Partial Globalization of Partitioned Address Spaces for Zero-copy Communication with Shared Memory

Fangzhou Jiao, Nilesh Mahajan, Jeremiah Willcock,
Arun Chauhan, Andrew Lumsdaine
Indiana University

HiPC 2011

Motivation

- Increasing popularity and availability of many-cores
- Abundance of legacy MPI code
- Simplifying programming model
  - single model, instead of hybrid
- Leveraging shared memory fully for performance
- Proving that shared memory could be used as an optimization for communication

Partitioned Address Space Programming on Shared Memory

- Avoids having to worry about race conditions
- Encourages programmers to think about locality
- Could make it easier to reason about program correctness
    - if done at the right level of abstraction

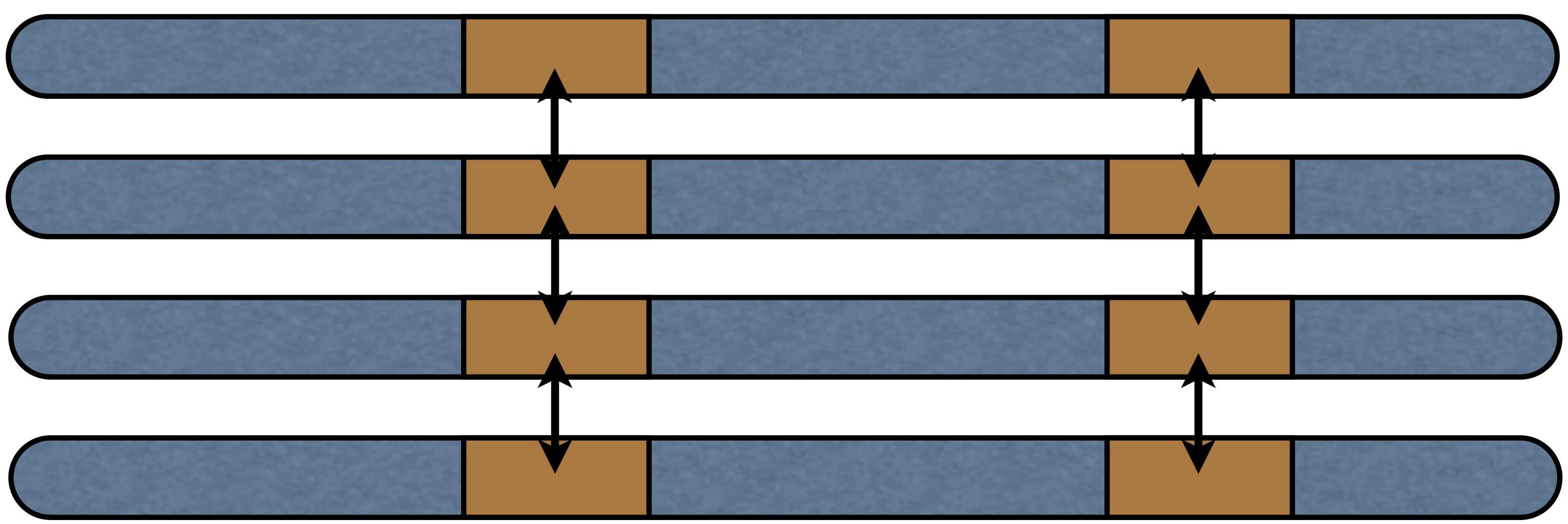
Needs special handling to compete in performance with threaded shared memory programs

Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
  - alternate between computation and communication
  - communication optimization breaks the structure

Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
  - alternate between computation and communication
  - communication optimization breaks the structure



Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
  - alternate between computation and communication
  - communication optimization breaks the structure

Declarative Approach

- Originally motivated by Block-synchronous Parallel (BSP) programs, especially for collective communication
    - alternate between computation and communication
    - communication optimization breaks the structure

- Extend to non BSP-style applications

Kanor for Clusters

$$\textbf{@communicate} \; \{ \; b@recv\_rank <<= a@send\_rank \; \}$$

Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. *Kanor: A Declarative Language for Explicit Communication. In Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL), 2011. Held in conjunction with the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).*

Kanor for Clusters

$$\textbf{@communicate} \{ b@recv\_rank <<= a@send\_rank \}$$

$$e_0@e_1 << op << e_2@e_3 \texttt{ where } e_4$$

Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. *Kanor: A Declarative Language for Explicit Communication. In Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL), 2011. Held in conjunction with the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).*

Kanor for Clusters

$$\textbf{@communicate} \{ b@recv\_rank <<= a@send\_rank \}$$

$$e_0@e_1 << op << e_2@e_3 \;\texttt{where}\; e_4$$

$$e_0@e_1 <<= e_2@e_3 \;\texttt{where}\; e_4$$

Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. *Kanor: A Declarative Language for Explicit Communication. In Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL), 2011. Held in conjunction with the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).*

Kanor for Clusters

$$\textbf{@communicate} \{ b@recv\_rank <<= a@send\_rank \}$$

$$e_0@e_1 << op << e_2@e_3 \texttt{ where } e_4$$

$$e_0@e_1 <<= e_2@e_3 \texttt{ where } e_4$$

$$\underbrace{\texttt{A[j]}}_{\text{storage location}} @ \underbrace{\texttt{i}}_{\text{receiver rank}} \underbrace{<<=}_{\text{reduction operator}} \underbrace{\texttt{B[i]}}_{\text{data}} @ \underbrace{\texttt{j}}_{\text{sender rank}} \texttt{ where } \underbrace{\texttt{i in world}}_{\text{generator}}, \underbrace{\texttt{j in } \{\texttt{0} \ldots \texttt{i}\}}_{\text{generator}}, \underbrace{\texttt{i} \% 2 == \texttt{0}}_{\text{filter}}$$

Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. *Kanor: A Declarative Language for Explicit Communication. In Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL), 2011. Held in conjunction with the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).*

