Static Dataflow: Compiling Global Control into Local Control

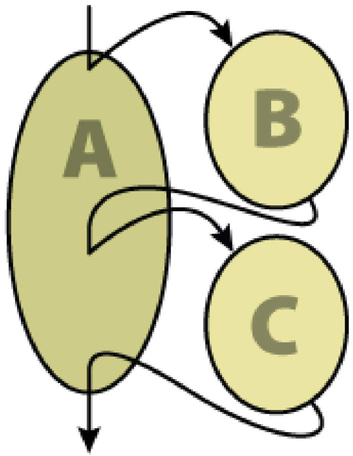
Pritish Jetley, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign

pjetley2@illinois.edu

The Need for Abstractions

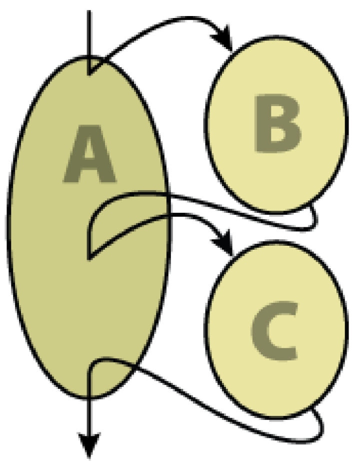
- Traditional programming models don't provide the right frameworks for complicated Science & Engineering applications
  - Modularity
  - Separation of concerns
  - Programming productivity

Modularity in MPI

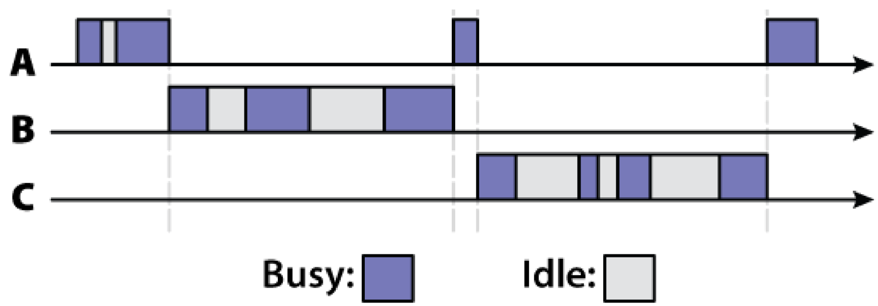


- A must call B & C (no order)

Modularity in MPI

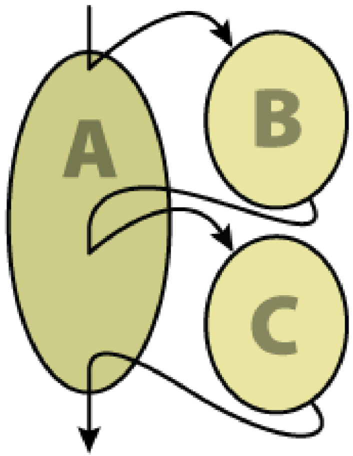


- A must call B & C (no order)
- In MPI, must serialize calls to different modules

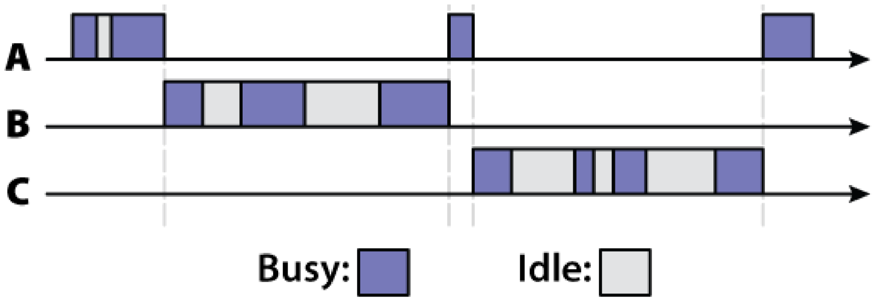


Images courtesy and ©David Kunzman

Modularity in MPI



- A must call B & C (no order)
- In MPI, must serialize calls to different modules
- Or, insert cross-module wildcard receives



Charm++

- Application composed of collections of objects
  - Collections = arrays

Charm++

- Application composed of collections of objects
  - Collections = arrays
- Object-based virtualization: adaptive overlap

Charm++

- Application composed of collections of objects
  - Collections = arrays
- Object-based virtualization: adaptive overlap
- Communication = Asynch. method invocation
  - Methods cannot be preempted
  - Scheduler picks message and invokes on target

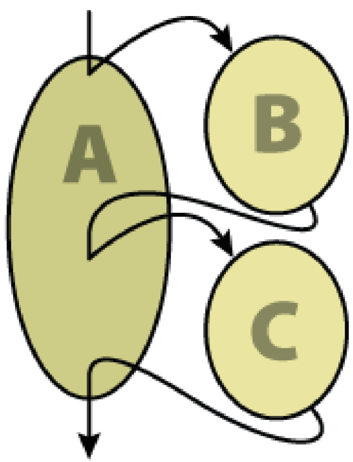
Charm++

- Application composed of collections of objects
  - Collections = arrays
- Object-based virtualization: adaptive overlap
- Communication = Asynch. method invocation
  - Methods cannot be preempted
  - Scheduler picks message and invokes on target
- Array-like syntax for addressing
  - `array1(17).f();`
  - `array2(F(x), G(z)).g();`
  - `thisProxy(thisIndex).h();`

Charm++

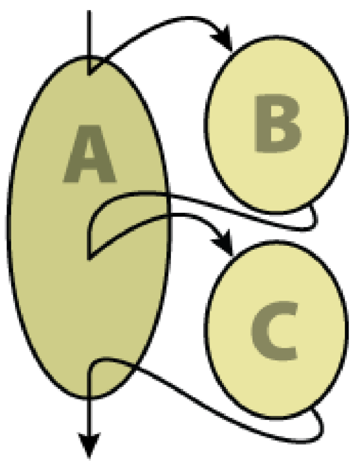
- Application composed of collections of objects
  - Collections = arrays
- Object-based virtualization: adaptive overlap
- Communication = Asynch. method invocation
  - Methods cannot be preempted
  - Scheduler picks message and invokes on target
- Array-like syntax for addressing
  - `array1(17).f();`
  - `array2(F(x), G(z)).g();`
  - `thisProxy(thisIndex).h();`
- Load balancing, communication optimization, etc.

