Kanor

An Embedded DSL for High Performance Declarative Communication

Nilesh Mahajan

Department of Computer Science
Indiana University, Bloomington.

November 6, 2015

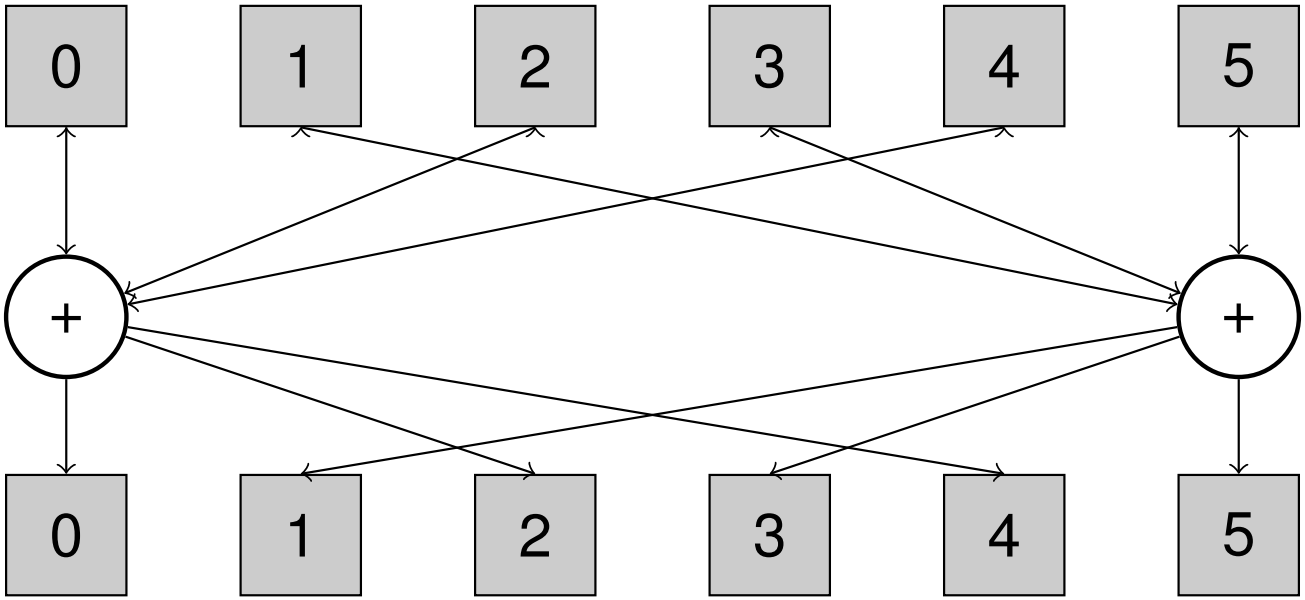
Motivation

- The Message Passing Interface (MPI) is popular.
- ... but has limitations.

Motivation

- The Message Passing Interface (MPI) is popular.
- ... but has limitations.

An Example



MPI

```
int rval;
std::vector<int> sbuff;
...
std::vector<MPI_Request> reqs;
int rmdr = me % 2;
for (int i = 0; i < nprocs; i++) {
  if (i % 2 == rmdr) {
    MPI_Request req;
    MPI_Isend(&sb[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD, &req);
    reqs.push_back(req);
  }
}
MPI_Waitall(reqs.size(), reqs.data(), MPI_STATUSES_IGNORE);
for (int i = 0; i < nprocs; i++) {
  if (i % 2 == rmdr) {
      int r;
      MPI_Recv(&r, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      rval += r;
  }
}
```

Kanor Equivalent

```
int rval;
kanor::CommBuff<int> sbuff;
...
Topology t;
rval _at_ rcvr << std::plus<int>() << sbuff[rcvr] _at_ sndr |
  _for_each(sndr, t.world) & _for_each(rcvr, t.world) &
  _if((sndr % 2) == (rcvr % 2)) _with t & GLOBAL;
```

Kanor Equivalent

```
int rval;
kanor::CommBuff<int> sbuff;
...
Topology t;
rval _at_ rcvr << std::plus<int>() << sbuff[rcvr] _at_ sndr |
 _for_each(sndr, t.world) & _for_each(rcvr, t.world) &
  _if((sndr % 2) == (rcvr % 2)) _with t & GLOBAL;
```

Kanor Equivalent

```
int rval;
kanor::CommBuff<int> sbuff;
...
Topology t;
rval _at_ rcvr << std::plus<int>() << sbuff[rcvr] _at_ sndr |
  _for_each(sndr, t.world) & _for_each(rcvr, t.world) &
_if((sndr % 2) == (rcvr % 2)) _with t & GLOBAL;
```

Kanor Equivalent

```
int rval;
kanor::CommBuff<int> sbuff;
...
Topology t;
rval _at_ rcvr << std::plus<int>() << sbuff[rcvr] _at_ sndr
  _for_each(sndr, t.world) & _for_each(rcvr, t.world) &
  _if((sndr % 2) == (rcvr % 2)) _with t & GLOBAL;
```

Kanor Equivalent

```
int rval;
kanor::CommBuff<int> sbuff;
...
Topology t;
rval _at_ rcvr << std::plus<int>() << sbuff[rcvr] _at_ sndr |
  _for_each(sndr, t.world) & _for_each(rcvr, t.world) &
  _if((sndr % 2) == (rcvr % 2)) _with t & GLOBAL;
```