Kanor for Clusters

$$\textbf{@communicate}\ \{\ b@recv\_rank <<= a@send\_rank\ \}$$

$$e_0@e_1 << op << e_2@e_3\ \texttt{where}\ e_4$$

$$e_0@e_1 <<= e_2@e_3\ \texttt{where}\ e_4$$

`A[j] @ i <<= B[i] @ j where i in world, j in {0...i}, i % 2 == 0`

- `A[j]` — *storage location*
- `i` — *receiver rank*
- `<<=` — *reduction operator*
- `B[i]` — *data*
- `j` — *sender rank*
- `i in world` — *generator*
- `j in {0...i}` — *generator*
- `i % 2 == 0` — *filter*

↓

Source-level compiler (using ROSE)

↓

standard C++ code

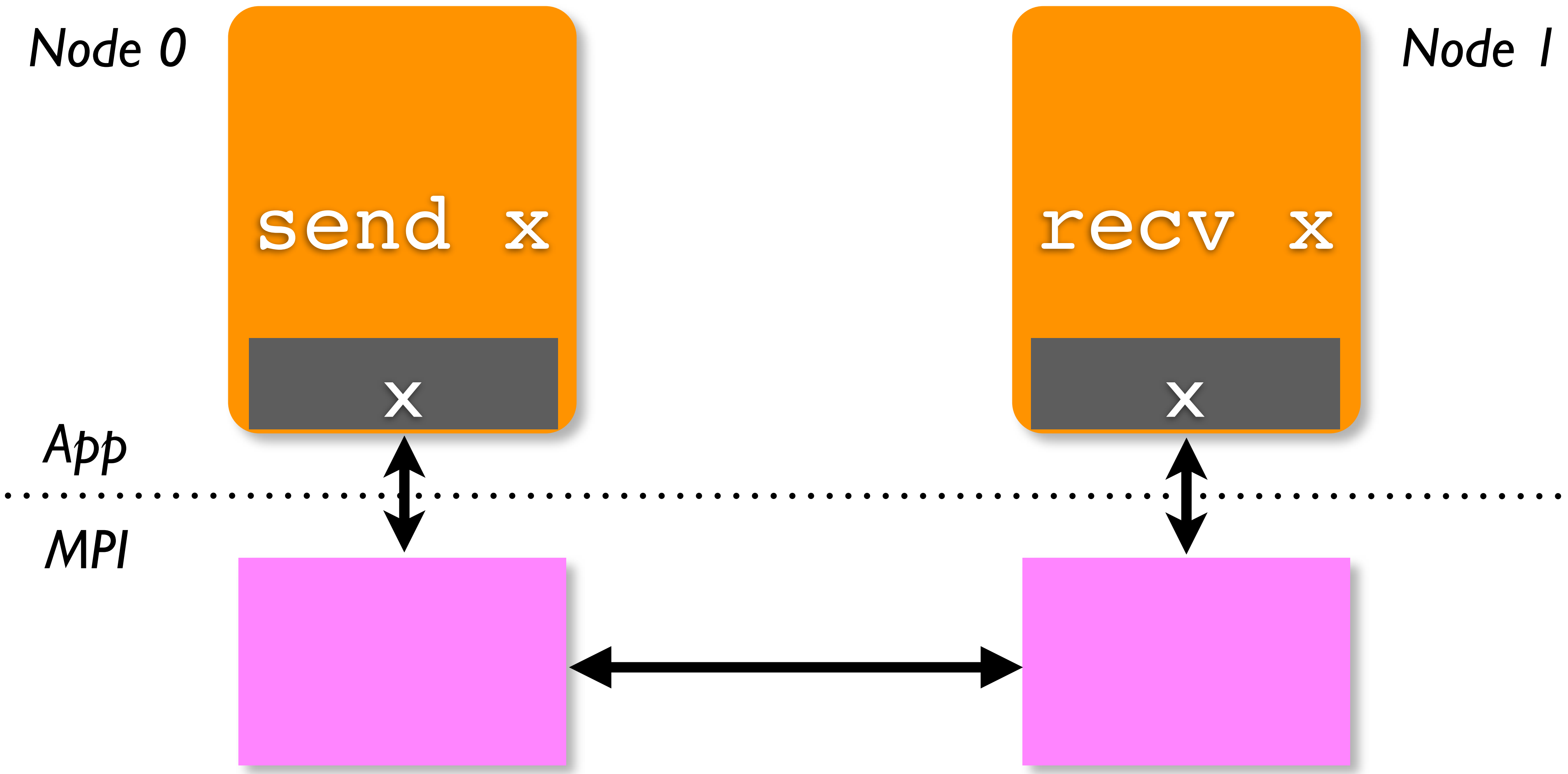
Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. *Kanor: A Declarative Language for Explicit Communication. In Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL), 2011. Held in conjunction with the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).*

Design Principles

- Users must think in parallel (creativity)
  - but not be encumbered with optimizations that can be automated, or proving synchronization correctness
- Compiler focuses on what it can do (mechanics)
  - not creative tasks, such as determining data distributions, or creating new parallel algorithms
- Incremental deployment
  - not a new programming language
  - more of a *coordination language* (DSL)
- Formal semantics
  - provable correctness

