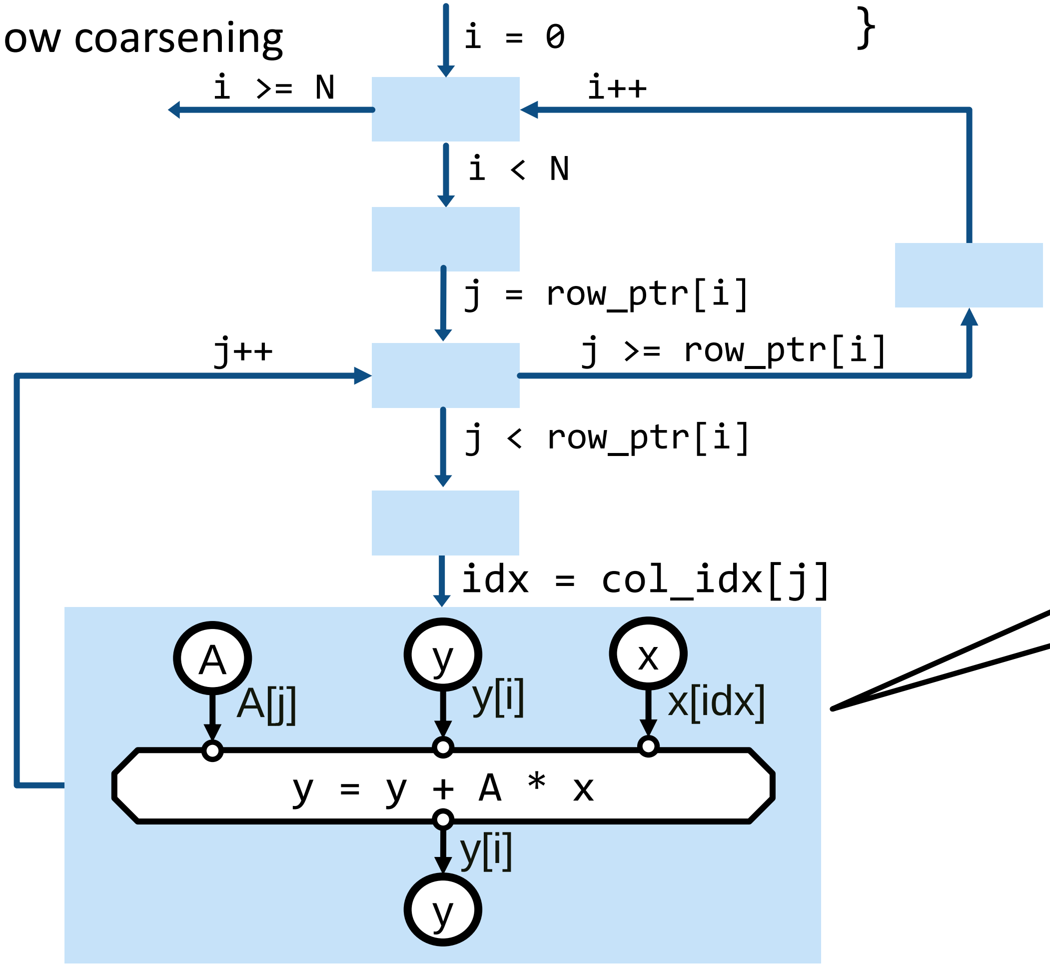


Symbolic scalar analysis:
Improves understanding of data access patterns by allowing symbolic manipulation

C to DaCe: SpMV

```c
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
    }
}
```

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening

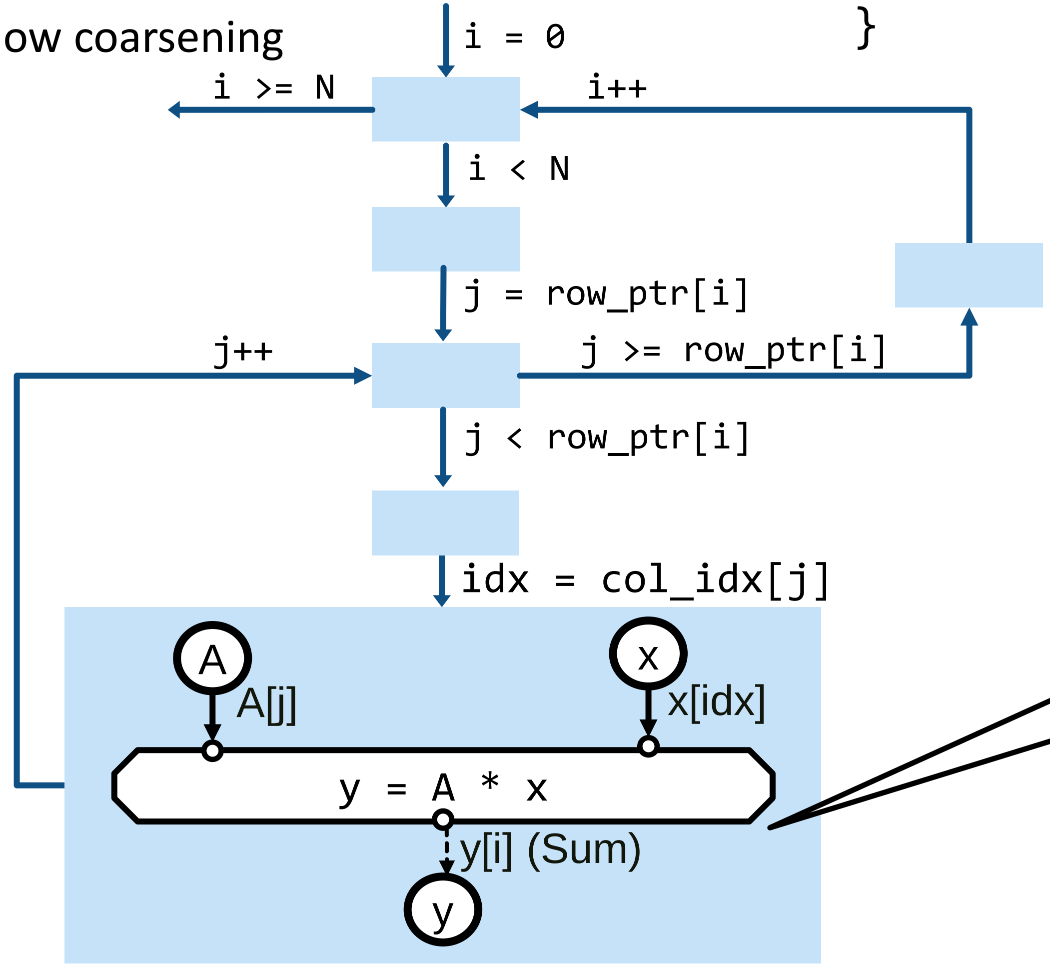


Update detection

C to DaCe: SpMV

```c
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
    }
}
```

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening

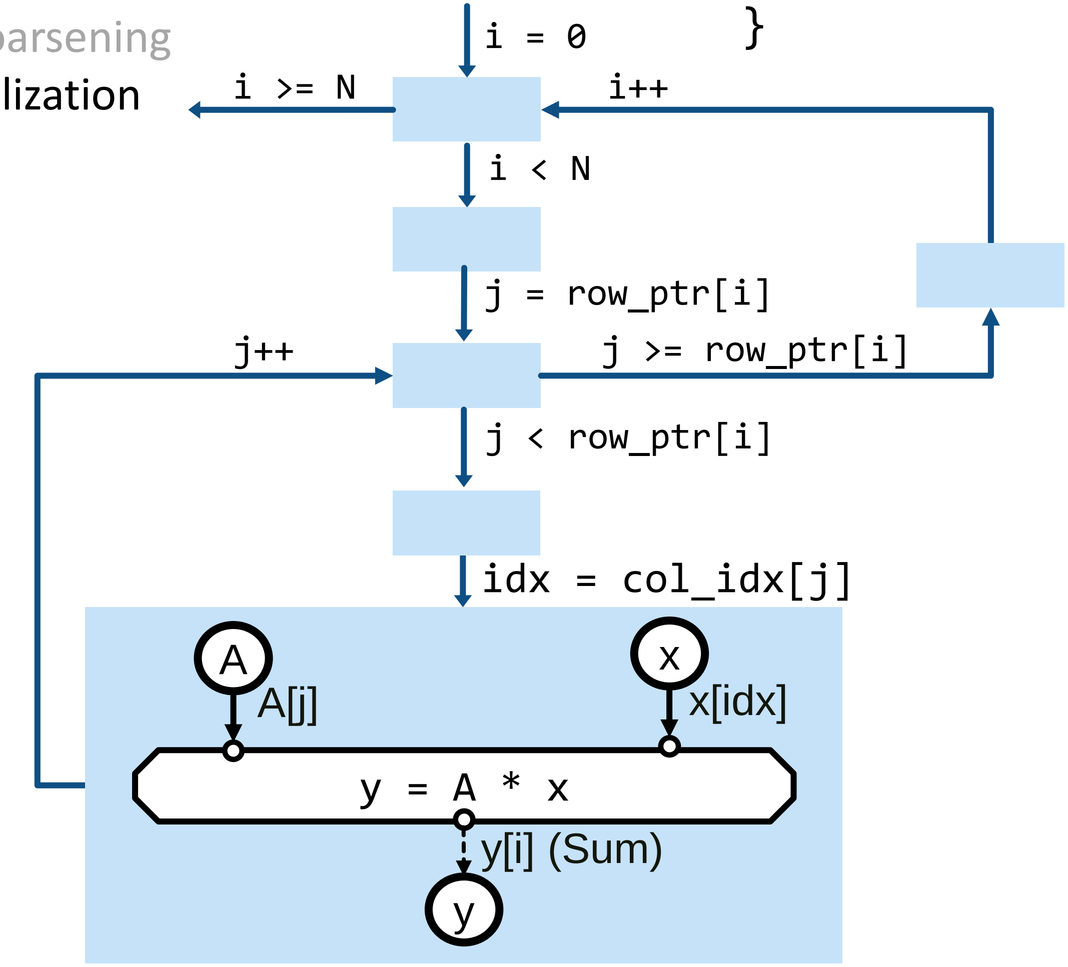


Update detection:
Allows more parallelization by creating the specialized *update* type of assignment, supporting conflict resolution for multiple operations