Modularity in Charm++



- Many objects/processor

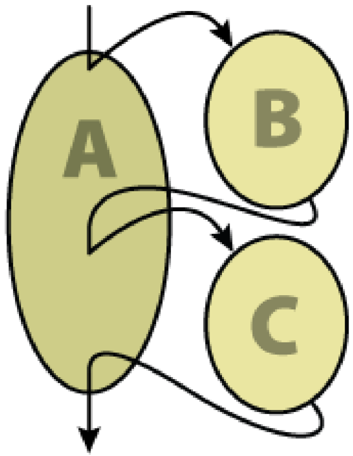
Modularity in Charm++



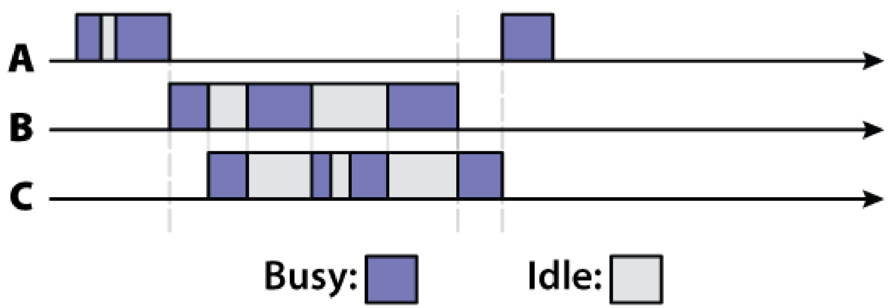
- Many objects/processor
- Scheduler sends messages to appropriate recipients

Images courtesy and ©David Kunzman

Modularity in Charm++



- Many objects/processor
- Scheduler sends messages to appropriate recipients
- Idle time of one overlapped with computation of other



Images courtesy and ©David Kunzman

However...

- Reactive specification of Charm++ programs

However...

- Reactive specification of Charm++ programs
  - Hard to follow global control/data flow

However...

- Reactive specification of Charm++ programs
  - Hard to follow global control/data flow
- Non-determinism in message delivery
  - Hard to reason about/debug programs

```
entry void call(){
 A[x].fun_1();
 A[x].fun_2();
}

entry void fun_1(){
  var = 2;
}

entry void fun_2(){
  var = 3;
}
```

Can we do better?

- Most Science/Engineering applications follow certain *patterns* of computation and communication

Can we do better?

- Most Science/Engineering applications follow certain *patterns* of computation and communication
- What is common among the following applications?
  - Matrix mult.
  - Jacobi
  - FFT
  - Unstructured Mesh Computations
  - Cutoff-Based Molecular Dynamics

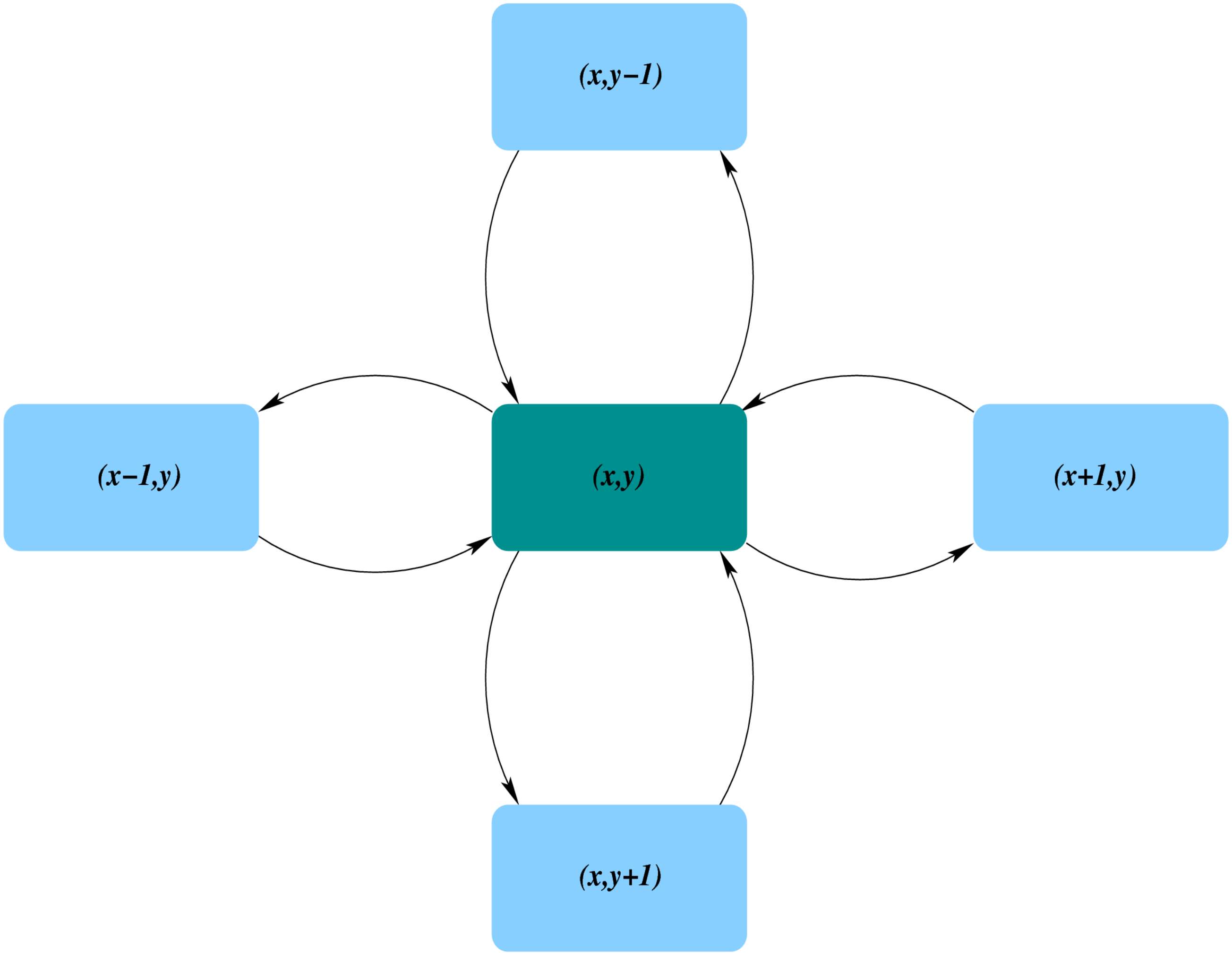
Can we do better?

- Most Science/Engineering applications follow certain *patterns* of computation and communication
- What is common among the following applications?
  - Matrix mult.
  - Jacobi
  - FFT
  - Unstructured Mesh Computations
  - Cutoff-Based Molecular Dynamics