Kanor Equivalent

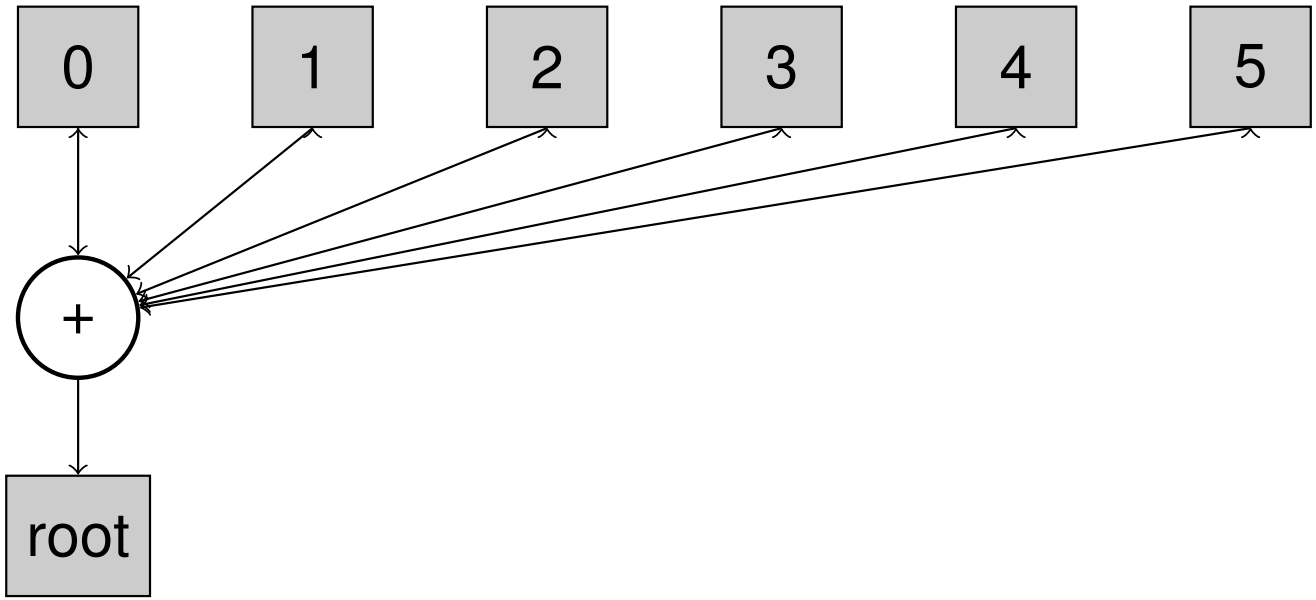
```
int rval;
kanor::CommBuff<int> sbuff;
...
Topology t;
rval _at_ rcvr << std::plus<int>() << sbuff[rcvr] _at_ sndr |
  _for_each(sndr, t.world) & _for_each(rcvr, t.world) &
  _if((sndr % 2) == (rcvr % 2)) _with t & GLOBAL;
```

Kanor Equivalent

```
int rval;
kanor::CommBuff<int> sbuff;
...
Topology t;
rval _at_ rcvr << std::plus<int>() << sbuff[rcvr] _at_ sndr |
  _for_each(sndr, t.world) & _for_each(rcvr, t.world) &
  _if((sndr % 2) == (rcvr % 2)) _with t & GLOBAL
```

Encoding MPI Collectives

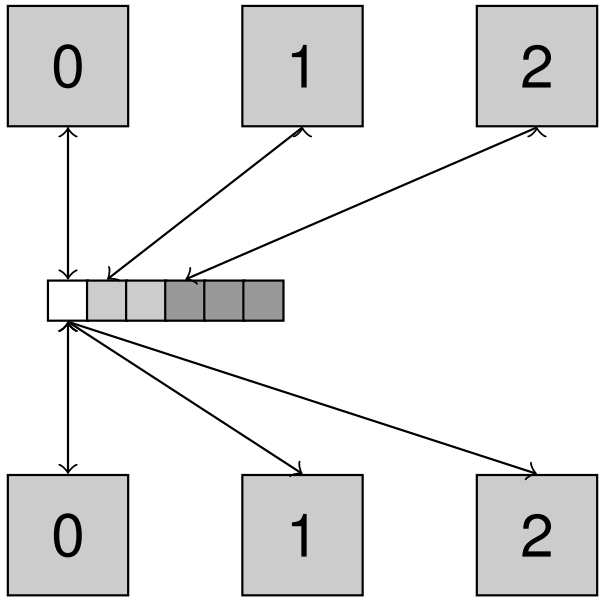
Reduce




```
int root = ...;
rb _at_ root << kanor::sum<btype> << sb _at_ s |
  _for_each(s, t.world) _with t & GLOBAL;
```

Encoding MPI Collectives

Allgatherv

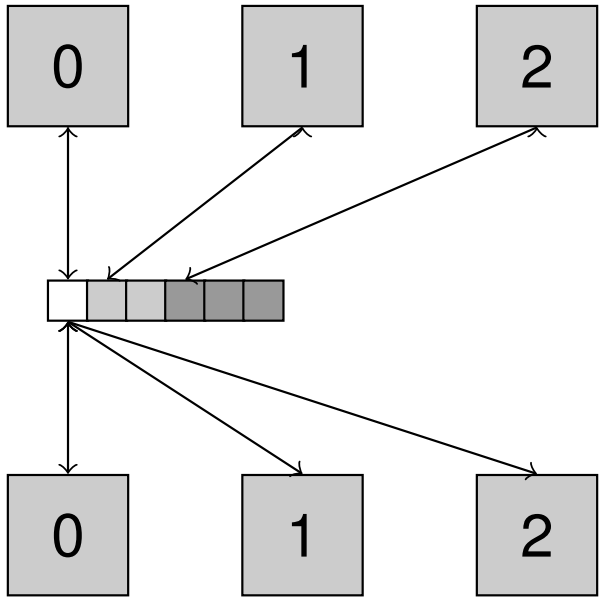



```
rb[Slice(dspls[s],counts[s])] _at_ r <<
sb[Slice(0, counts[s])] _at_ s |
  _for_each(s, t.world) & _for_each(r, t.world) _with t & GLOBAL;
```

```
  rb[Slice(dspls[s],counts[s])] _at_ r << sb[Slice(0, counts[s])] _at_
      s
  | _for_each(s, t.world) & _for_each(r, t.world) _with t & GLOBAL;
```

Encoding MPI Collectives

Allgatherv




```
rb[Slice(dspls[s],counts[s])] _at_ r <<
sb[Slice(0, counts[s])] _at_ s |
  _for_each(s, t.world) & _for_each(r, t.world) _with t & GLOBAL;
```

```
  rb[ Slice(dspls[s],counts[s]) _at_ r << sb[ Slice(0, counts[s]) _at_
      s
  | _for_each(s, t.world) & _for_each(r, t.world) _with t & GLOBAL;
```