C to DaCe: SpMV

```
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
    }
}
```

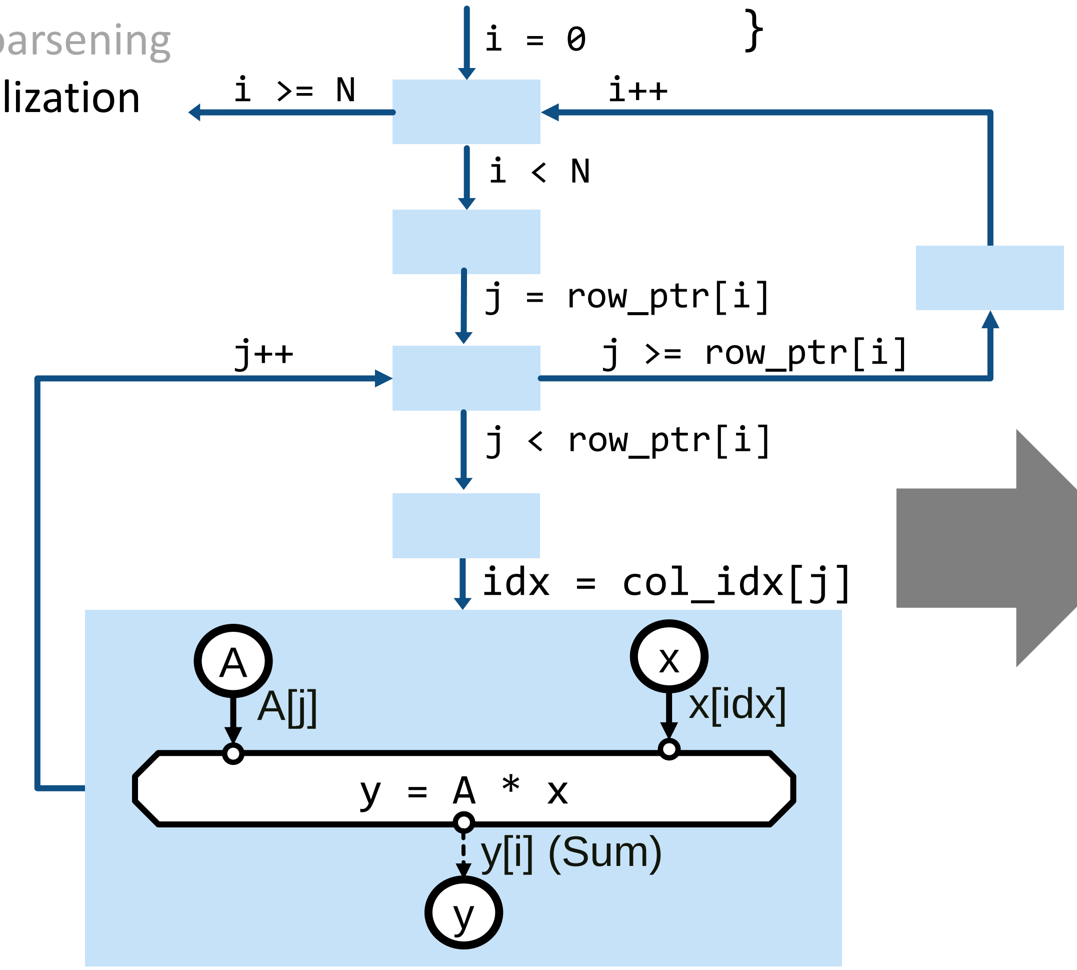
1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening
4. Autoparallelization

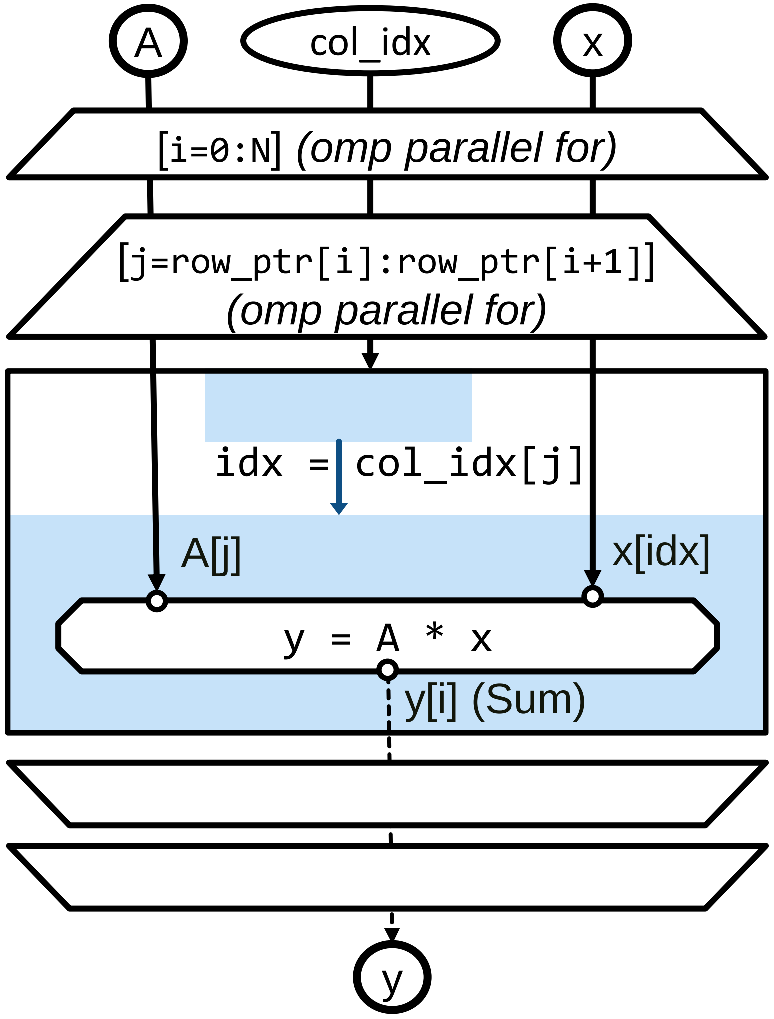


C to DaCe: SpMV

```
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
    }
}
```

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening
4. Autoparallelization