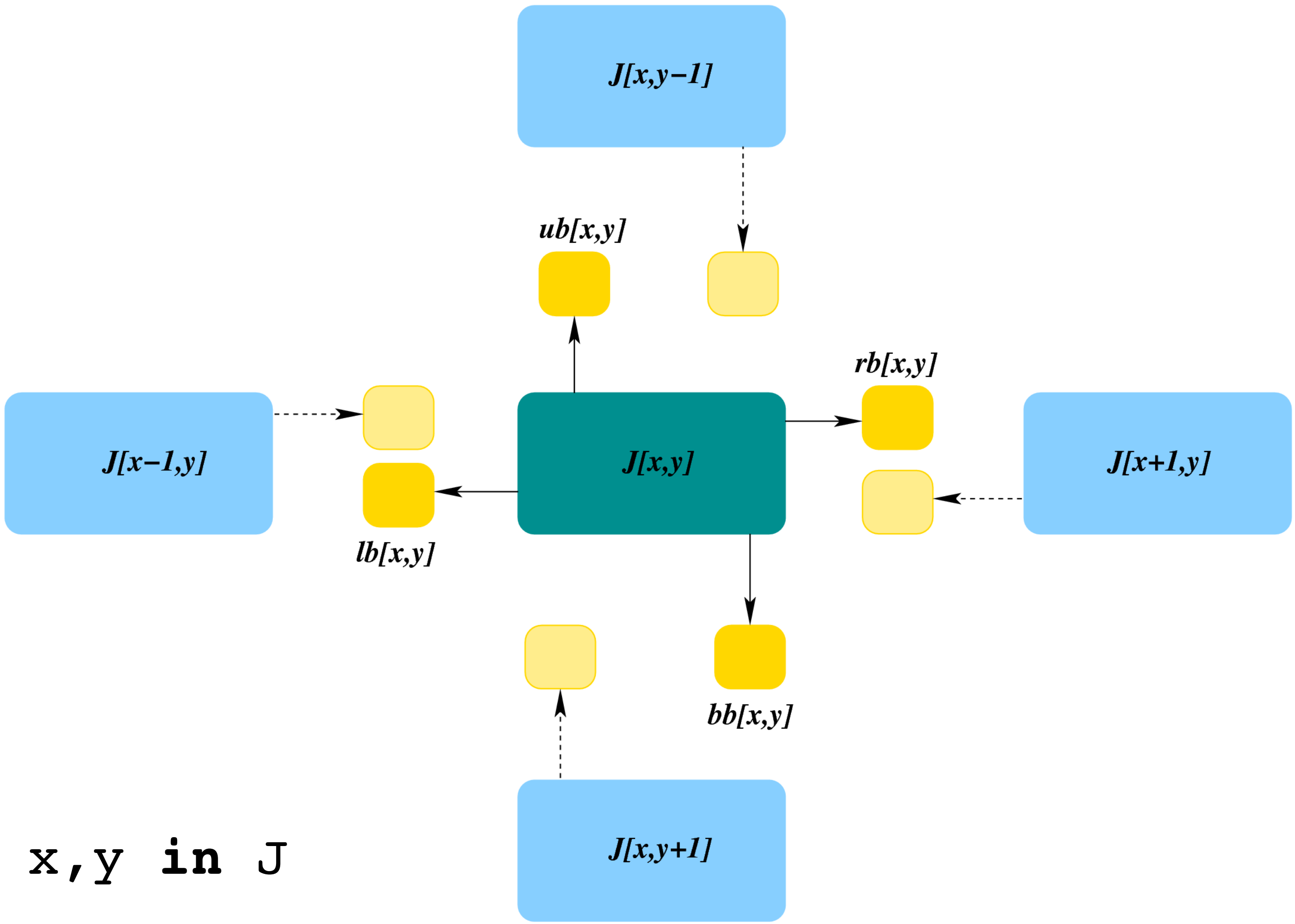
Static communication pattern

Static Dataflow

- Static patterns of communication
- Objects *produce* and *consume* data

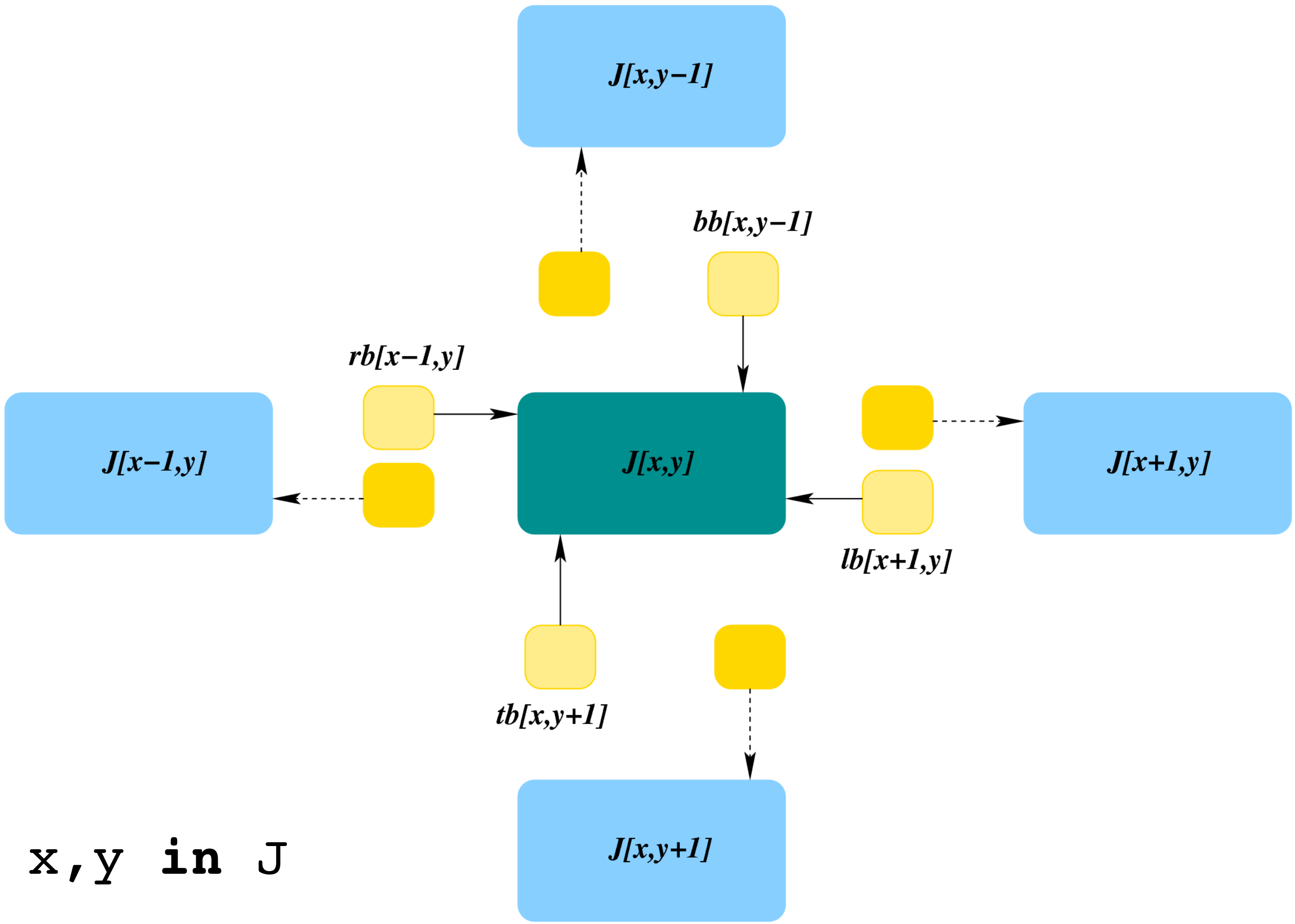


Jacobi in Charisma




```
foreach x,y in J
(lb[x,y],rb[x,y],tb[x,y],bb[x,y]) ← J[x,y].prodBorders();
J[x,y].consume(lb[x+1,y],rb[x-1,y],tb[x,y+1],bb[x,y-1]);
end-foreach
```