Our Approach

- Similar to MPI
- Bulk Synchronous Parallel(BSP) style communication, no send-receives
- Separate communication code from computation
- Embedded in C++
- Powerful enough to express complex communication patterns

Our Approach

- Similar to MPI
- Bulk Synchronous Parallel(BSP) style communication, no send-receives
- Separate communication code from computation
- Embedded in C++
- Powerful enough to express complex communication patterns

Our Approach

- Similar to MPI
- Bulk Synchronous Parallel(BSP) style communication, no send-receives
- Separate communication code from computation
- Embedded in C++
- Powerful enough to express complex communication patterns

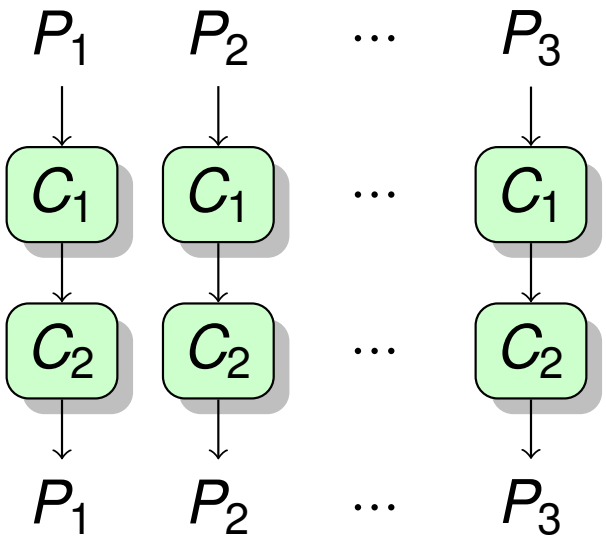
Our Approach

- Similar to MPI
- Bulk Synchronous Parallel(BSP) style communication, no send-receives
- Separate communication code from computation
- Embedded in C++
- Powerful enough to express complex communication patterns

Our Approach

- Similar to MPI
- Bulk Synchronous Parallel(BSP) style communication, no send-receives
- Separate communication code from computation
- Embedded in C++
- Powerful enough to express complex communication patterns

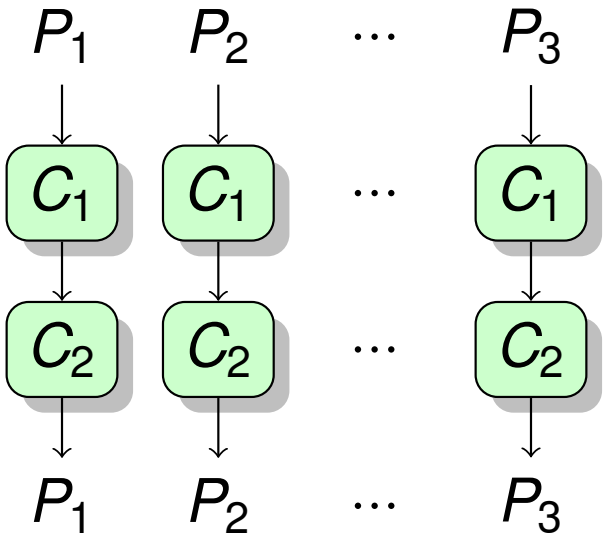
Kanor Execution



Questions

- Determinism - does it always produce the same results on the same inputs?
- Does it deadlock?

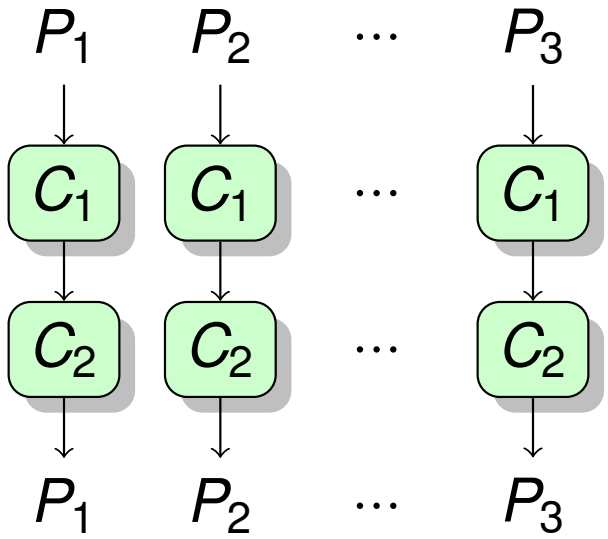
Well-formedness



We say a Kanor program is well-formed iff

- All processes participating in communication statement C, execute C,
- All processes participating in communication statements C_1 and C_2, execute C_1 and C_2 in the same order

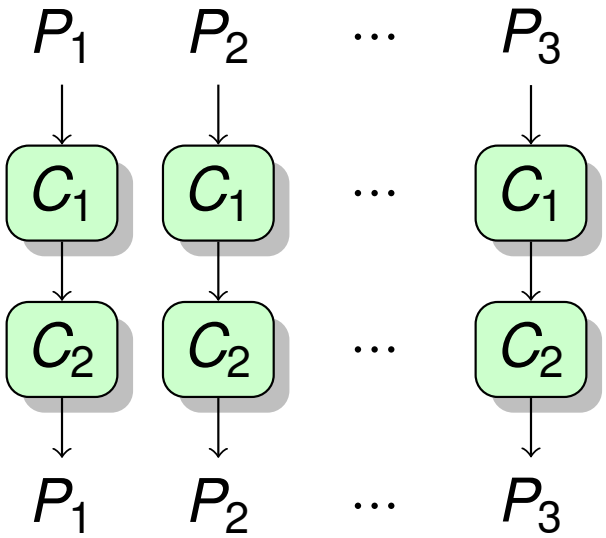
Well-formedness



We say a Kanor program is well-formed iff

- All processes participating in communication statement C, execute C,
- All processes participating in communication statements C_1 and C_2, execute C_1 and C_2 in the same order

Well-formedness



We say a Kanor program is well-formed iff

- All processes participating in communication statement C, execute C,
- All processes participating in communication statements C_1 and C_2, execute C_1 and C_2 in the same order