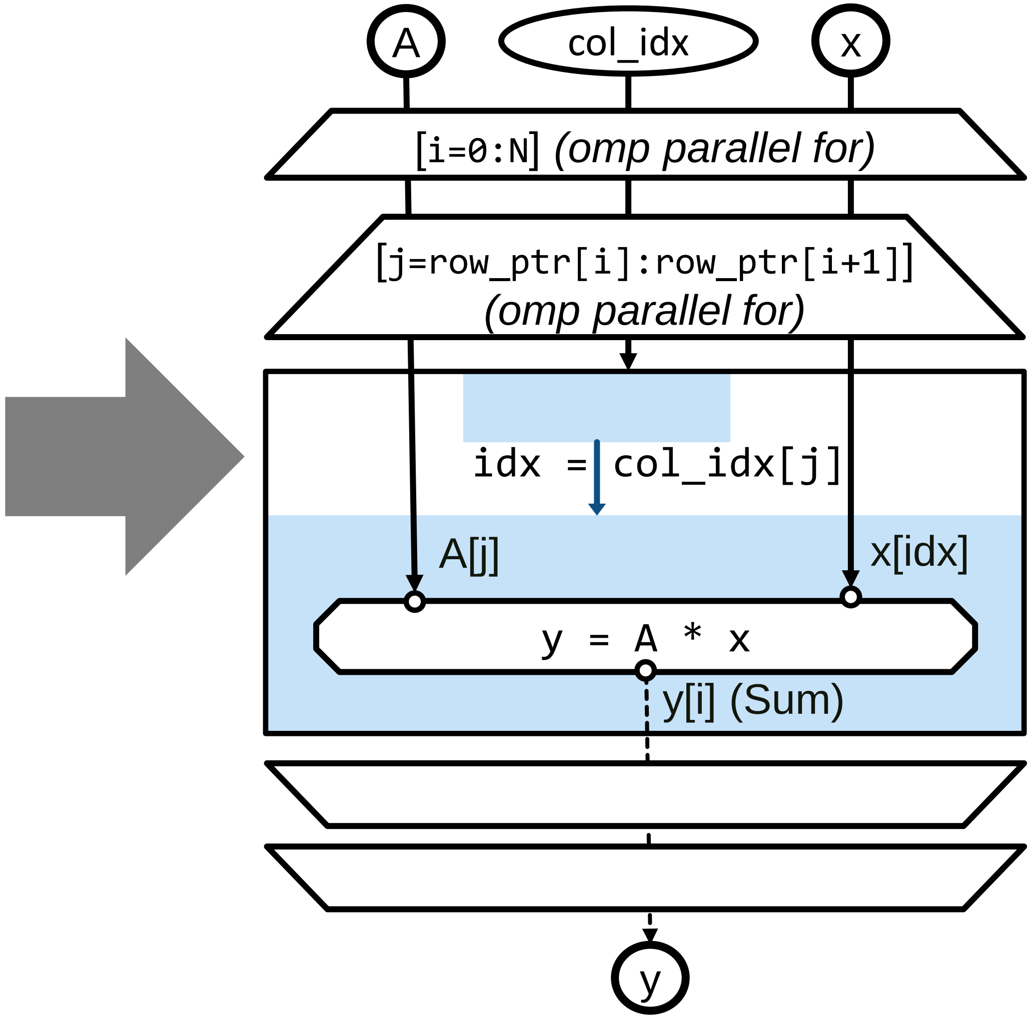




C to DaCe: SpMV

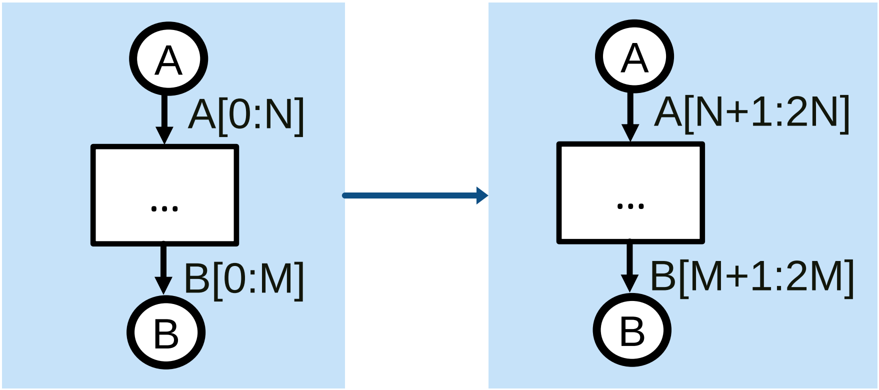
1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening
4. Autoparallelization

```
for (int i = 0; i < N ; i++){
   for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx =  col_idx[j];
        y[i] +=  A[j] * x[idx];
   }
}
```



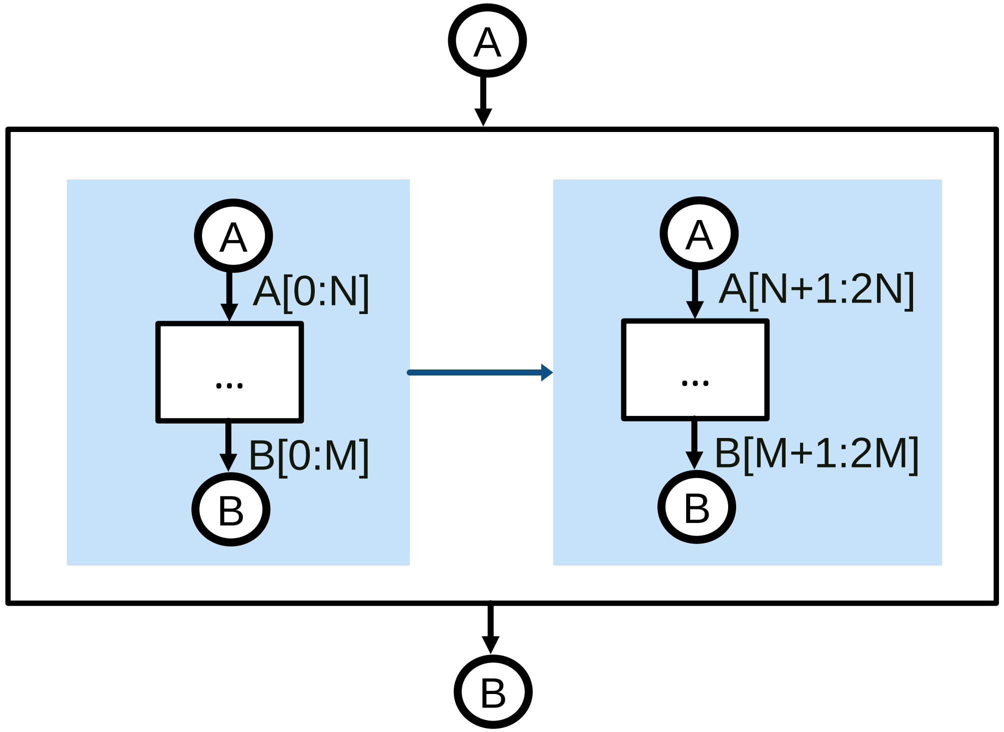
DaCe	polly	pluto	icc - parallel
✓	✗	✗✗	✓

Access pattern propagation



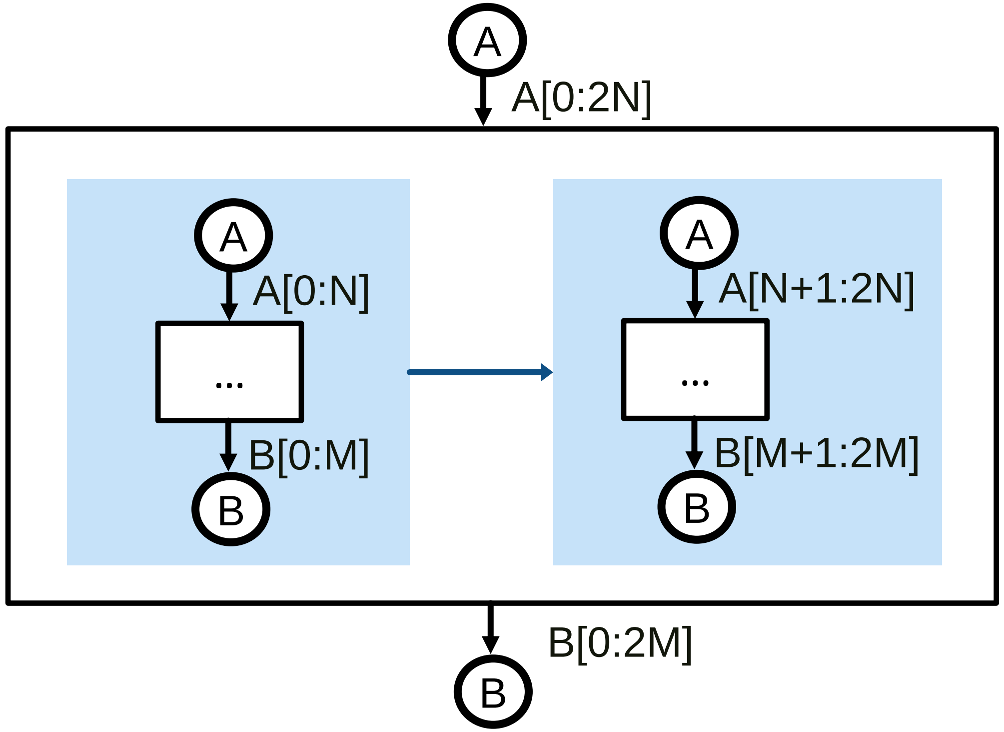
Difficult & expensive whole program analysis

Access pattern propagation



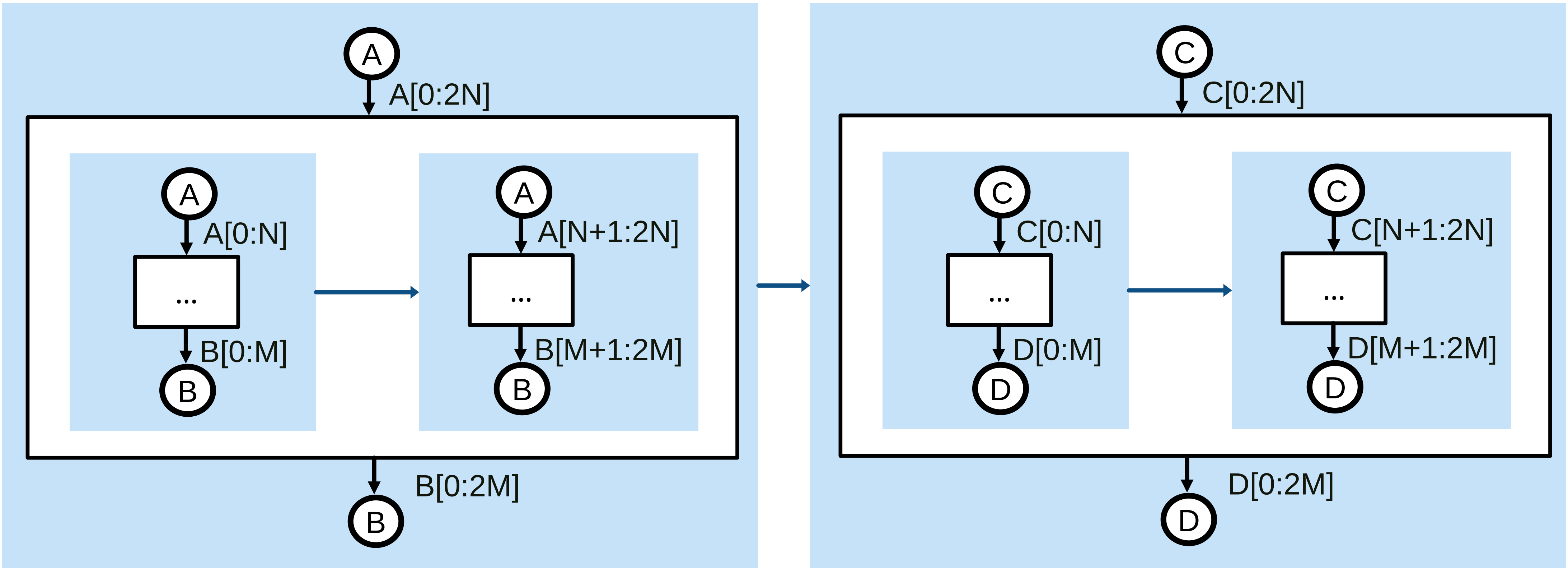
Difficult & expensive whole program analysis