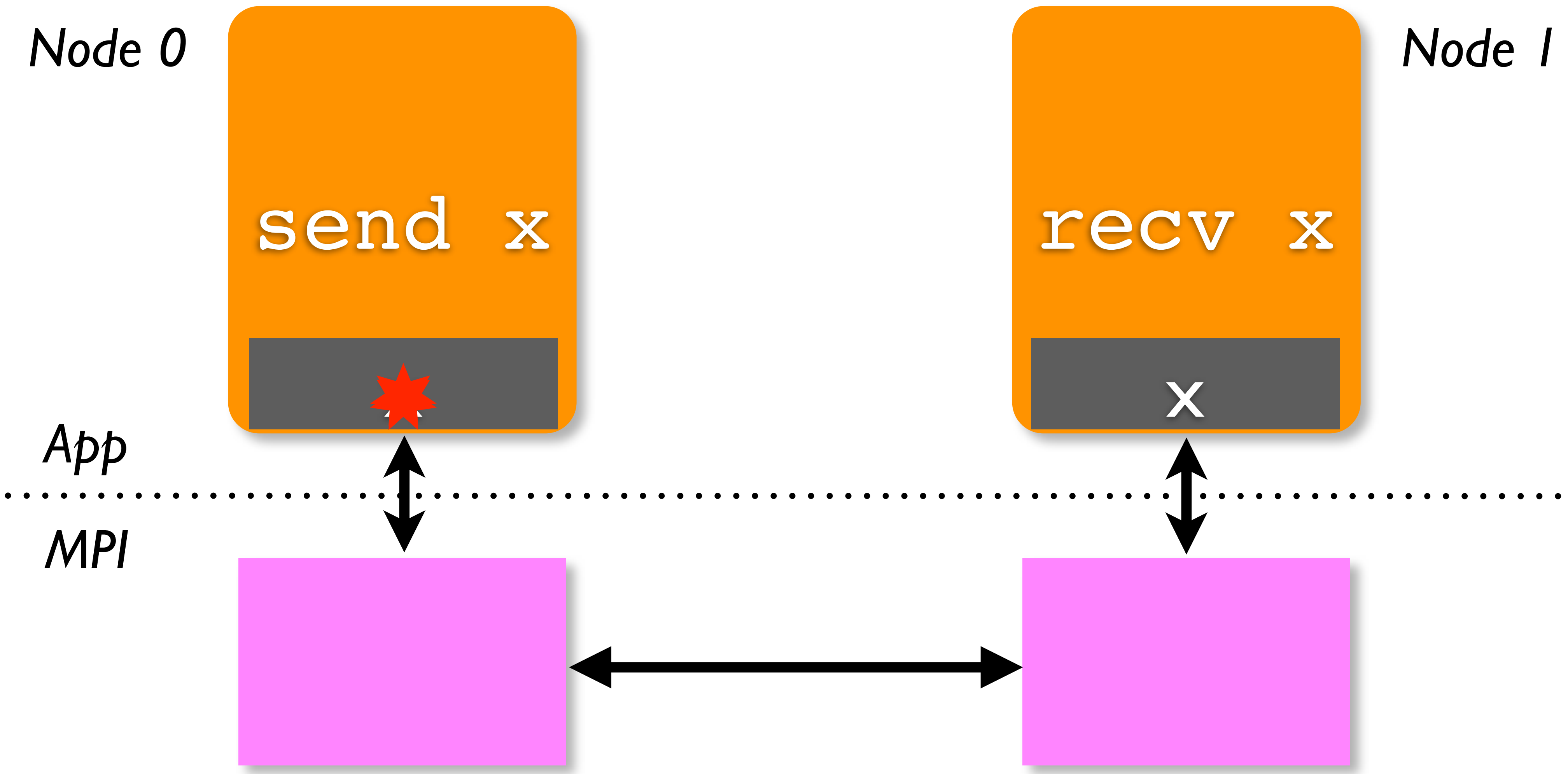
Compiling to MPI

```
@communicate {x@1 <<= x@0}
```