Jacobi in Charisma




```
foreach x,y in J
(lb[x,y],rb[x,y],tb[x,y],bb[x,y]) ← J[x,y].prodBorders();
J[x,y].consume(lb[x+1,y],rb[x-1,y],tb[x,y+1],bb[x,y-1]);
end-foreach
```

Charisma Semantics

- `foreach` statements execute across object arrays
  - Have associated methods

```
for I = 1 to MAX_ITER

  foreach x in A
   (p[x]) <- A[x].f();
  end-foreach

  foreach x in B
   (...) <- B[x].g();
  end-foreach

  foreach x in A
   B[x].h(p[x-1]);
  end-foreach

end-for
```

Charisma Semantics

- `foreach` statements execute across object arrays
  - Have associated methods
- Objects *produce* and *consume* parameters

```
for I = 1 to MAX_ITER

  foreach x in A
   (p[x]) <- A[x].f();
  end-foreach

  foreach x in B
   (...) <- B[x].g();
  end-foreach

  foreach x in A
   B[x].h(p[x-1]);
  end-foreach

end-for
```

Charisma Semantics

- `foreach` statements execute across object arrays
  - Have associated methods
- Objects *produce* and *consume* parameters
- Statements executed on *individual* objects in *program order*

```
for I = 1 to MAX_ITER

  foreach x in A
   (p[x]) <- A[x].f();
  end-foreach

  foreach x in B
   (...) <- B[x].g();
  end-foreach

  foreach x in A
   B[x].h(p[x-1]);
  end-foreach

end-for
```

Data Dependences

- **`A::f()`** produces **`p[]`**

```
for I = 1 to MAX_ITER

  foreach x in A
   (p[x]) <- A[x].f();
  end-foreach

  foreach x in B
   (...) <- B[x].g();
  end-foreach

  foreach x in A
   B[x].h(p[x-1]);
  end-foreach

end-for
```

Data Dependence

Data Dependences

- **`A::f()`** produces **`p[]`**
- **`f()`** has embedded **`produce()`** function

Data Dependence

```
for I = 1 to MAX_ITER
  foreach x in A
   (p[x]) <- A[x].f();
  end-foreach

  foreach x in B
   (...) <- B[x].g();
  end-foreach

  foreach x in A
   B[x].h(p[x-1]);
  end-foreach
end-for
```

Data Dependences

- **`A::f()`** produces **`p[]`**
- **`f()`** has embedded **`produce()`** function
- **`B::h()`** consumes **`p[]`**

Data Dependence

```
for I = 1 to MAX_ITER

  foreach x in A
   (p[x]) <- A[x].f();
  end-foreach

  foreach x in B
   (...) <- B[x].g();
  end-foreach

  foreach x in A
   B[x].h(p[x-1]);
  end-foreach

end-for
```

Data Dependences

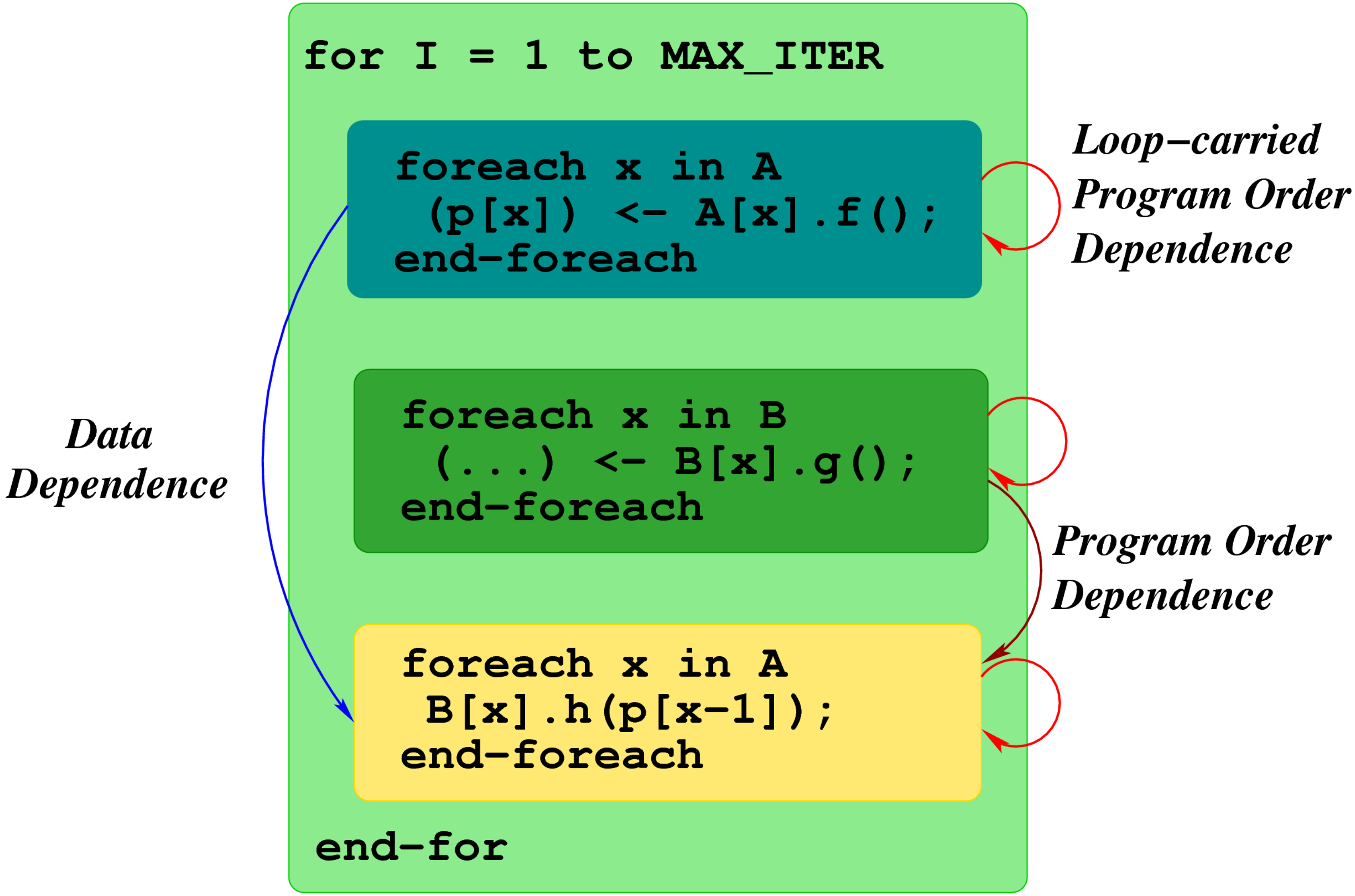
- **`A::f()`** produces **`p[]`**
- **`f()`** has embedded **`produce()`** function
- **`B::h()`** consumes **`p[]`**
- Indices decide dependences