Access pattern propagation

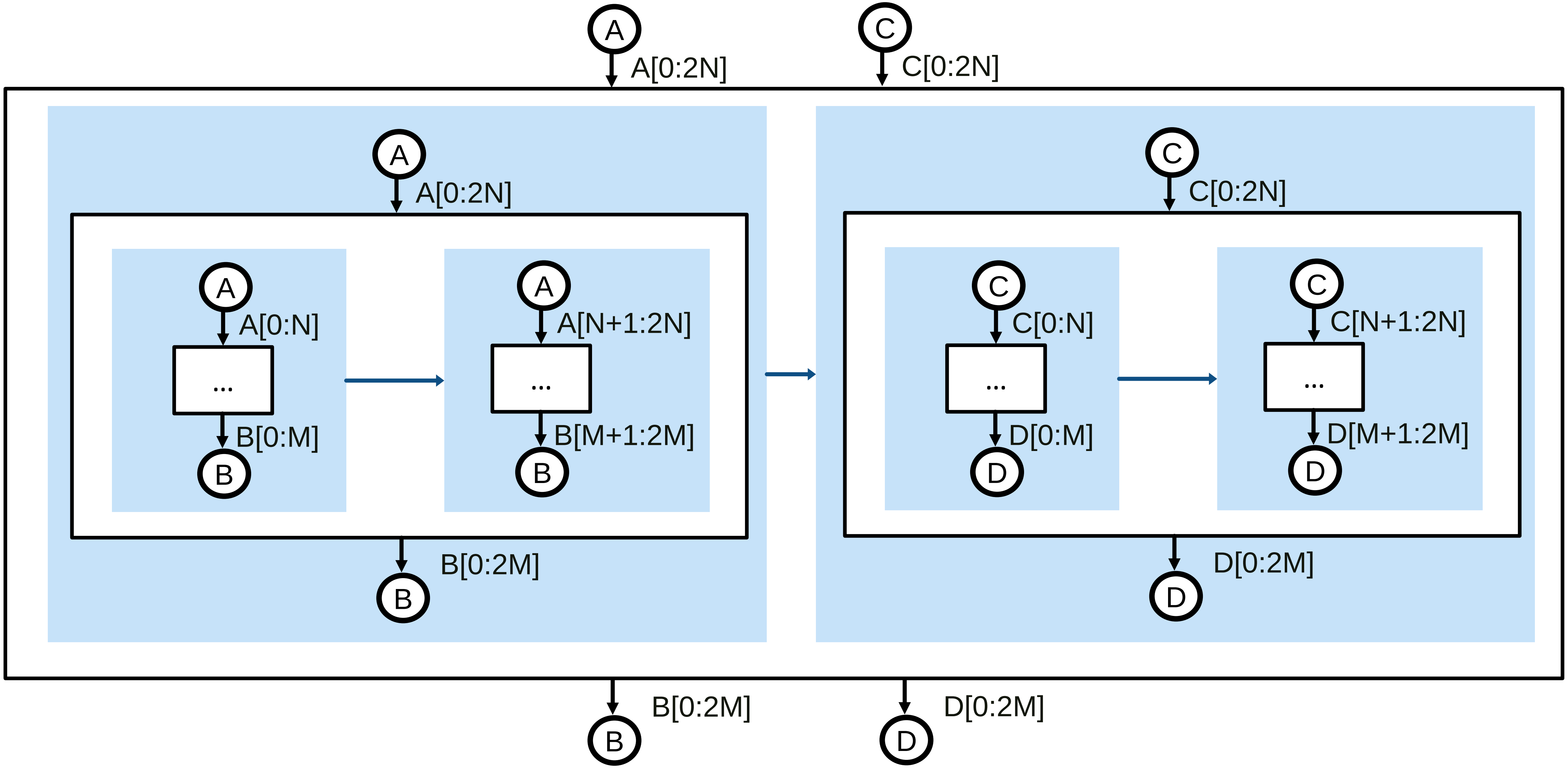


Difficult & expensive whole program analysis

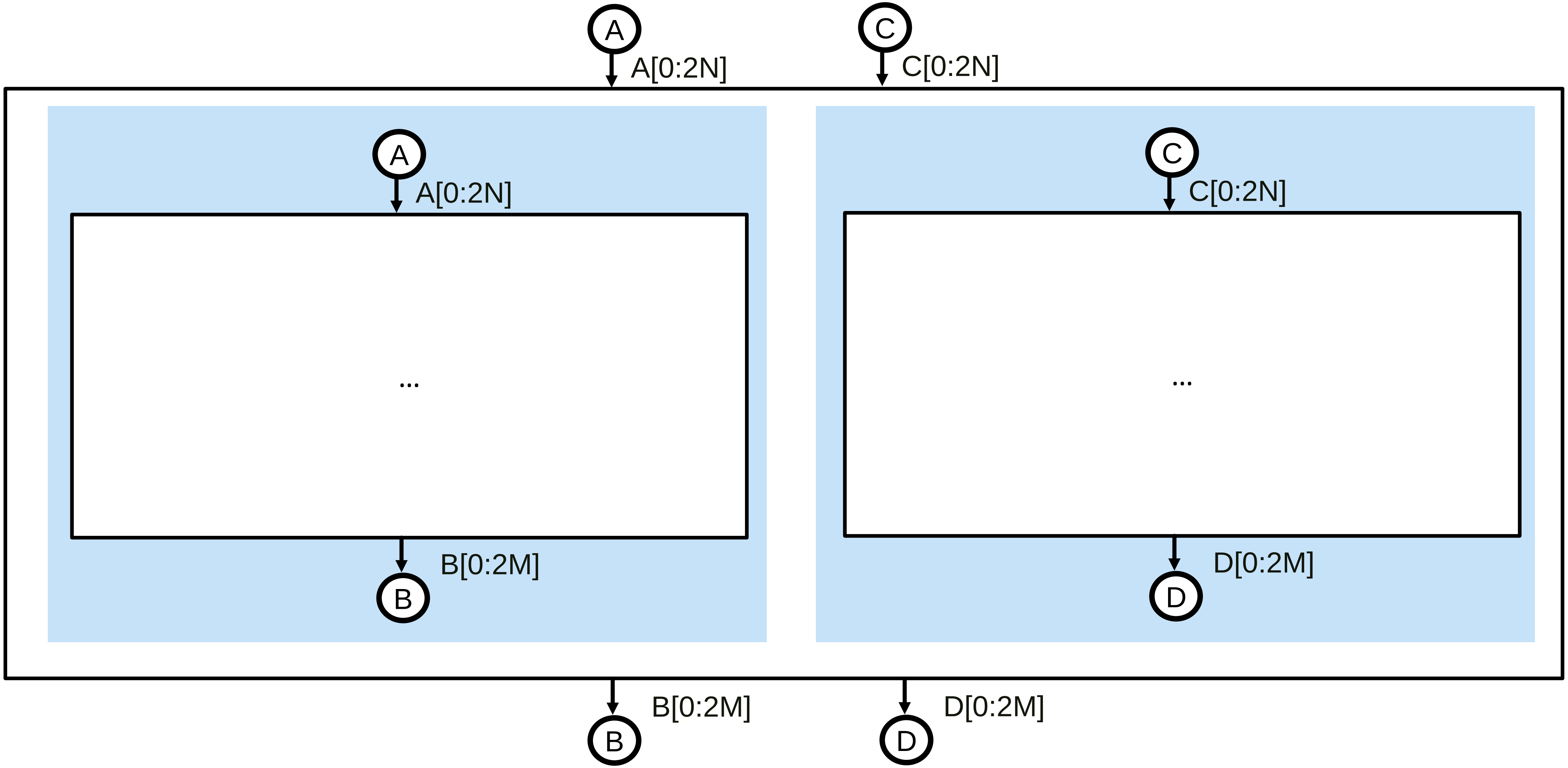
Access pattern propagation



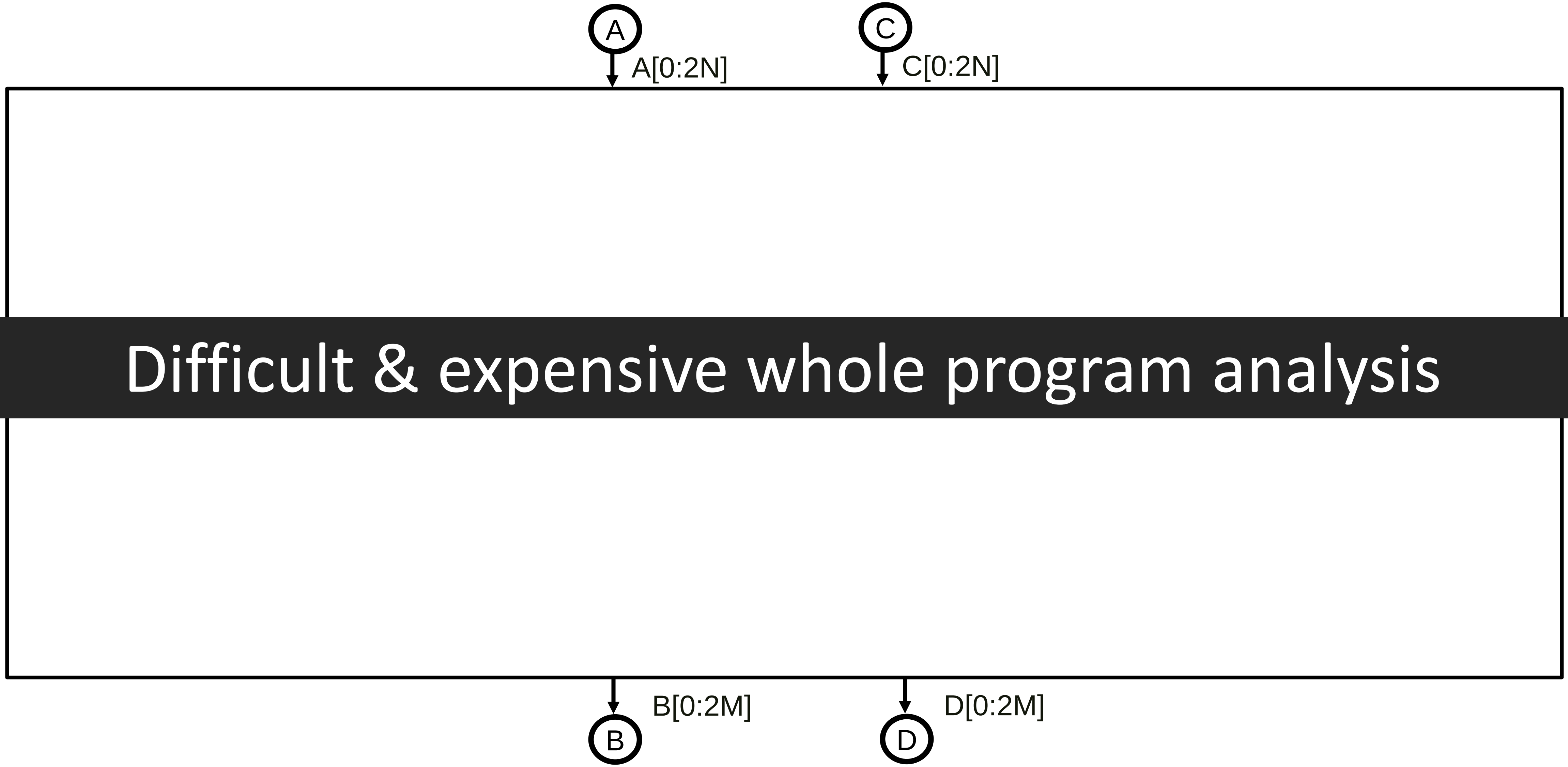
Access pattern propagation



Access pattern propagation

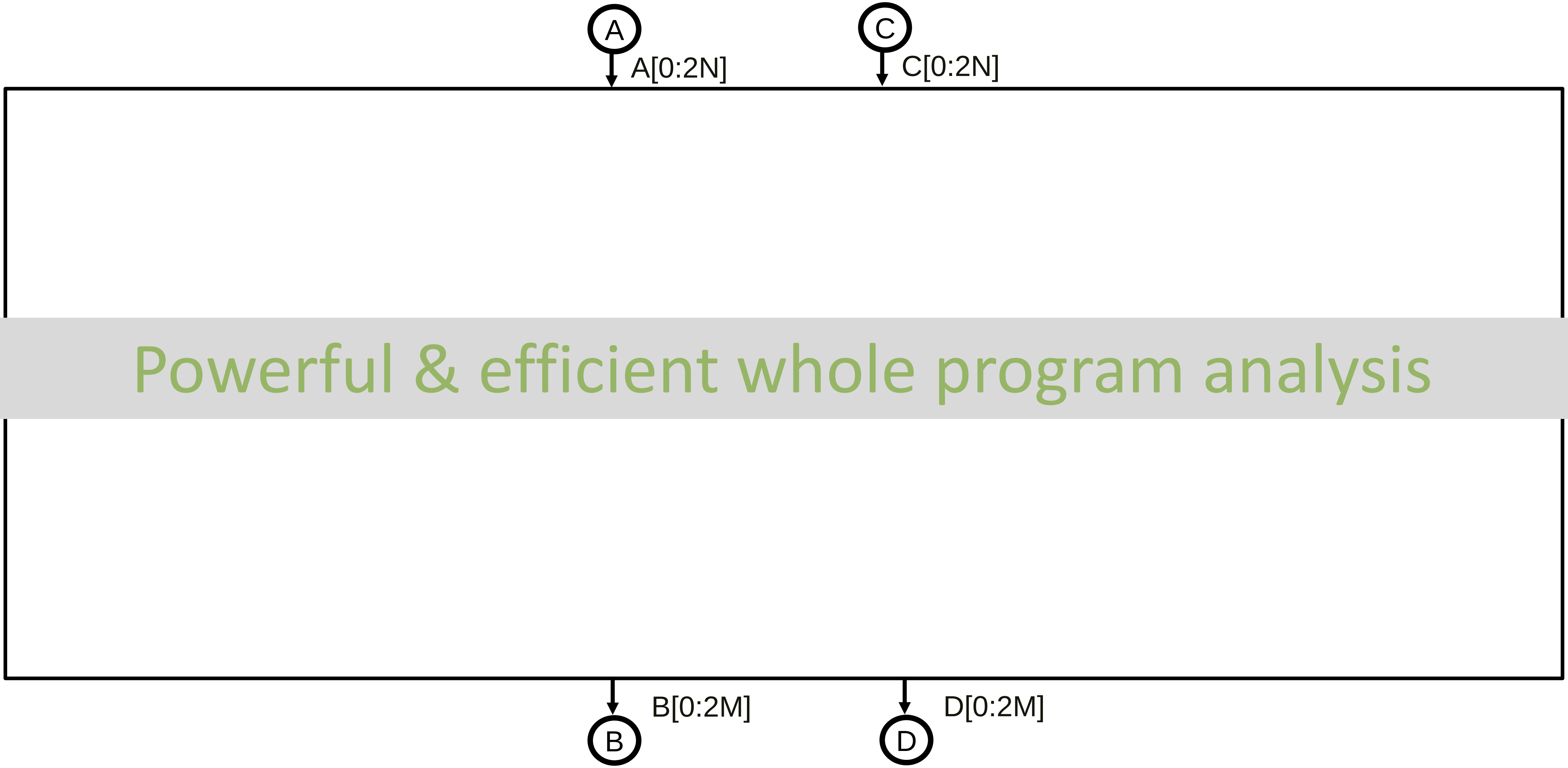


Access pattern propagation