Compiling to MPI

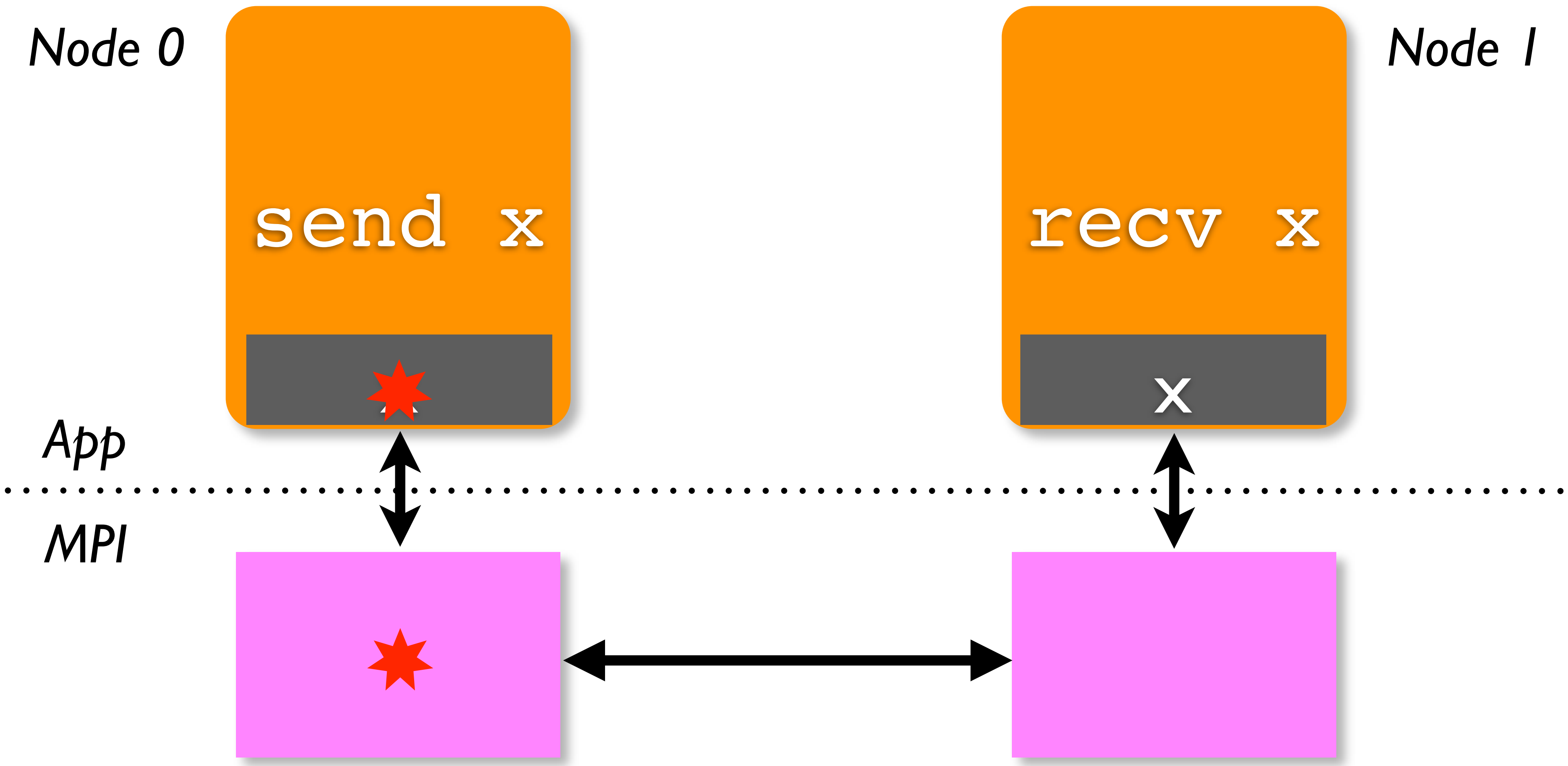
`@communicate {x@1 <<= x@0}`



Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

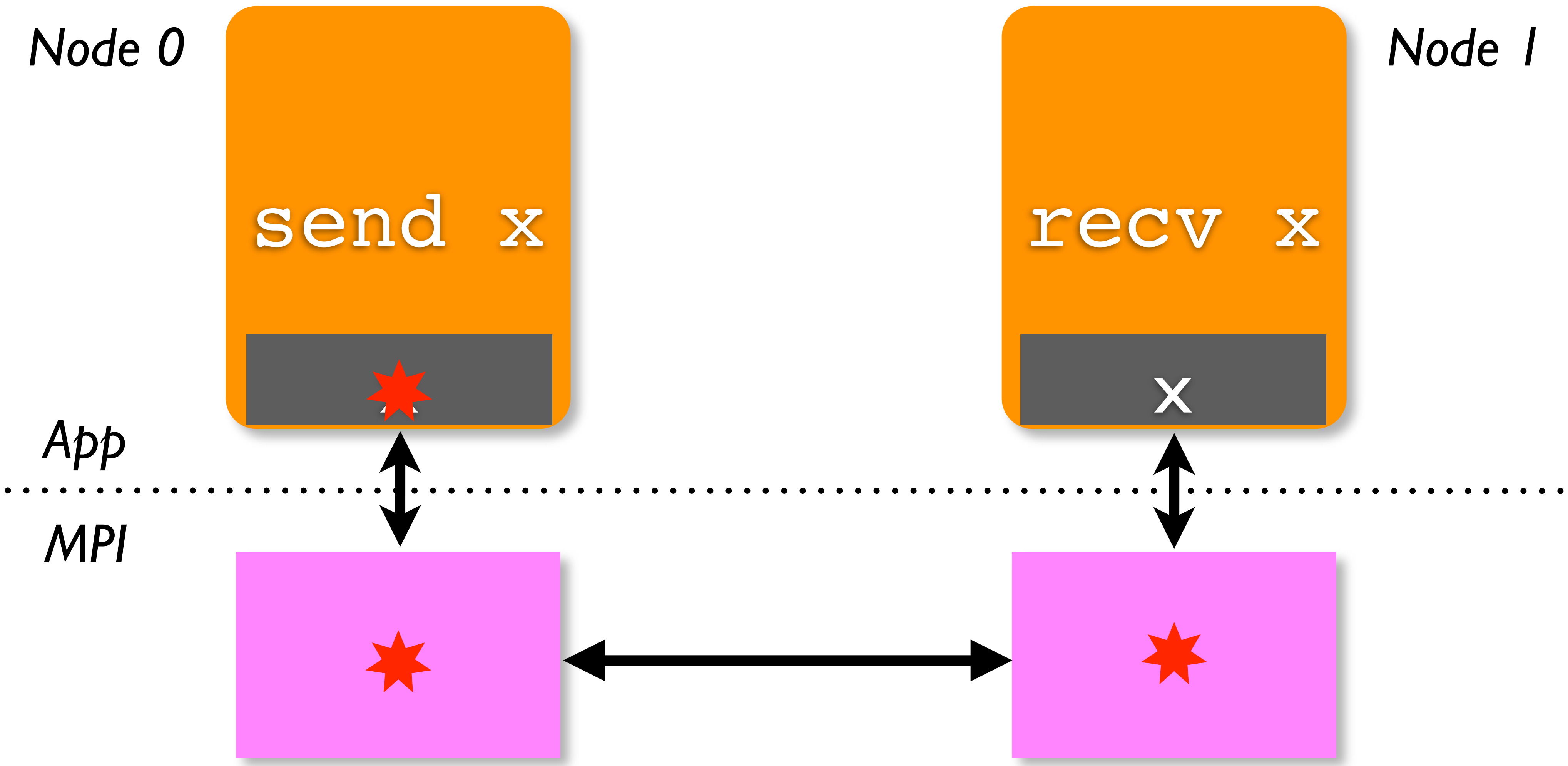
Compiling to MPI

```
@communicate {x@1 <<= x@0}
```