Data Dependence

```
for I = 1 to MAX_ITER
  foreach x in A
   (p[x]) <- A[x].f();
  end-foreach

  foreach x in B
   (...) <- B[x].g();
  end-foreach

  foreach x in A
   B[x].h(p[x-1]);
  end-foreach
end-for
```

Program Order

- `B[x].g()` **executes before** `B[x].h()`
- But `B[x].g()` **concurrent with** `B[y].h()` if $x \neq y$



Ensuring Determinism

- Determinism = Data dependences +
  Program order

Ensuring Determinism

- Determinism = Data dependences +
  Program order
- **Data dependences** enforce causal order on statements *across* objects

Ensuring Determinism

- Determinism = Data dependences +
  Program order
- **Data dependences** enforce causal order on statements *across* objects
- **Program order** removes non-determinism within objects due to *message-reordering*

Implementing Semantics

```
for I = 1 to MAX_ITER

  foreach x in A
    (p[x]) <- A[x].f();
  end-foreach

  foreach x in B
    B[x].g(p[x]);
  end-foreach

  foreach x in C
    (p[x]) <- C[x].f();
  end-foreach

  foreach x in B
    B[x].g(p[x]);
  end-foreach

end-for
```

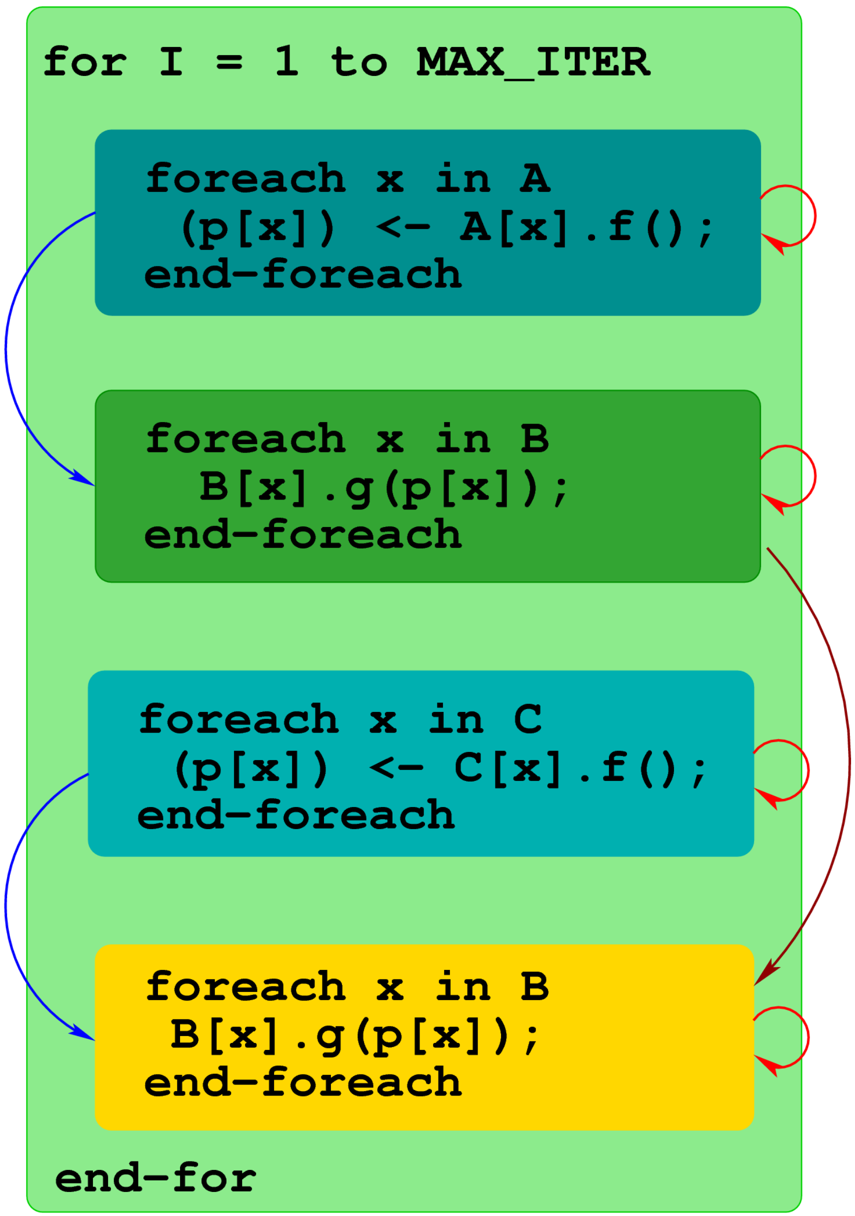
- Barrier after every **`for`** loop?

Implementing Semantics

```
for I = 1 to MAX_ITER

    foreach x in A
      (p[x]) <- A[x].f();
    end-foreach

    foreach x in B
       B[x].g(p[x]);
    end-foreach

    foreach x in C
      (p[x]) <- C[x].f();
    end-foreach

    foreach x in B
      B[x].g(p[x]);
    end-foreach

end-for
```

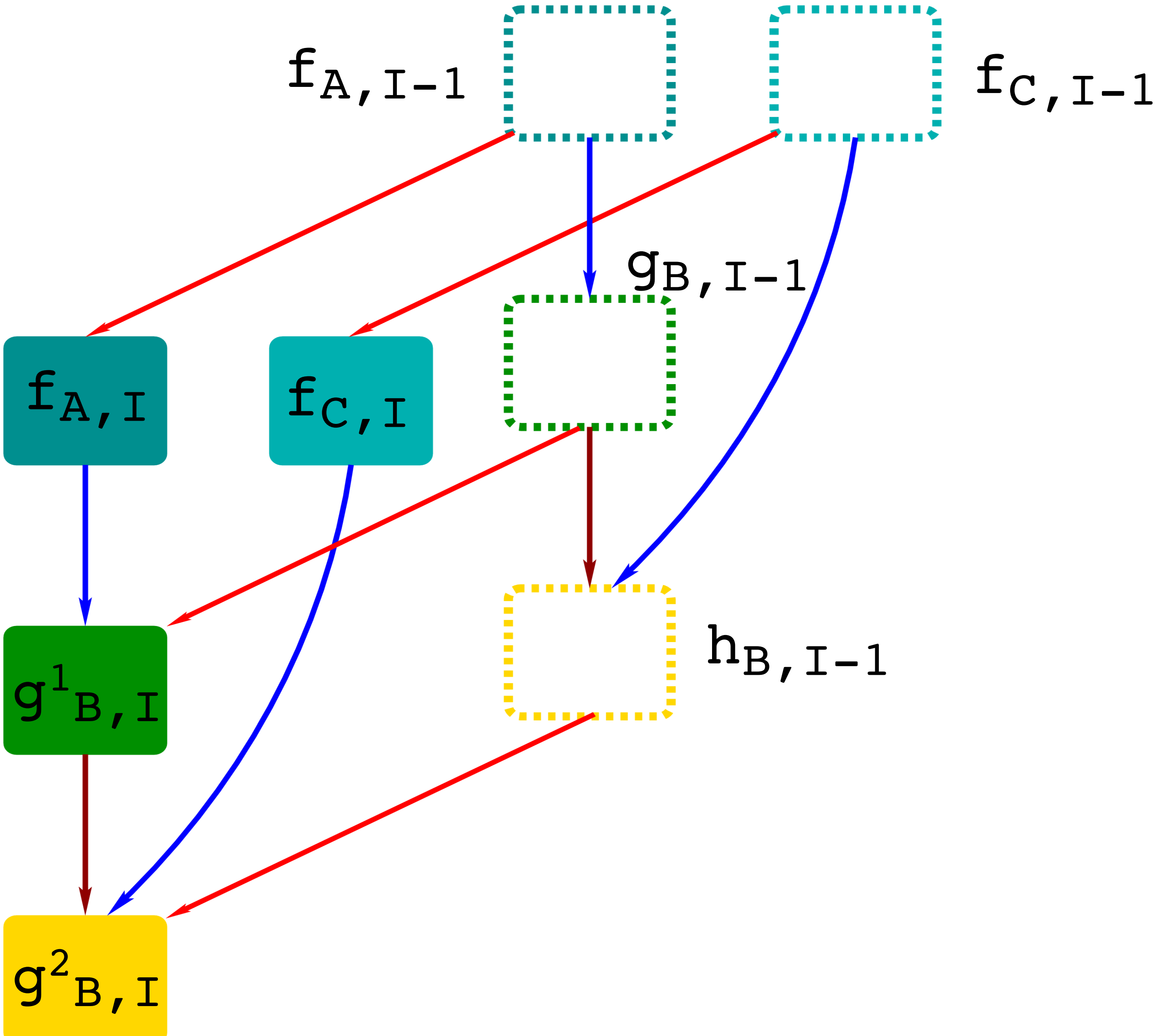
- Barrier after every **`for`** loop?
- Does it work here?