Difficult & expensive whole program analysis

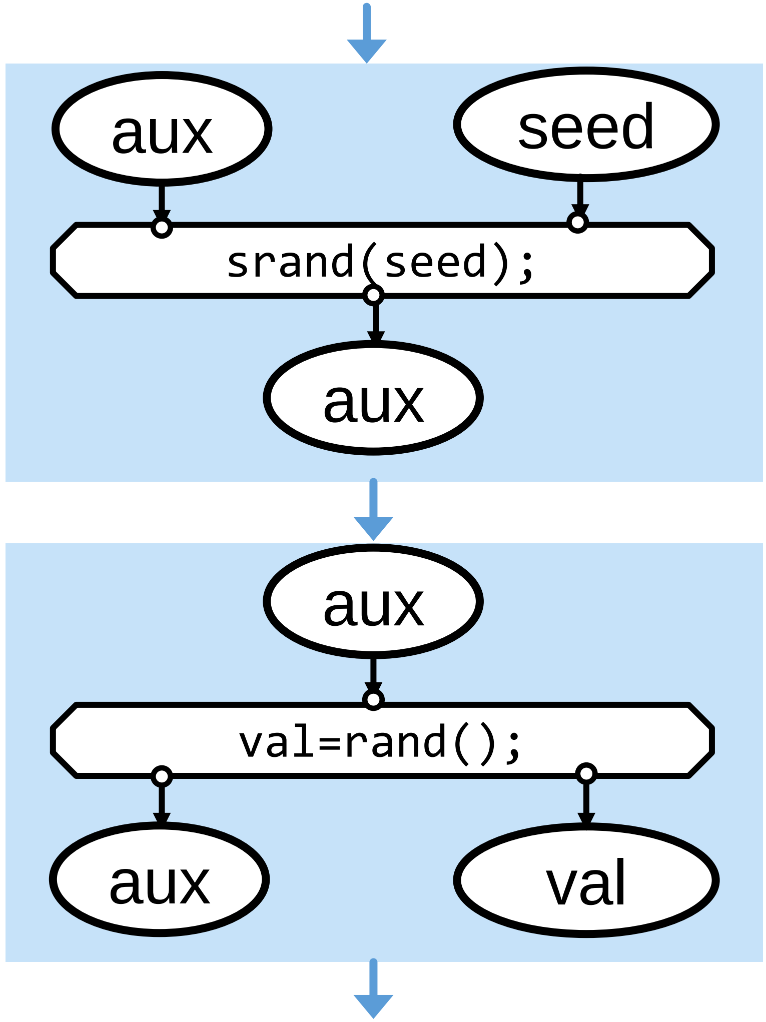
Access pattern propagation



Powerful & efficient whole program analysis

Enforcing order for data-less dependencies

- Global dictionary of states
- By default, all calls within a library should share a state
- Lists of stateless functions/libraries avoid needless complication of the SDFG