Properties

- Communication command always successful + well-formed Kanor programs = No deadlock.
- Operator application always deterministic + local computations deterministic = Deterministic behavior.
- `rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...`

Properties

- Communication command always successful + well-formed Kanor programs = No deadlock.
- Operator application always deterministic + local computations deterministic = Deterministic behavior.
- `rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...`

Properties

- Communication command always successful + well-formed Kanor programs = No deadlock.
- Operator application always deterministic + local computations deterministic = Deterministic behavior.
- `rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...`

Properties

- Communication command always successful + well-formed Kanor programs = No deadlock.
- Operator application always deterministic + local computations deterministic = Deterministic behavior.
- `rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...`

Properties

- Communication command always successful + well-formed Kanor programs = No deadlock.
- Operator application always deterministic + local computations deterministic = Deterministic behavior.
- `rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...`

Properties

- Communication command always successful + well-formed Kanor programs = No deadlock.
- Operator application always deterministic + local computations deterministic = Deterministic behavior.
- `rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...`

Collective Detection

- Can we do better at runtime?
- Maybe identify common pattern and call a well-known collective?
- A process can look at its sender and receiver sets.
- Do the other processes agree?

Collective Detection

- Can we do better at runtime?
- Maybe identify common pattern and call a well-known collective?
- A process can look at its sender and receiver sets.
- Do the other processes agree?

Collective Detection

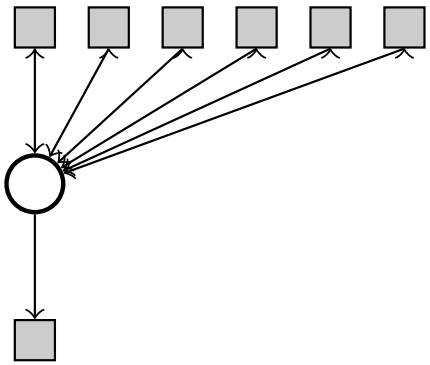
- Can we do better at runtime?
- Maybe identify common pattern and call a well-known collective?
- A process can look at its sender and receiver sets.
- Do the other processes agree?

Collective Detection

- Can we do better at runtime?
- Maybe identify common pattern and call a well-known collective?
- A process can look at its sender and receiver sets.
- Do the other processes agree?