Implementing Semantics



- Barrier after every **`for`** loop?
- Does it work here?
- No, need barrier after each statement!
  - Too much parallel overhead

Programs are Distributed DAGS



Translation Strategy

- Use Charm++ for performance & productivity

Translation Strategy

- Use Charm++ for performance & productivity
- Translate Charisma's global control and data flows into local behavior of Charm++ objects

Translation Strategy

- Use Charm++ for performance & productivity
- Translate Charisma's global control and data flows into local behavior of Charm++ objects
- Instead of translating to Charm++ code, generate local DAGs specified in SDAG
  - Abstract target
  - Efficient implementation
  - Easier to write compiler

From Global to Local Flows (I)

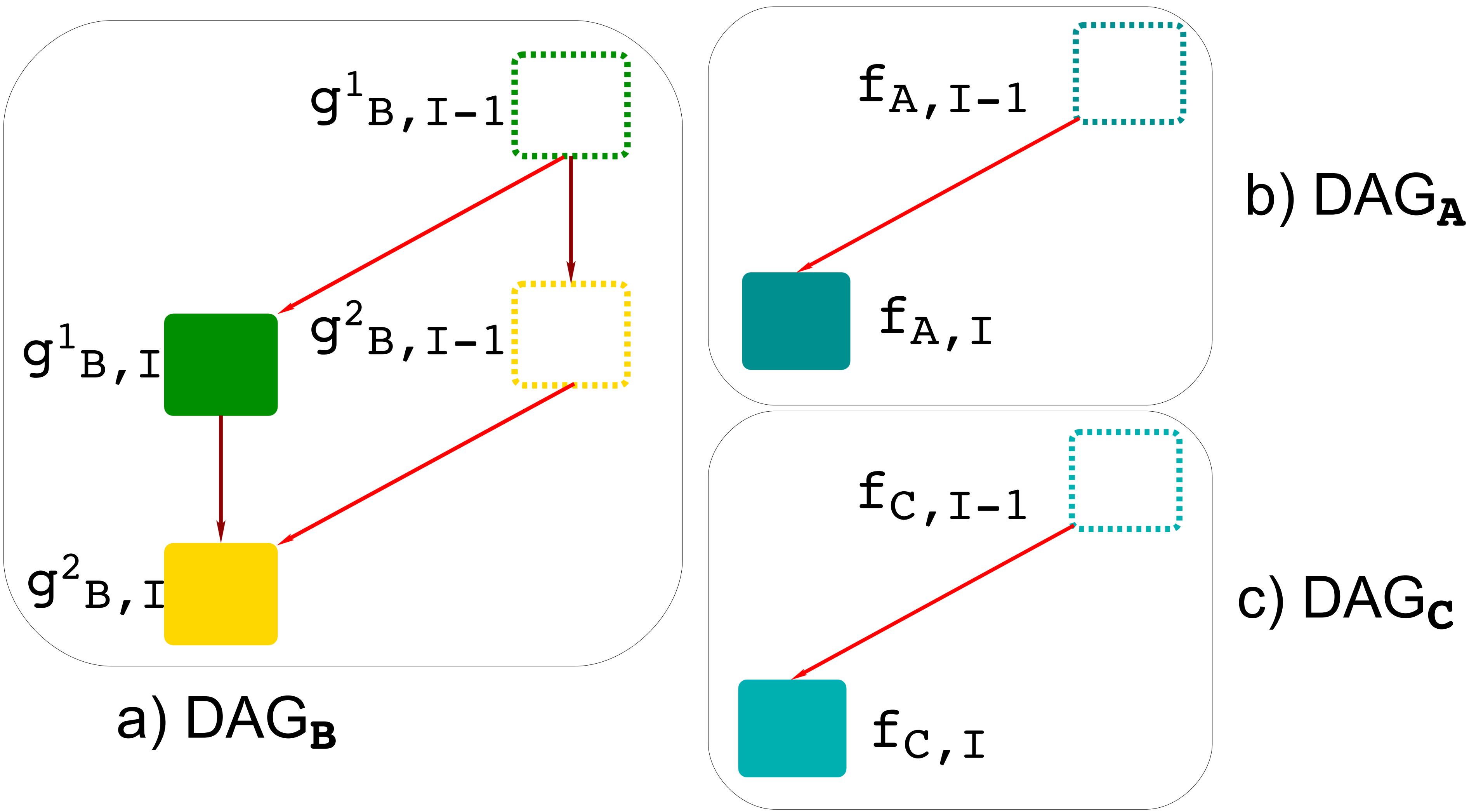
- Generate **unique targets**

From Global to Local Flows (I)

- Generate **unique targets**
- **Project** global control flow onto objects

From Global to Local Flows (I)