Polybench

- 30 benchmarks
- Provided as tests for polyhedral compilers
- Small kernels (tens – hundreds LOC)

Pointers and aliasing

No manual tuning

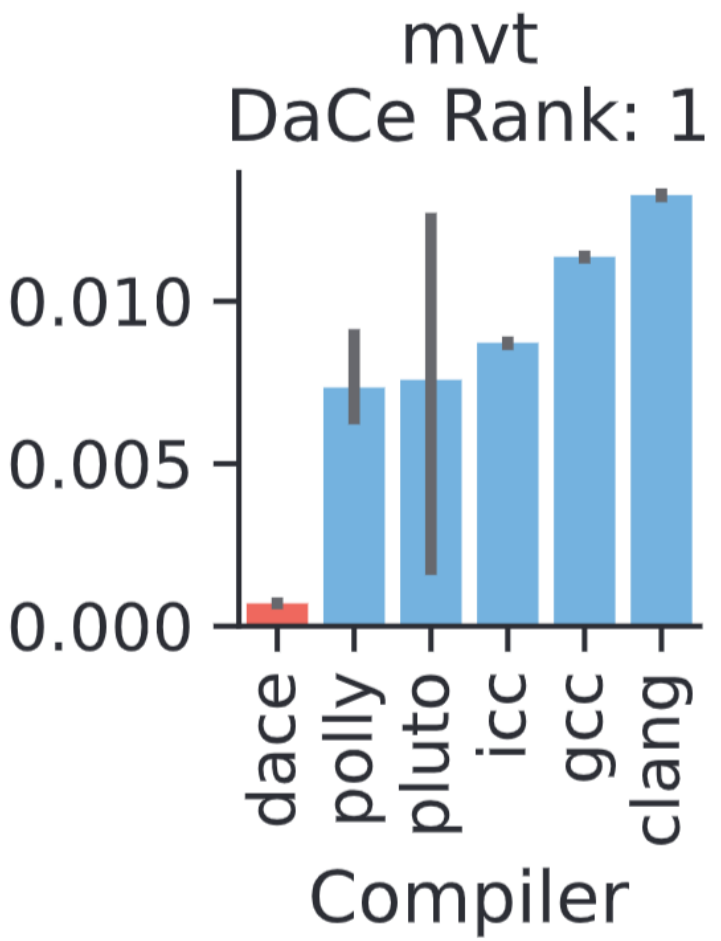
Full code analysis (No SCOP)

*No polyhedral analysis (yet)

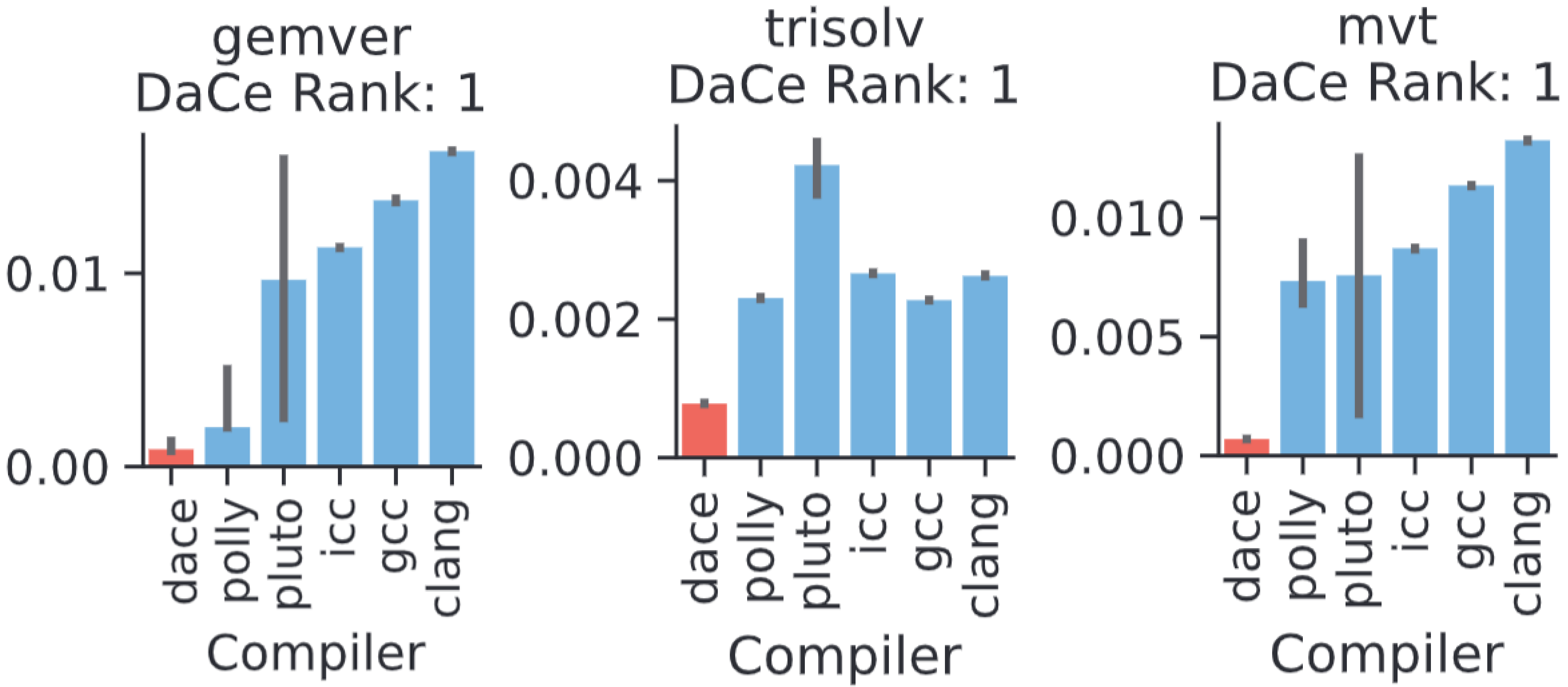
Polybench - MVT

```
for (i = 0; i < n; i++)
{
  for (j = 0; j < n; j++)
  {
    x1[i] = x1[i] + A[i][j] * y_1[j];
  }
}
for (i = 0; i < n; i++)
{
  for (j = 0; j < n; j++)
  {
    x2[i] = x2[i] + A[j][i] * y_2[j];
  }
}
```