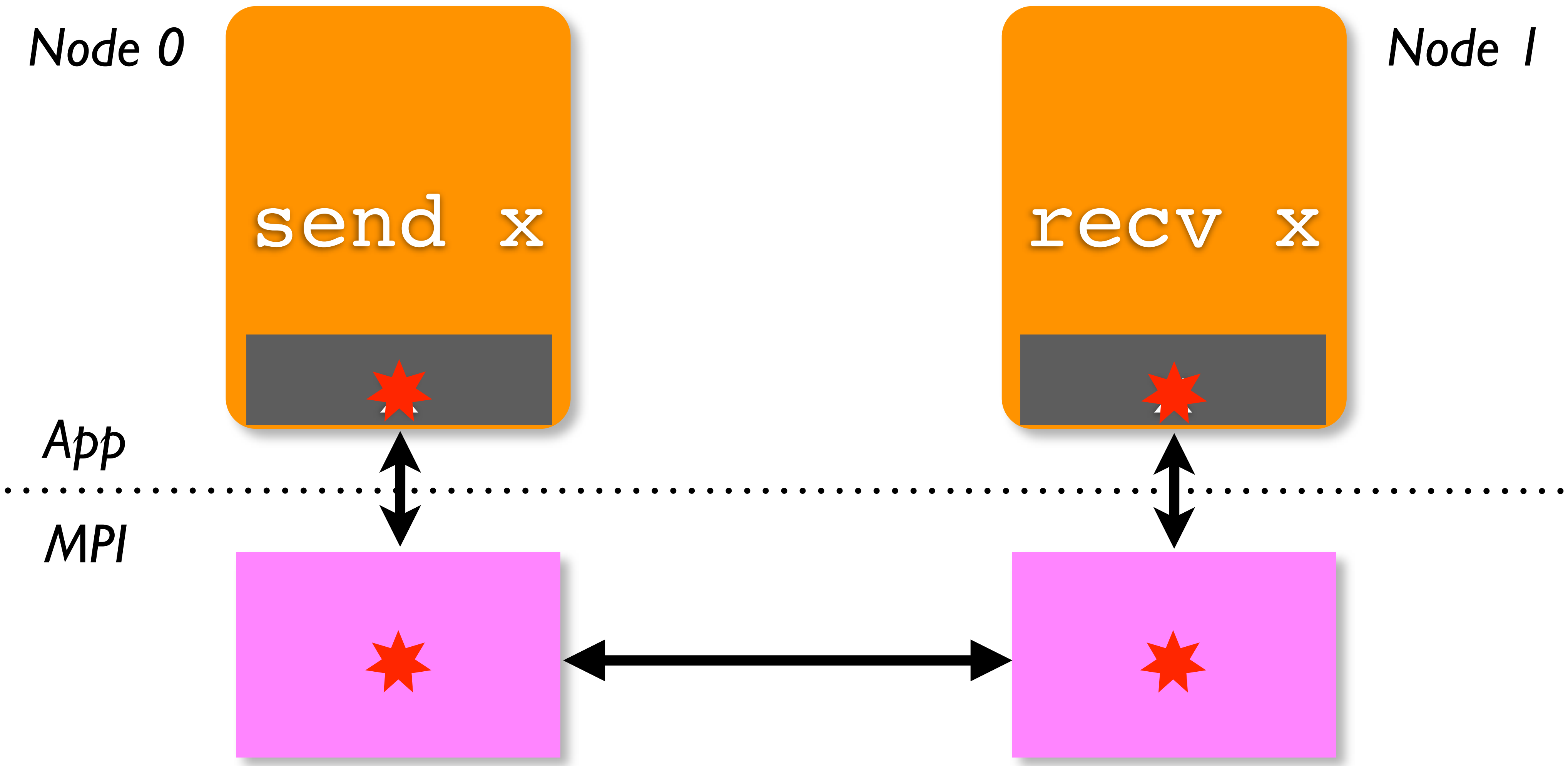
Compiling to MPI

`@communicate {x@1 <<= x@0}`



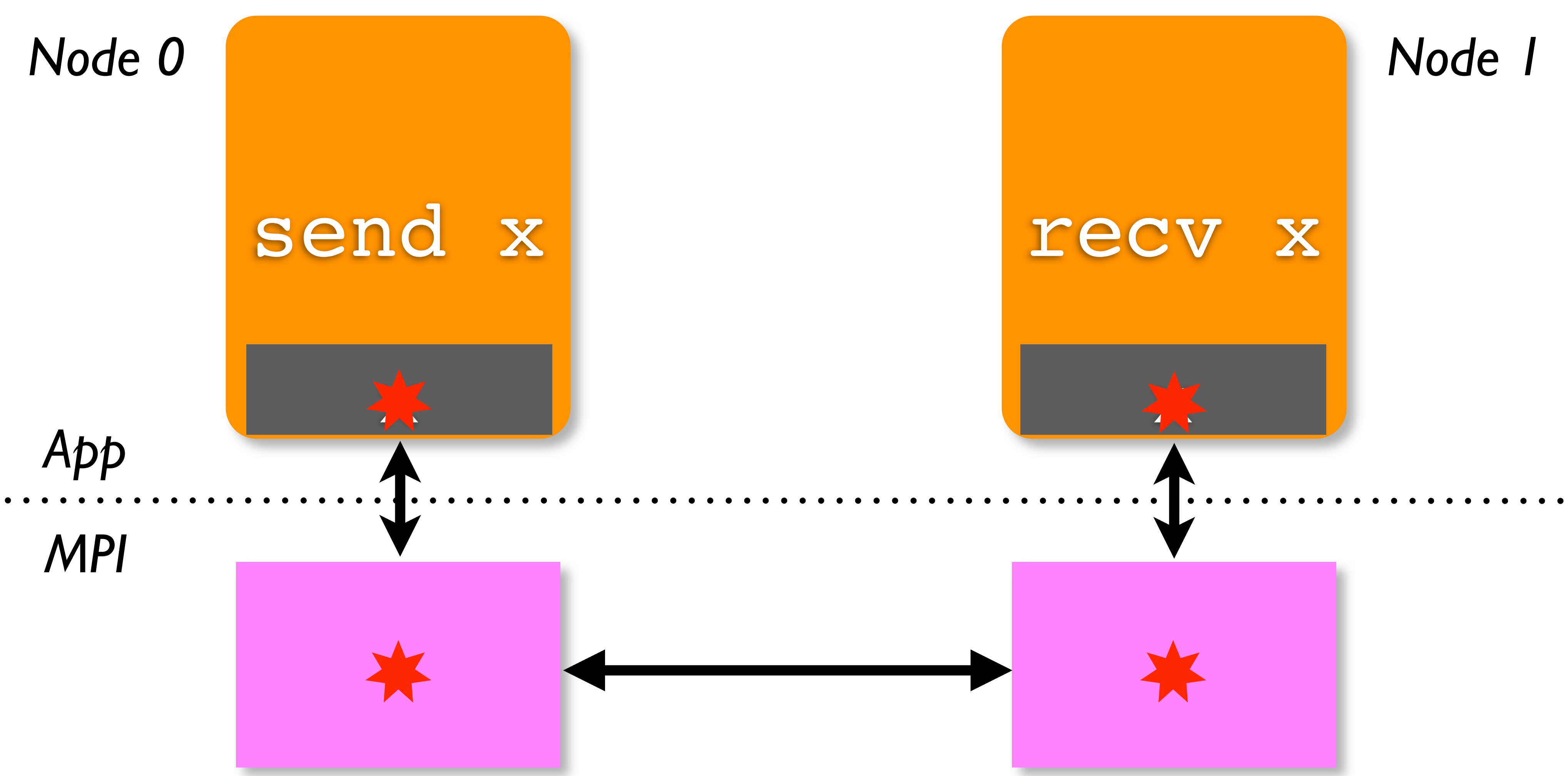
Compiling to MPI

`@communicate {x@1 <<= x@0}`