Process Knowledge

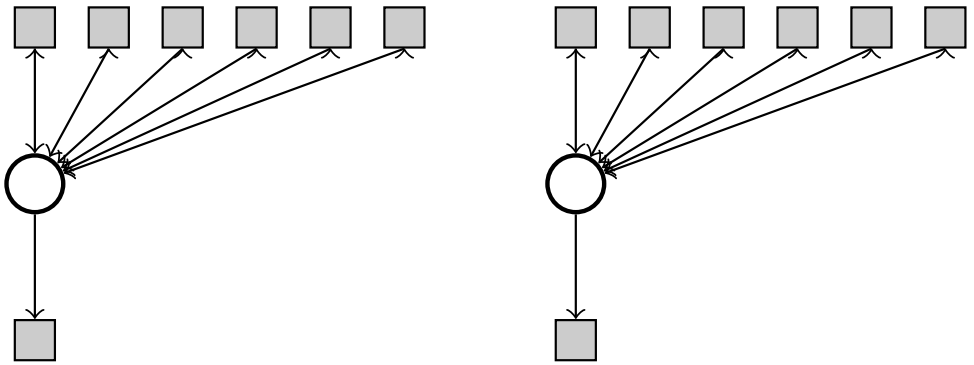
P_1



- Safe to enter Reduce!
- *Global Knowledge*

Process Knowledge

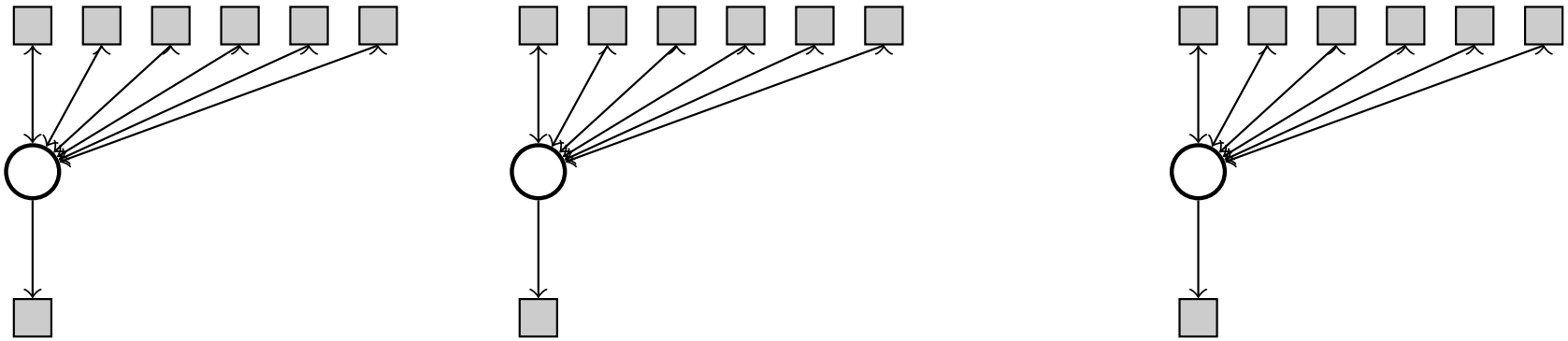
P_1 P_2



- Safe to enter Reduce!
- *Global Knowledge*

Process Knowledge

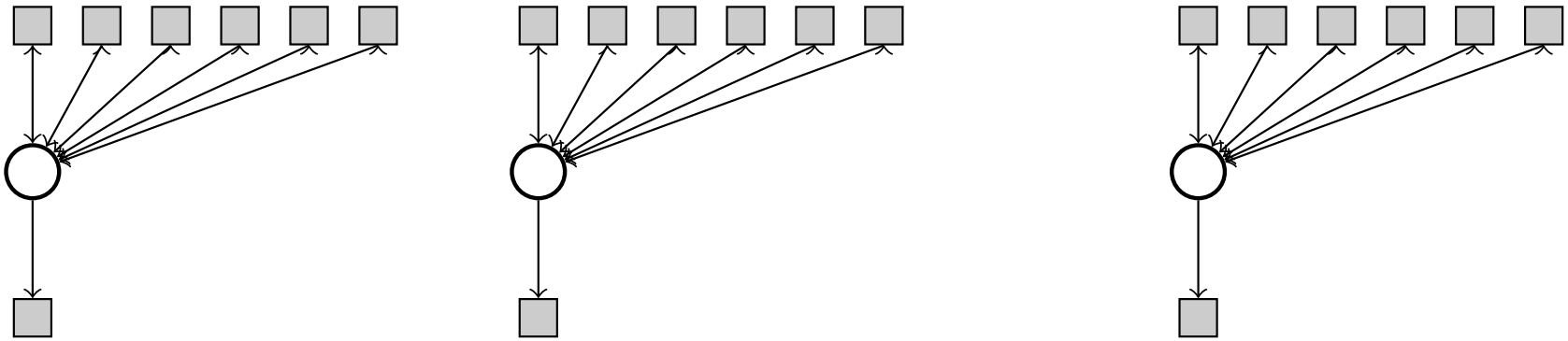
P_1 P_2 *and so on . . .*



- Safe to enter Reduce!
- *Global Knowledge*

Process Knowledge

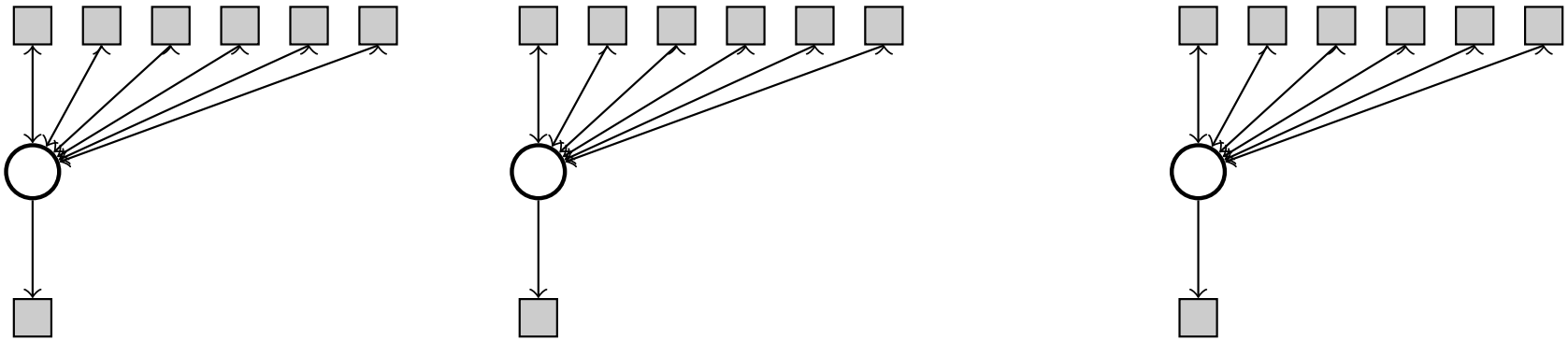
P_1 P_2 *and so on . . .*



- Safe to enter Reduce!
- *Global Knowledge*

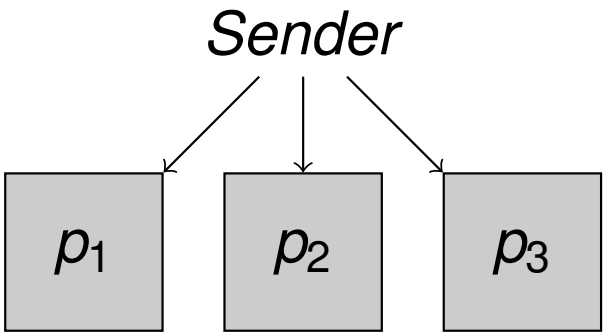
Process Knowledge

P_1 P_2 *and so on . . .*

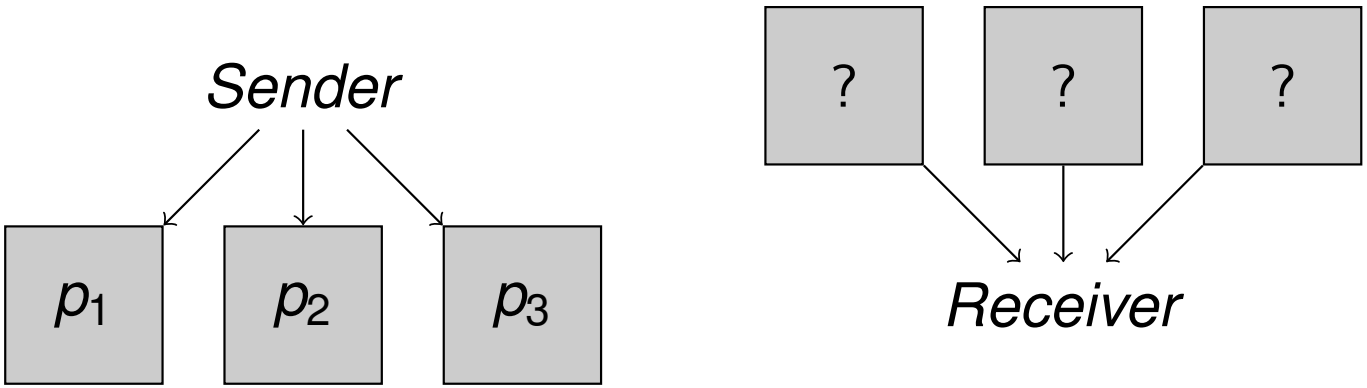


- Safe to enter Reduce!
- *Global Knowledge*

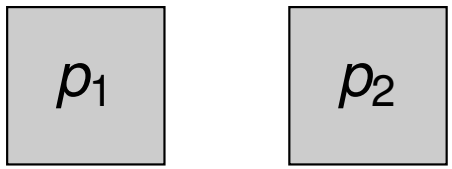
Sender Knowledge



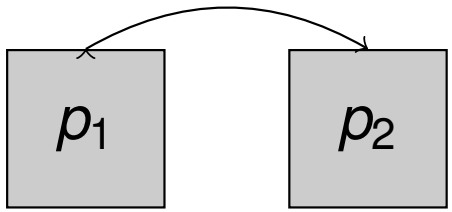
Sender Knowledge



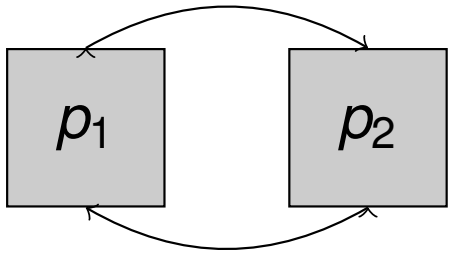
Corresponding Knowledge



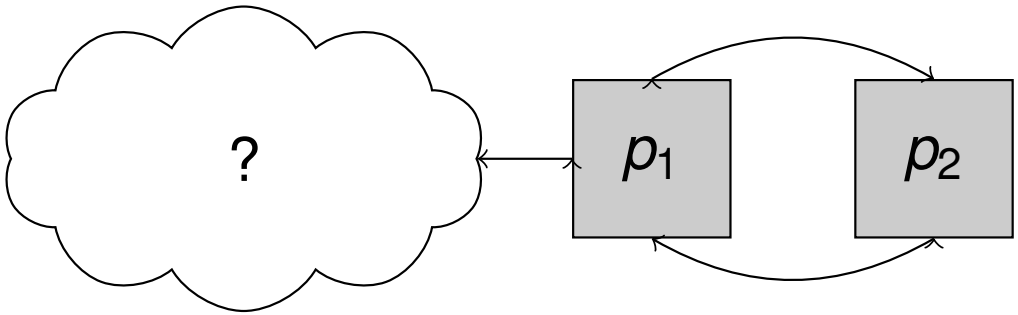
Corresponding Knowledge



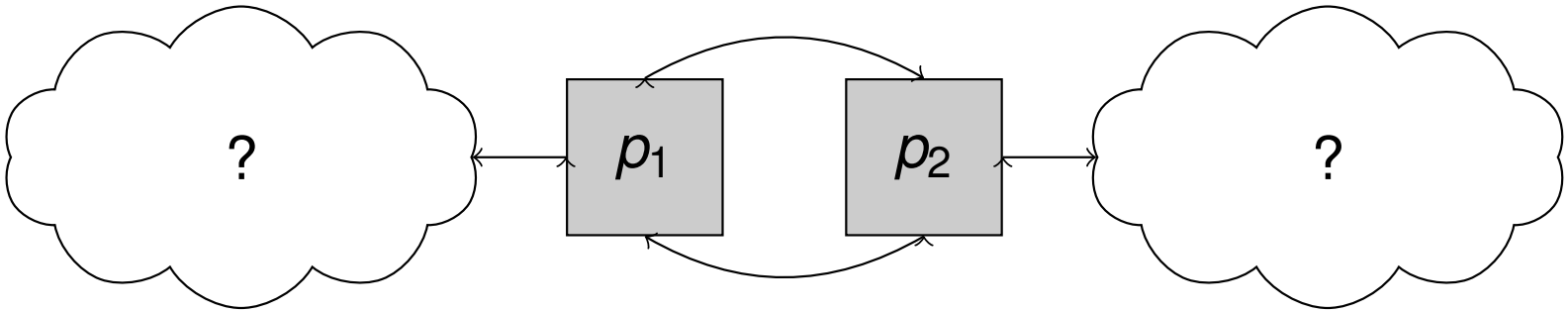
Corresponding Knowledge



Corresponding Knowledge



Corresponding Knowledge



Communication Knowledge

- *Global* - All processes know about other processes
- *Sender* - Only the senders know about the receivers.
- *Corresponding* - Only the sender and receiver know about each other

Communication Knowledge

- *Global* - All processes know about other processes
- *Sender* - Only the senders know about the receivers.
- *Corresponding* - Only the sender and receiver know about each other

Communication Knowledge

- *Global* - All processes know about other processes
- *Sender* - Only the senders know about the receivers.
- *Corresponding* - Only the sender and receiver know about each other

Algorithm

```
Input: Communication Statement S
Output: Set of Collective Calls C
// G is a directed graph, in which vertices are process IDs,
// an edge connects sender to a receiver
G = build from S;
n = number of vertices in vertex set V(G);
if each vertex v in V(G) has degree n then
    if send and receiver buffers contiguous then
        C = {Alltoall};
    else
        C = {AllGather};
    end