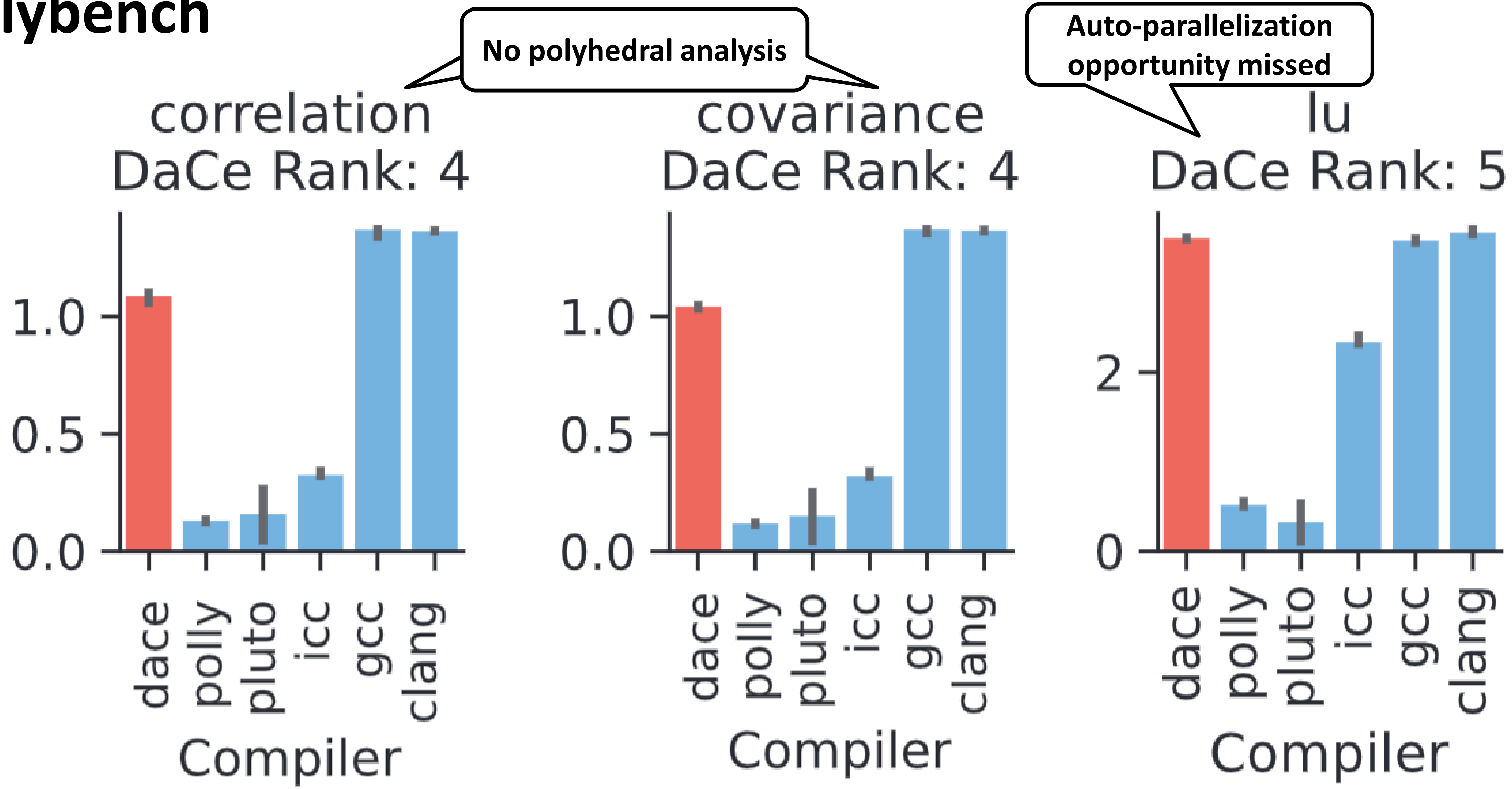
Update detection



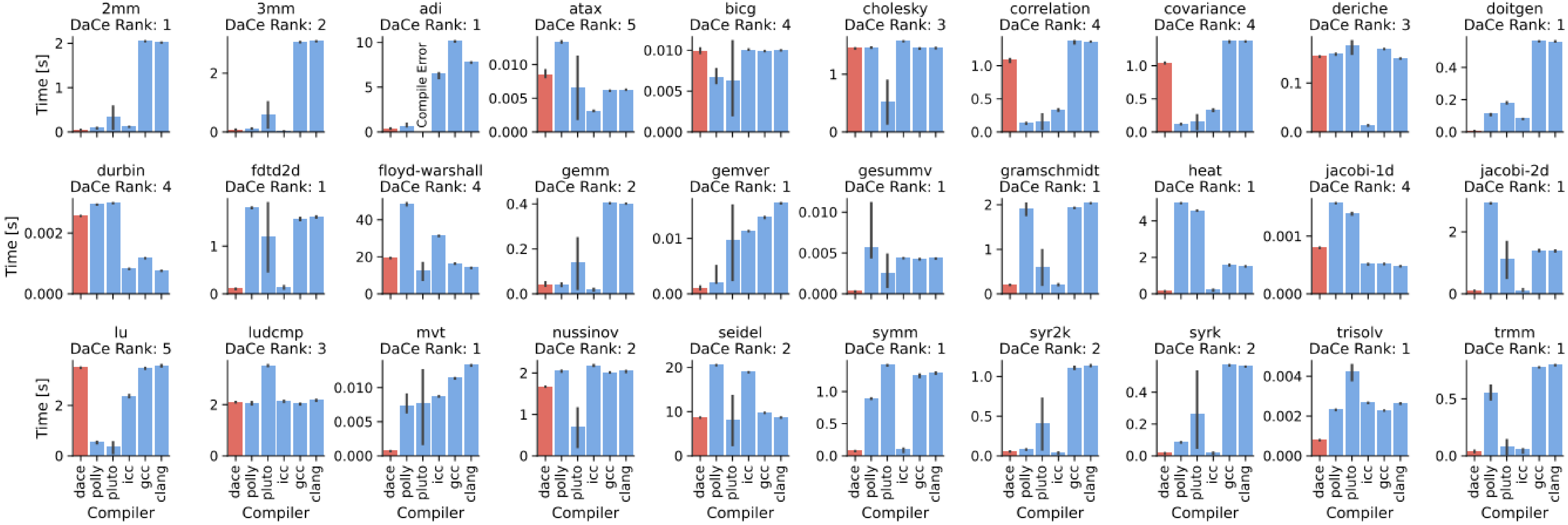
Polybench


gemver
DaCe Rank: 1
0.01
0.00
dace
polly
pluto
icc
gcc
clang
Compiler
trisolv
DaCe Rank: 1
0.004
0.002
0.000
dace
polly
pluto
icc
gcc
clang
Compiler
mvt
DaCe Rank: 1
0.010
0.005
0.000
dace
polly
pluto
icc
gcc
clang
Compiler

Polybench


No polyhedral analysis
Auto-parallelization opportunity missed
correlation
DaCe Rank: 4
covariance
DaCe Rank: 4
lu
DaCe Rank: 5
1.0
0.5
0.0
2
0
dace
polly
pluto
icc
gcc
clang
Compiler

Polybench Overview

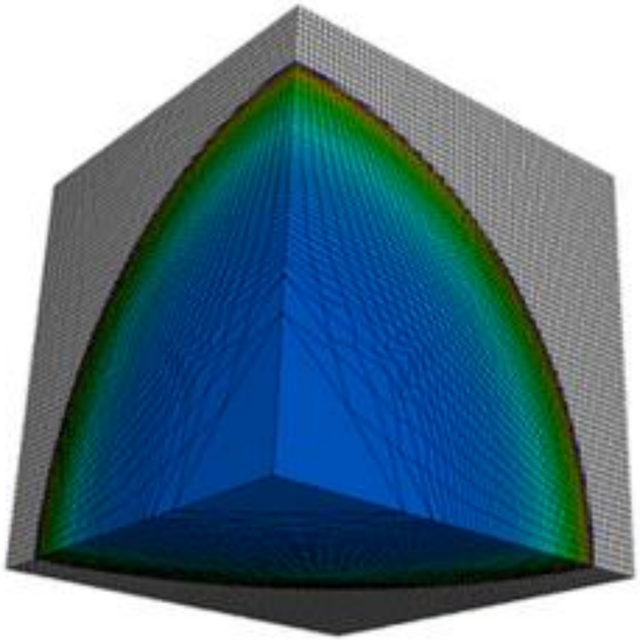


LULESH

- **Exascale proxy app from LLNL**

- ***Unstructured* hydrodynamics solver**

```
elemX[0] = domx[nd0i];
elemX[1] = domx[nd1i];
elemX[2] = domx[nd2i];
elemX[3] = domx[nd3i];
elemX[4] = domx[nd4i];
elemX[5] = domx[nd5i];
elemX[6] = domx[nd6i];
elemX[7] = domx[nd7i];
```