- Generate **unique targets**
- **Project** global control flow onto objects



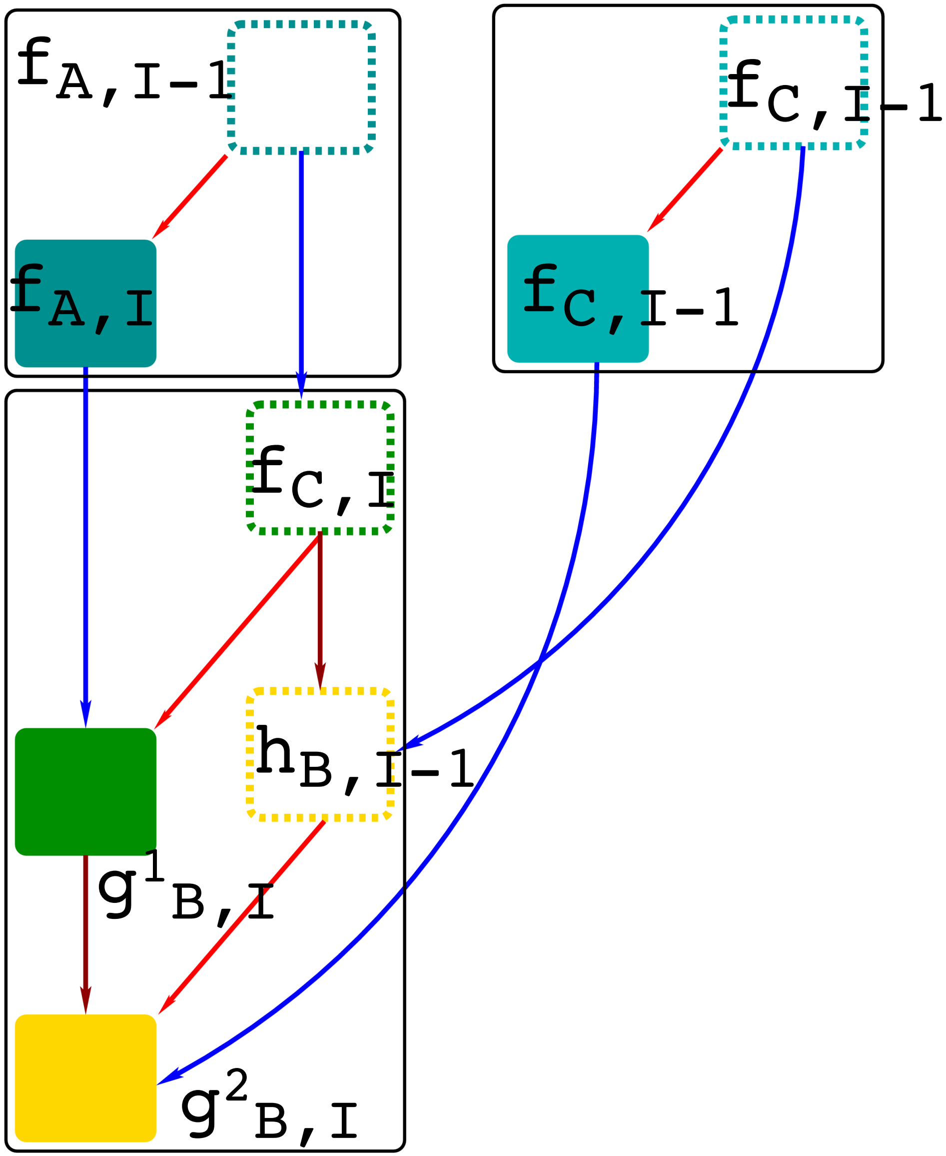
From Global to Local Flows (II)

- Generate asynch. message sends for **data dependences**

From Global to Local Flows (II)

- Generate asynch. message sends for **data dependences**
- Generated code sets reference numbers to ensure match between sender and receiver iterations

From Global to Local Flows (II)



- Generate asynch. message sends for **data dependences**
- Generated code sets reference numbers to ensure match between sender and receiver iterations

Performance Comparisons

- Compare code generated by previous and new versions of Charisma compiler
  - CTC: Charisma to Charm++
  - CTS: Charisma to SDAG

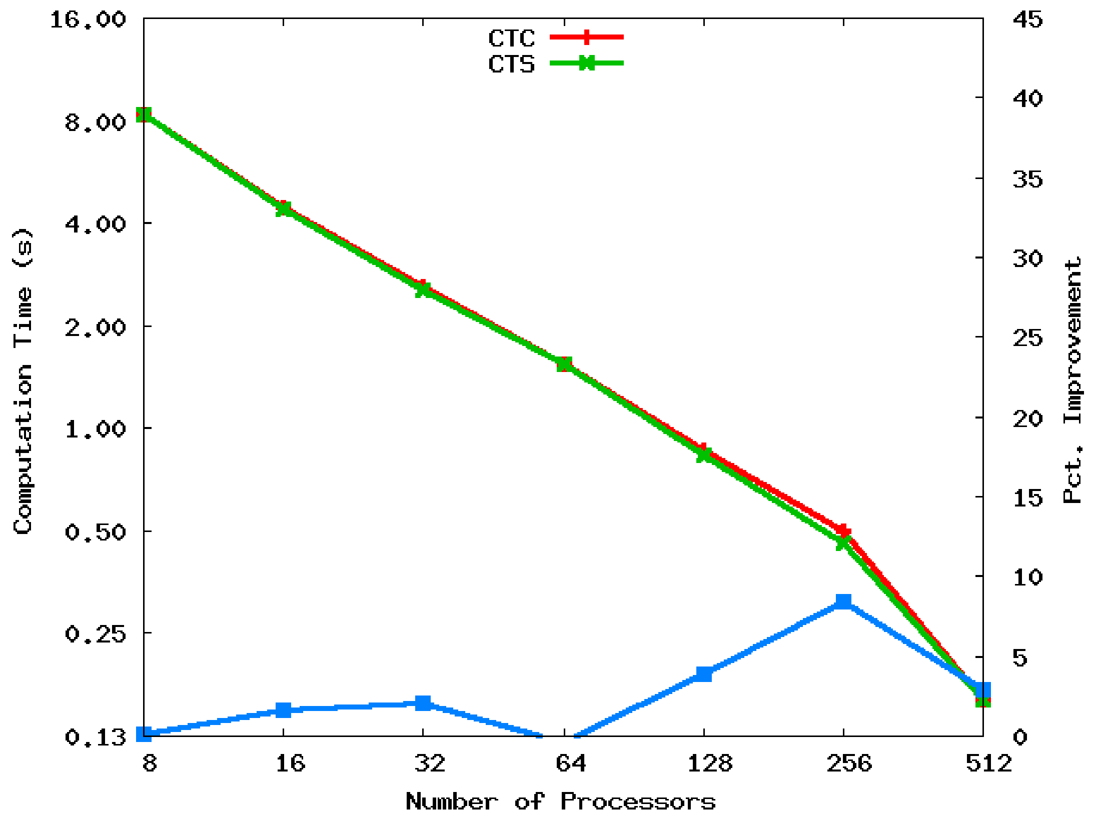
Performance Comparisons

- Compare code generated by previous and new versions of Charisma compiler
  - CTC: Charisma to Charm++
  - CTS: Charisma to SDAG
- CTS eliminates barriers at end of **`for`** loops