    return;
end
// build rooted collectives to be executed independently
foreach v in V(G) with no incoming edges do
    if send and receiver buffers contiguous then
        C = C ∪ {broadcast};
    else
        C = C ∪ {scatter};
    end
end
```

Algorithm 1: Algorithm to detect MPI collectives.

Caching

- Communication knowledge is across processes.
- To reuse, runtime needs to know the pattern does not change across statement instances.
- Invariant characteristics:
  - message length
  - process sets
  - message contiguity

Caching

- Communication knowledge is across processes.
- To reuse, runtime needs to know the pattern does not change across statement instances.
- Invariant characteristics:
  - message length
  - process sets
  - message contiguity

Caching

- Communication knowledge is across processes.
- To reuse, runtime needs to know the pattern does not change across statement instances.
- Invariant characteristics:
  - message length
  - process sets
  - message contiguity

Caching

- Communication knowledge is across processes.
- To reuse, runtime needs to know the pattern does not change across statement instances.
- Invariant characteristics:
  - message length
  - process sets
  - message contiguity

Caching

- Communication knowledge is across processes.
- To reuse, runtime needs to know the pattern does not change across statement instances.
- Invariant characteristics:
  - message length
  - process sets
  - message contiguity

Caching

- Communication knowledge is across processes.
- To reuse, runtime needs to know the pattern does not change across statement instances.
- Invariant characteristics:
  - message length
  - process sets
  - message contiguity

Implementation

- C++ operator overloading
- C++11 meta-programming for certain type checks

```
rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...
```

- Runtime uses MPI underneath
- No compiler support yet

Implementation

- C++ operator overloading
- C++11 meta-programming for certain type checks

```
rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...
```