Indirect accesses

LULESH

- **Developers provide manually tuned OpenMP implementation**

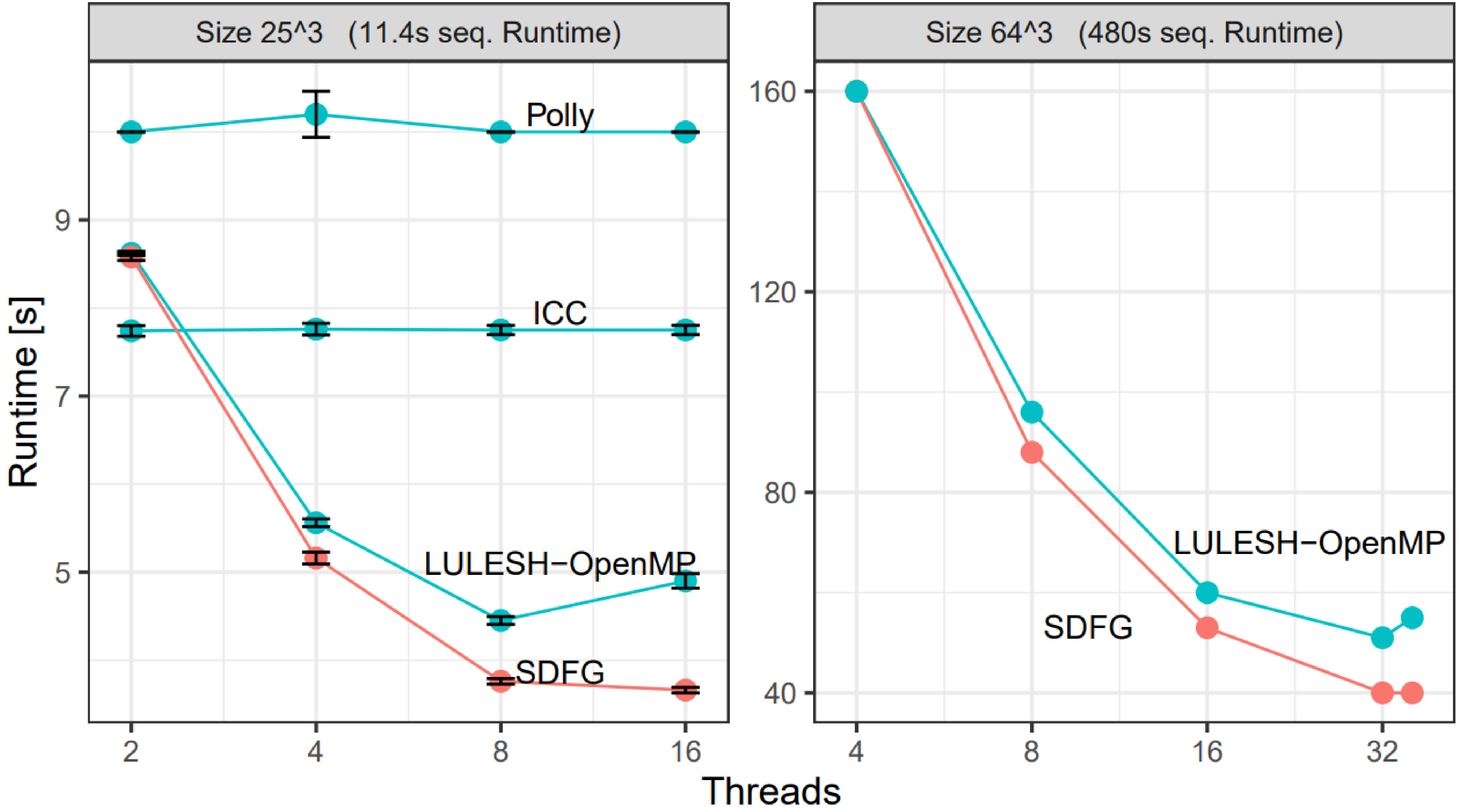
Serial result accumulation:

```
for( Index_t lnode=0 ; lnode<8 ; ++lnode ) {
        Index_t gnode = elemToNode[lnode];
        domain.fx(gnode) += fx_local[lnode];
        domain.fy(gnode) += fy_local[lnode];
        domain.fz(gnode) += fz_local[lnode];
     }
```

Parallel result accumulation:

```
#pragma omp parallel for firstprivate(numNode)
     for( Index_t gnode=0 ; gnode<numNode ; ++gnode )
     {
        Index_t count = domain.nodeElemCount(gnode) ;
        Index_t *cornerList = domain.nodeElemCornerList(gnode) ;
        Real_t fx_tmp = Real_t(0.0) ;
        Real_t fy_tmp = Real_t(0.0) ;
        Real_t fz_tmp = Real_t(0.0) ;
        for (Index_t i=0 ; i < count ; ++i) {
           Index_t ielem = cornerList[i] ;
           fx_tmp += fx_elem[ielem] ;
           fy_tmp += fy_elem[ielem] ;
           fz_tmp += fz_elem[ielem] ;
        }
        domain.fx(gnode) = fx_tmp ;
        domain.fy(gnode) = fy_tmp ;
        domain.fz(gnode) = fz_tmp ;
     }
```

LULESH


Size 25^3 (11.4s seq. Runtime)
Size 64^3 (480s seq. Runtime)
Runtime [s]
Threads
Polly
ICC
LULESH−OpenMP
SDFG
9
7
5
2
4
8
16
160
120
80
40
4
8
16
32
LULESH−OpenMP
SDFG

Conclusion

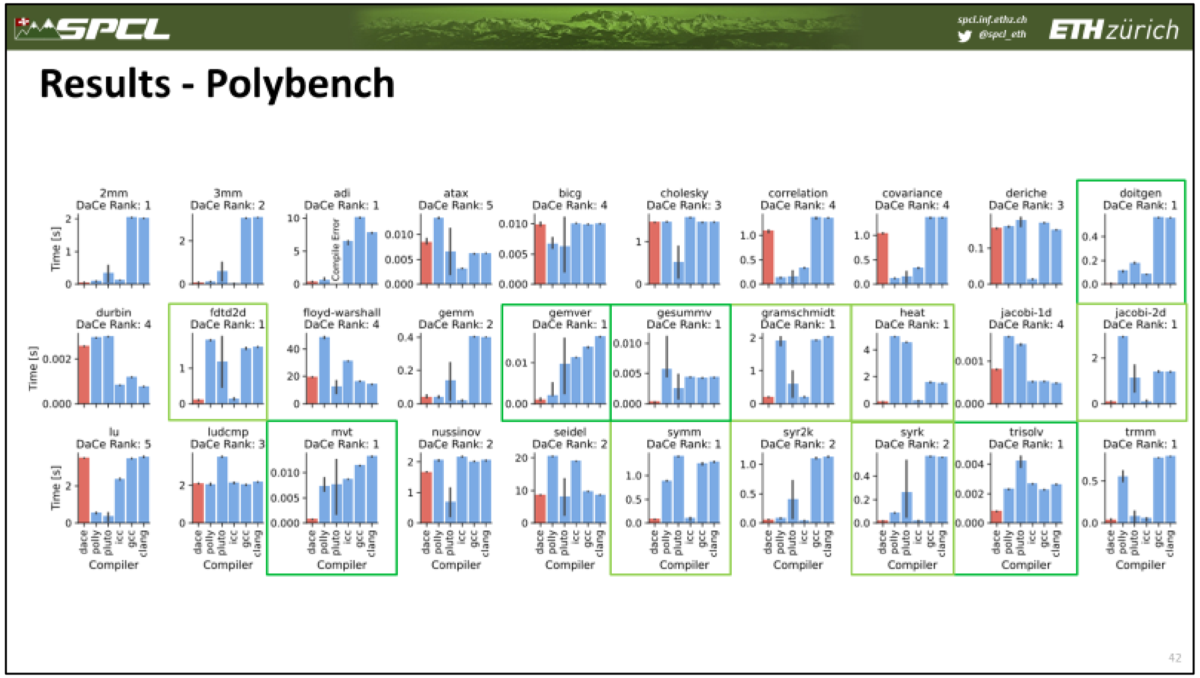
Symbolic analysis is needed to understand data movement

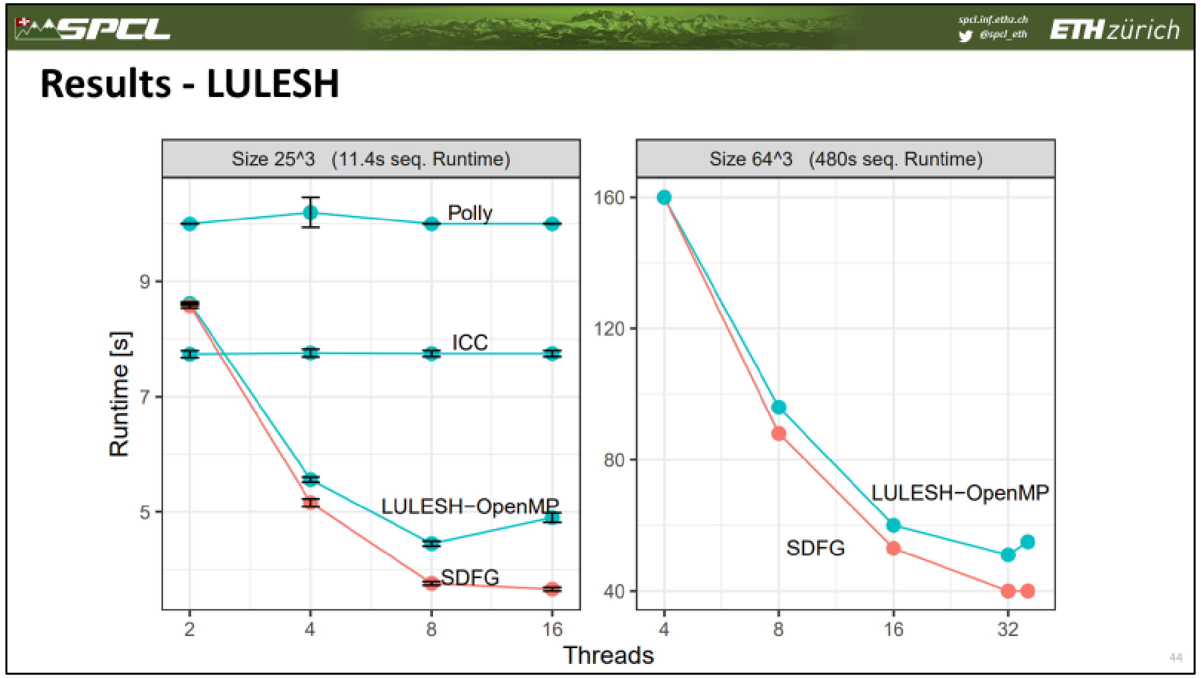
Update detection opens parallelization opportunities

Access pattern propagation allows efficient whole program analysis

Understanding data movement is key to performance & portability

Thank you!





Paper: https://arxiv.org/abs/2112.11879

Code prototype: https://github.com/spcl/c2dace DaCe: https://github.com/spcl/dace