Compiling to MPI

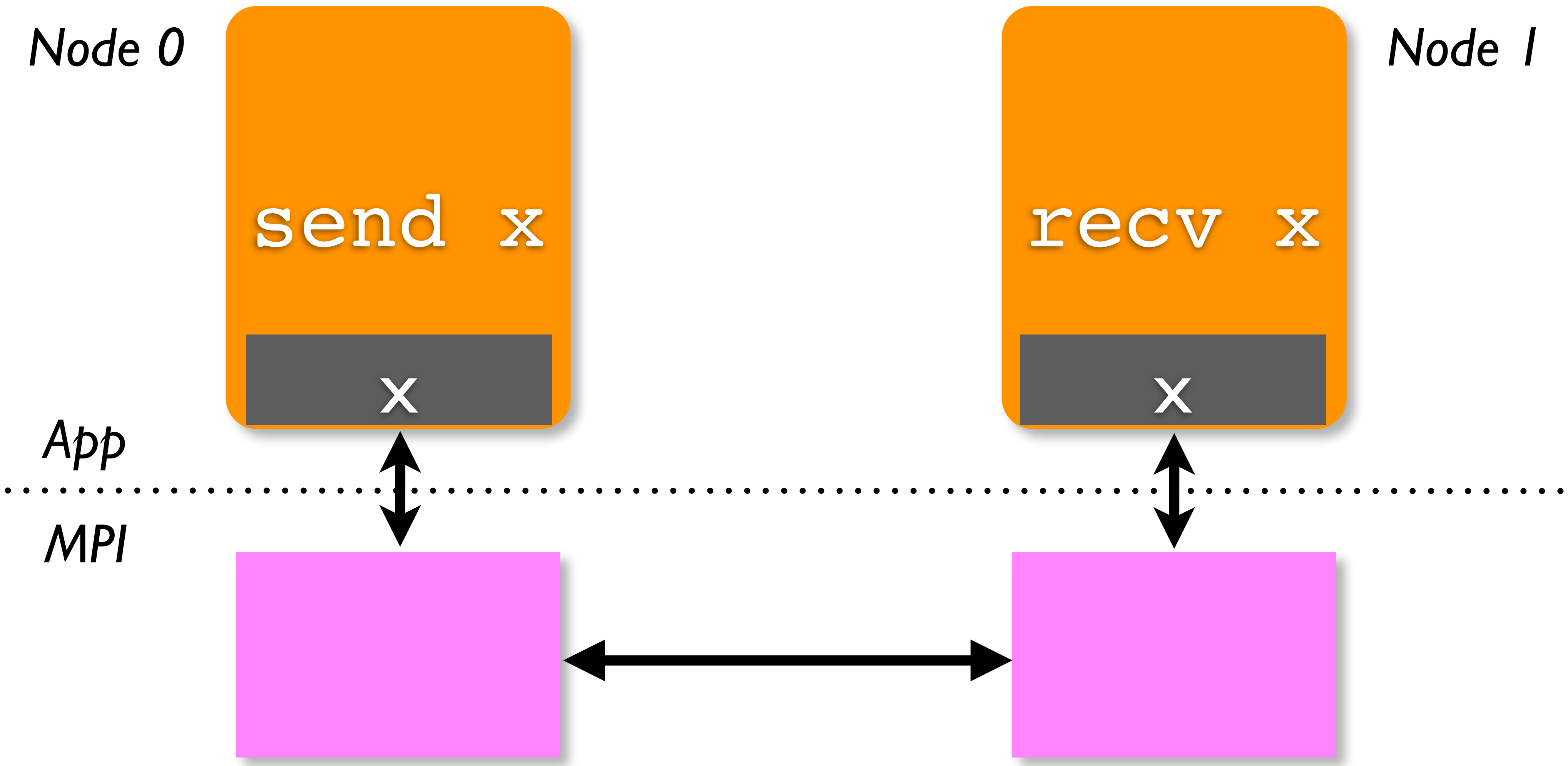
`@communicate {x@1 <<= x@0}`



3 copies

MPI Optimized for Shared Memory

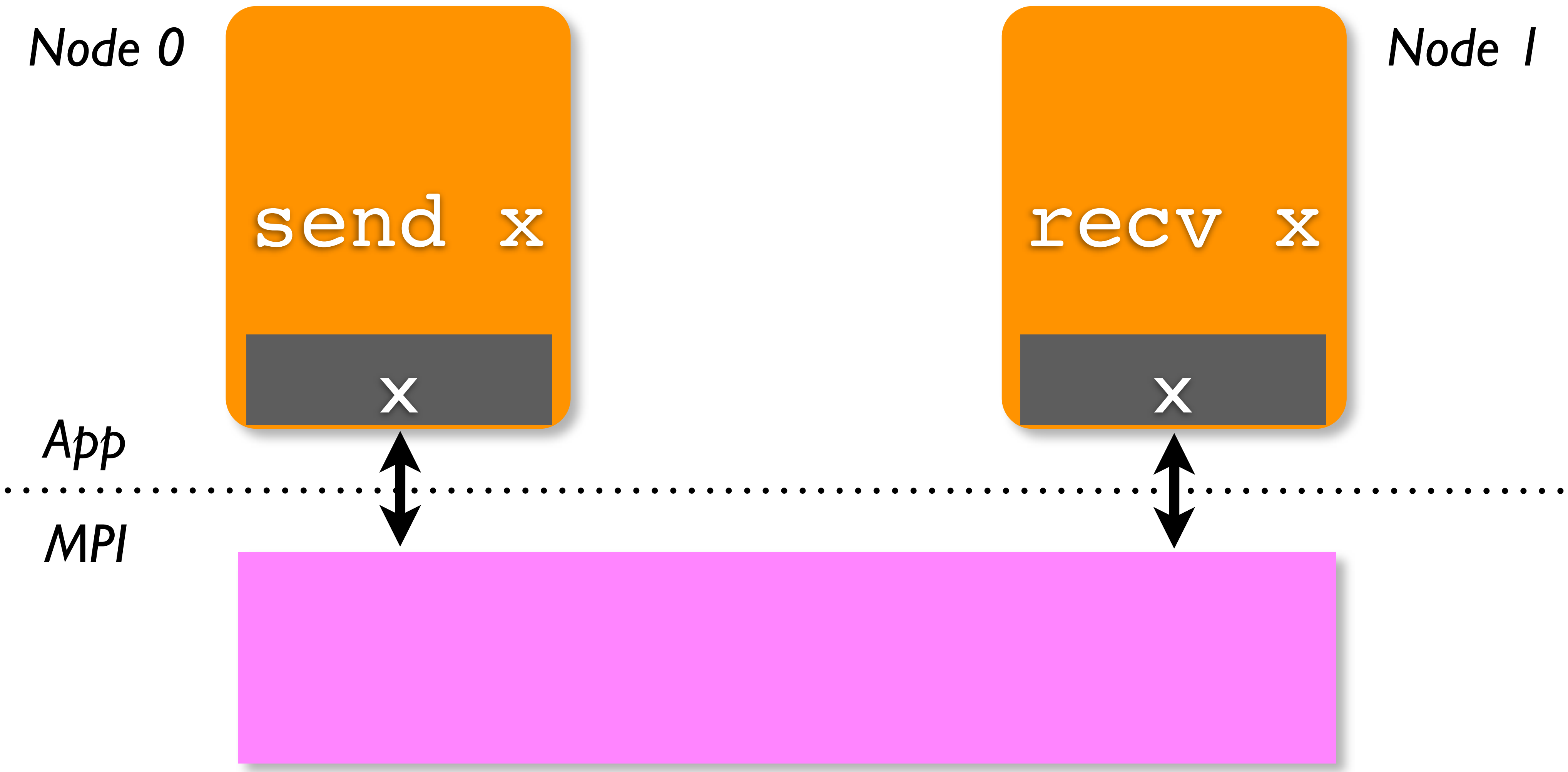