Performance Comparisons

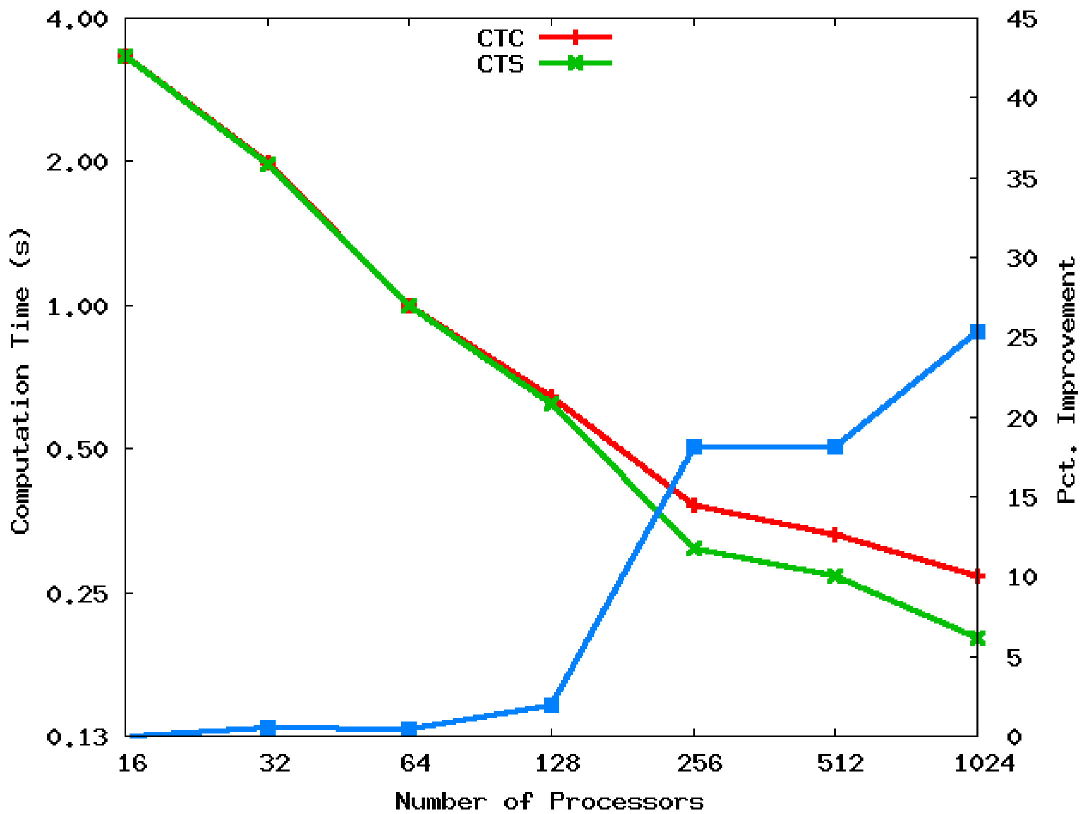
- Compare code generated by previous and new versions of Charisma compiler
  - CTC: Charisma to Charm++
  - CTS: Charisma to SDAG
- CTS eliminates barriers at end of **`for`** loops
- Similar CTC implementation would have required significantly more construction effort

3D FFT



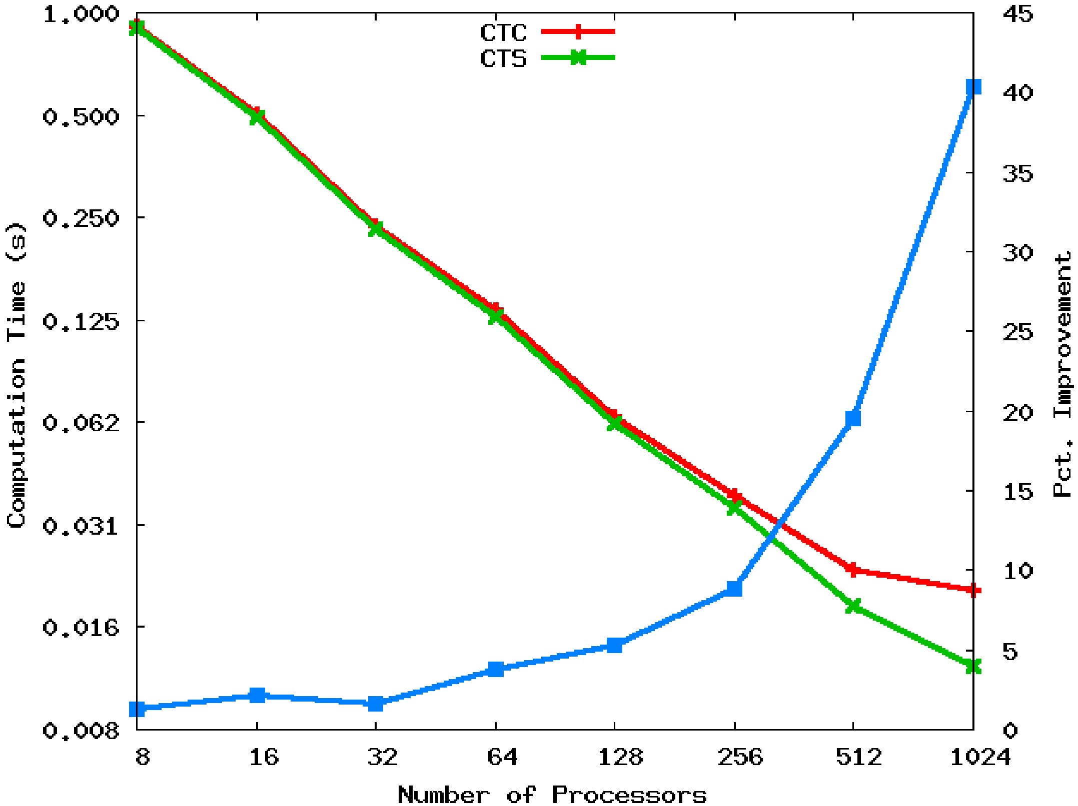

```
foreach x in planes1
   (pencildata[x,*]) <- planes1[x].fft1d();
end-foreach
foreach y in planes2
   planes2[y].fft2d(pencildata[*,y]);
end-foreach
```

Cannon Matrix Multiplication




```
for I = 1 to (N/T)
  foreach x,y in M
    (A[x,y], B[x,y]) <- M[x,y].prodTiles();
    workers[x,y].mult(A[x+1, y], B[x, y+1]);
  end-foreach
end-for
```

Five-Point Jacobi Relaxation




```
for I = 1 to 100
 foreach i,j in J
  (lb[i,j],rb[i,j],tb[i,j],bb[i,j]) ← J[i,j].prodBorders();
  J[i,j].compute(lb[i+1,j],rb[i-1,j],tb[i,j-1],bb[i,j+1]);
 end-foreach
end-for
```

Conclusion

- Benefits of translating Charisma to SDAG
  - Less impedance mismatch
    - Compiler easier to write
  - Existing dependence satisfaction, loop tagging frameworks
  - Performance gain (!)