- Runtime uses MPI underneath
- No compiler support yet

Implementation

- C++ operator overloading
- C++11 meta-programming for certain type checks

```
rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...
```

- Runtime uses MPI underneath
- No compiler support yet

Implementation

- C++ operator overloading
- C++11 meta-programming for certain type checks

```
rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...
```

- Runtime uses MPI underneath
- No compiler support yet

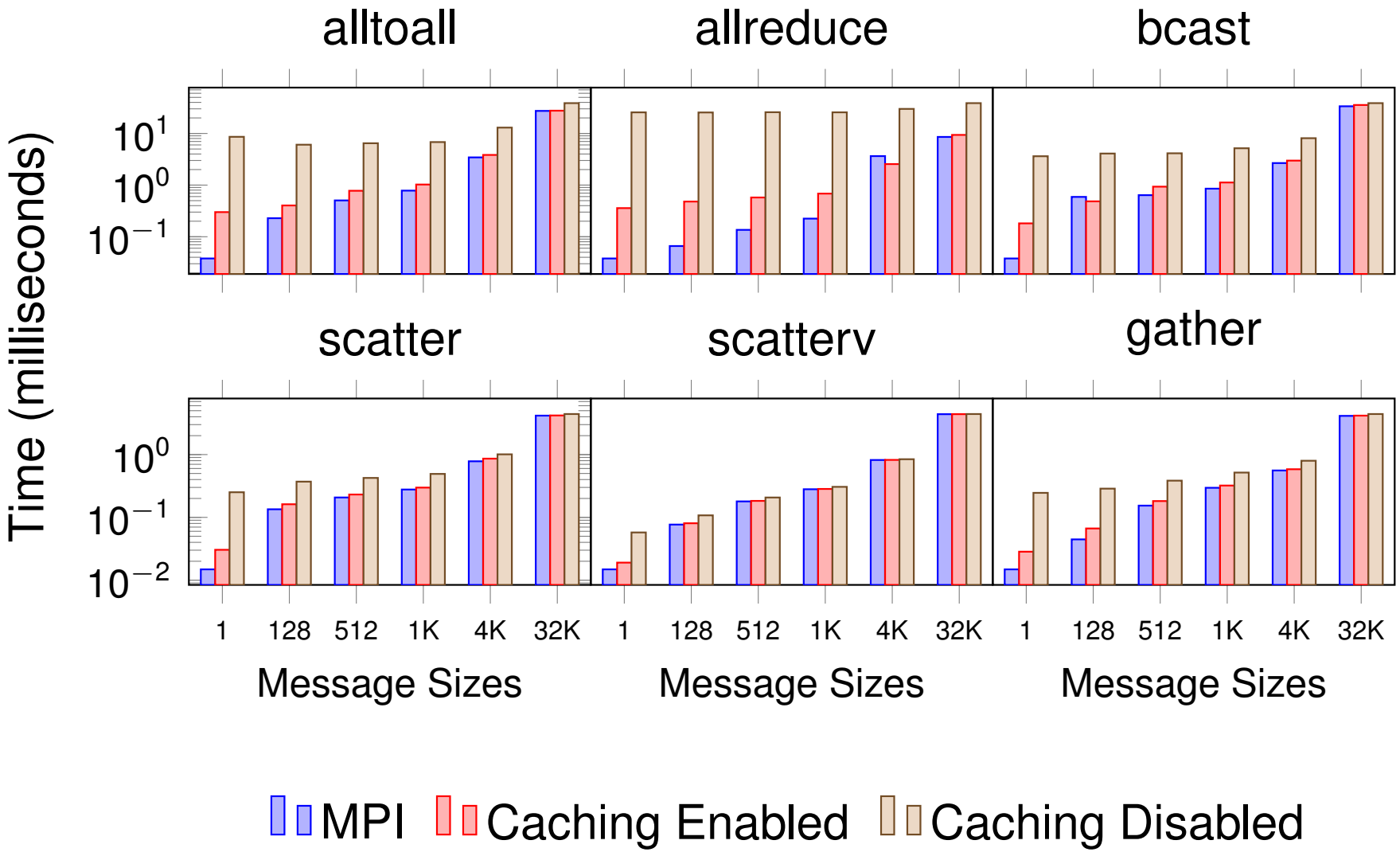
Implementation

- C++ operator overloading
- C++11 meta-programming for certain type checks

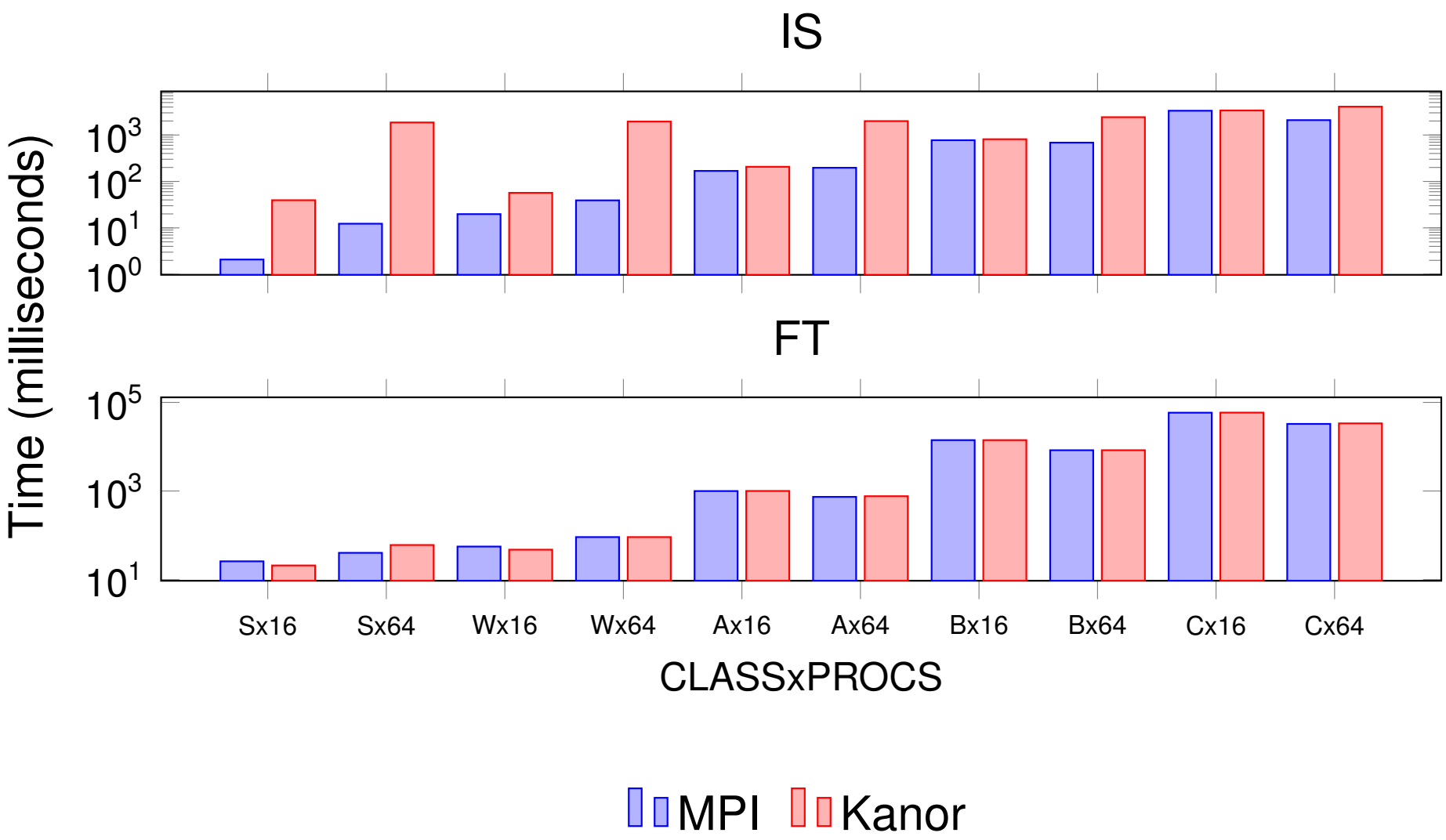
```
rval _at_ rcvr << std::plus<int>()<< sbuff[rcvr] _at_ sndr | ...
```

- Runtime uses MPI underneath
- No compiler support yet

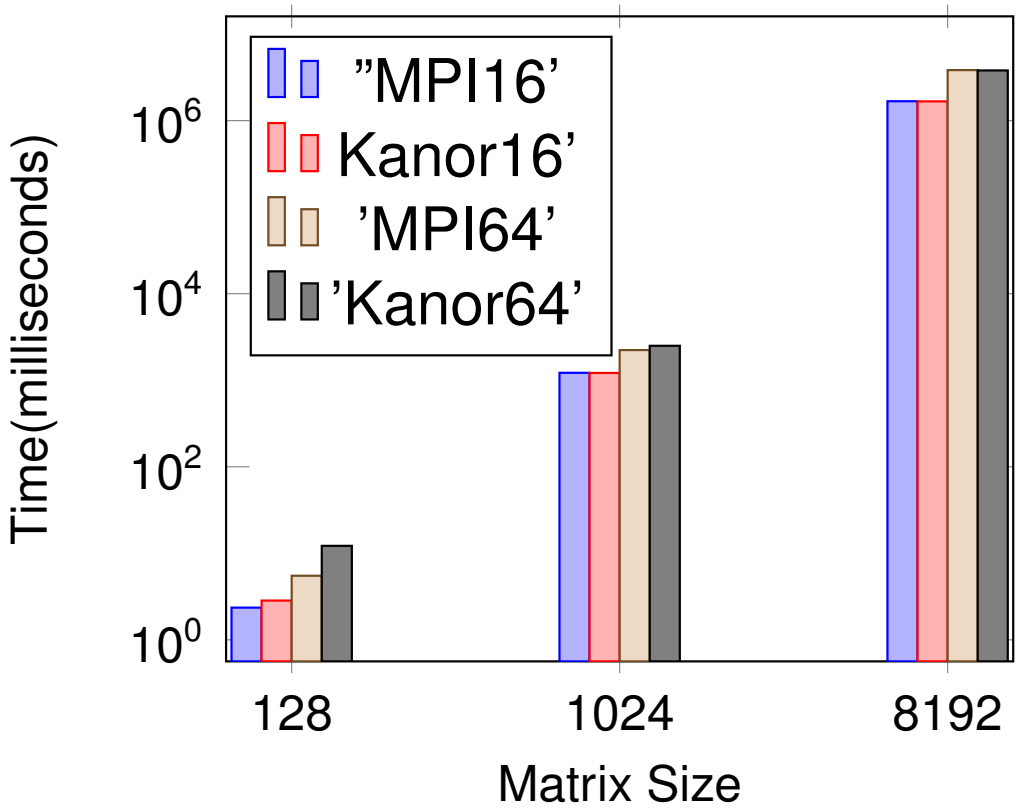
MPI collectives



NAS Benchmarks



Dense Cholesky Kernel



Async Window Expansion

- Use asynchronous calls to start communication early and overlap it with computation

```
// sb is sent buffer
// rb is received buffer
... = rb;
sb = ...;
...;
for (i=0; i < N; i++) {
  rb _at_ r << sb _at_ s | ...;
  ... = rb;
  // compute
  sb = ...;
}
```

```
... = rb;
sb = ...;
start_send(sb, ...);
...;
for (i=0; i < N; i++) {
  wait(...);
  ... = rb;
  // compute
  sb = ...;
  start_send(sb, ...);
}
```

Strip Mining

- Adjust the granularity of communication with computation that produces or uses the communicated data.

```
... = rb;
sb = ...;
start_send(sb, ...);
for (i=1; i < min(B,N); i+=B) {
  for (int b = 0; b < B; b++) {
     ... = rb;
     sb = ...;
     ...
  }
  start_send(sb, B, ...);
}
```

Thank You

Questions?

- Kanor allows declarative specification of communication.
- Kanor helps programmers express complex communication patterns.
- Kanor programs are SPMD written in BSP style.
- Kanor makes reasoning easier for compiler, programmers.