`@communicate {x@1 <<= x@0}`



Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

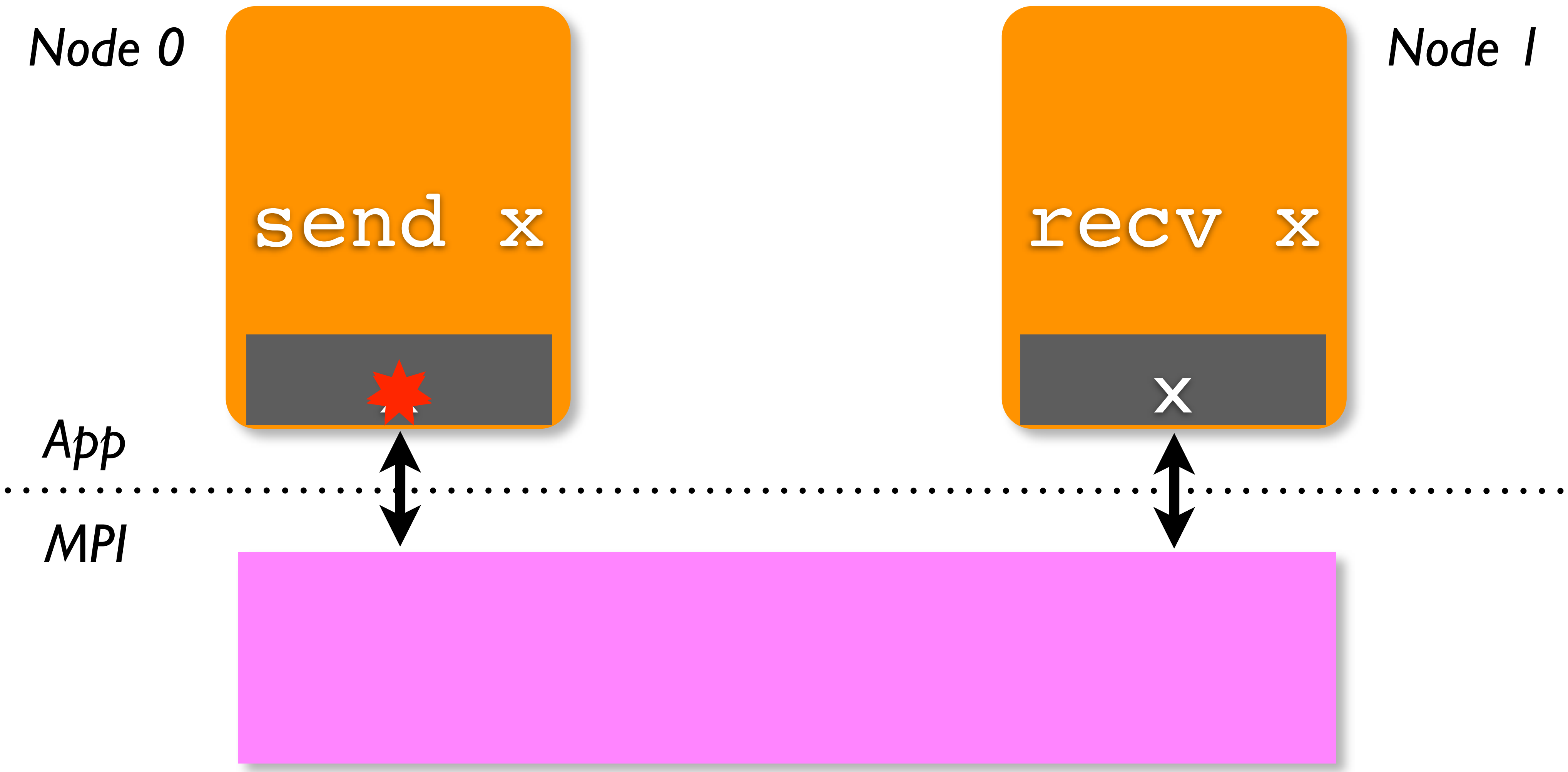
MPI Optimized for Shared Memory

`@communicate {x@1 <<= x@0}`



MPI Optimized for Shared Memory

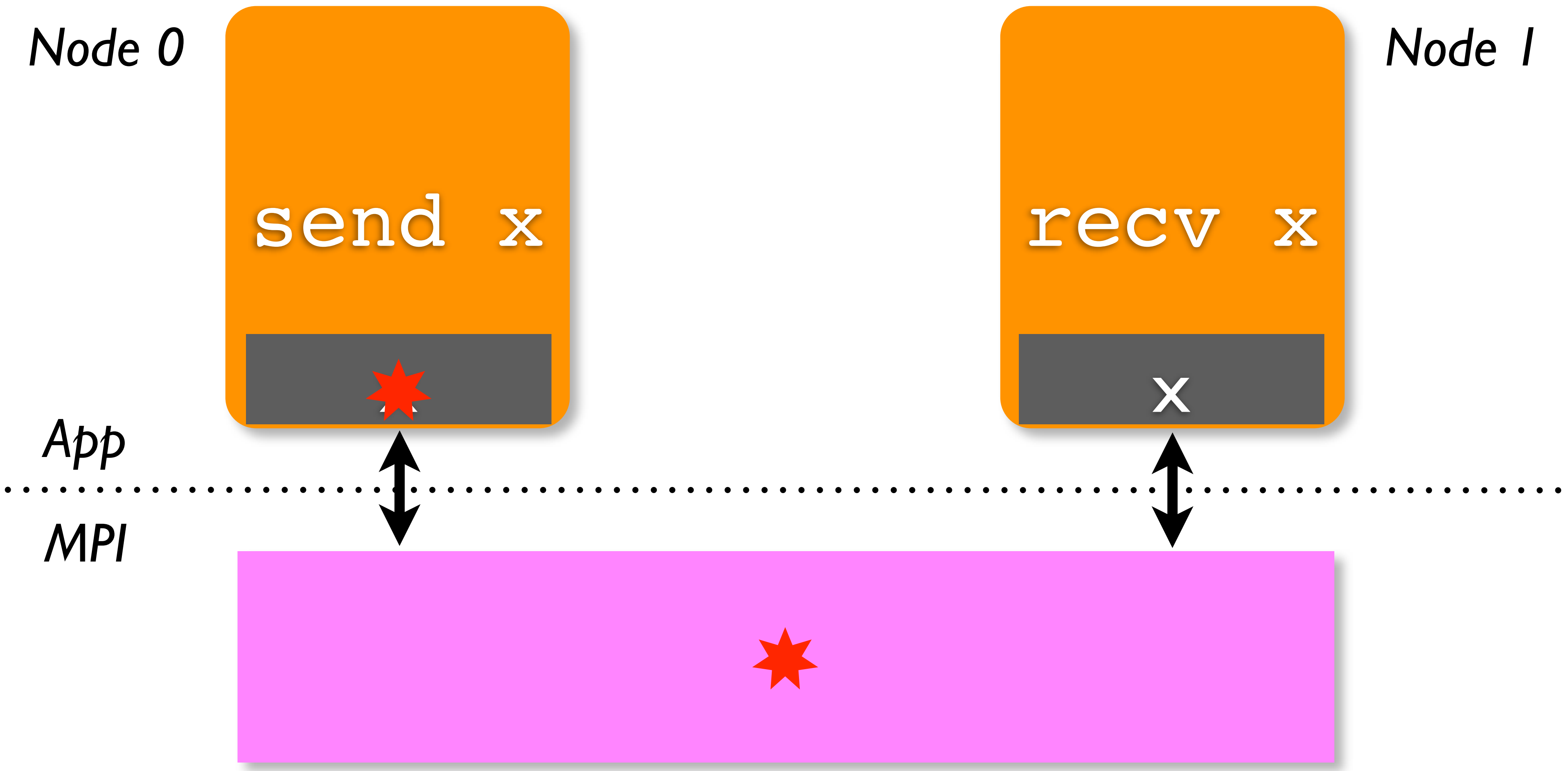
`@communicate {x@1 <<= x@0}`



Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

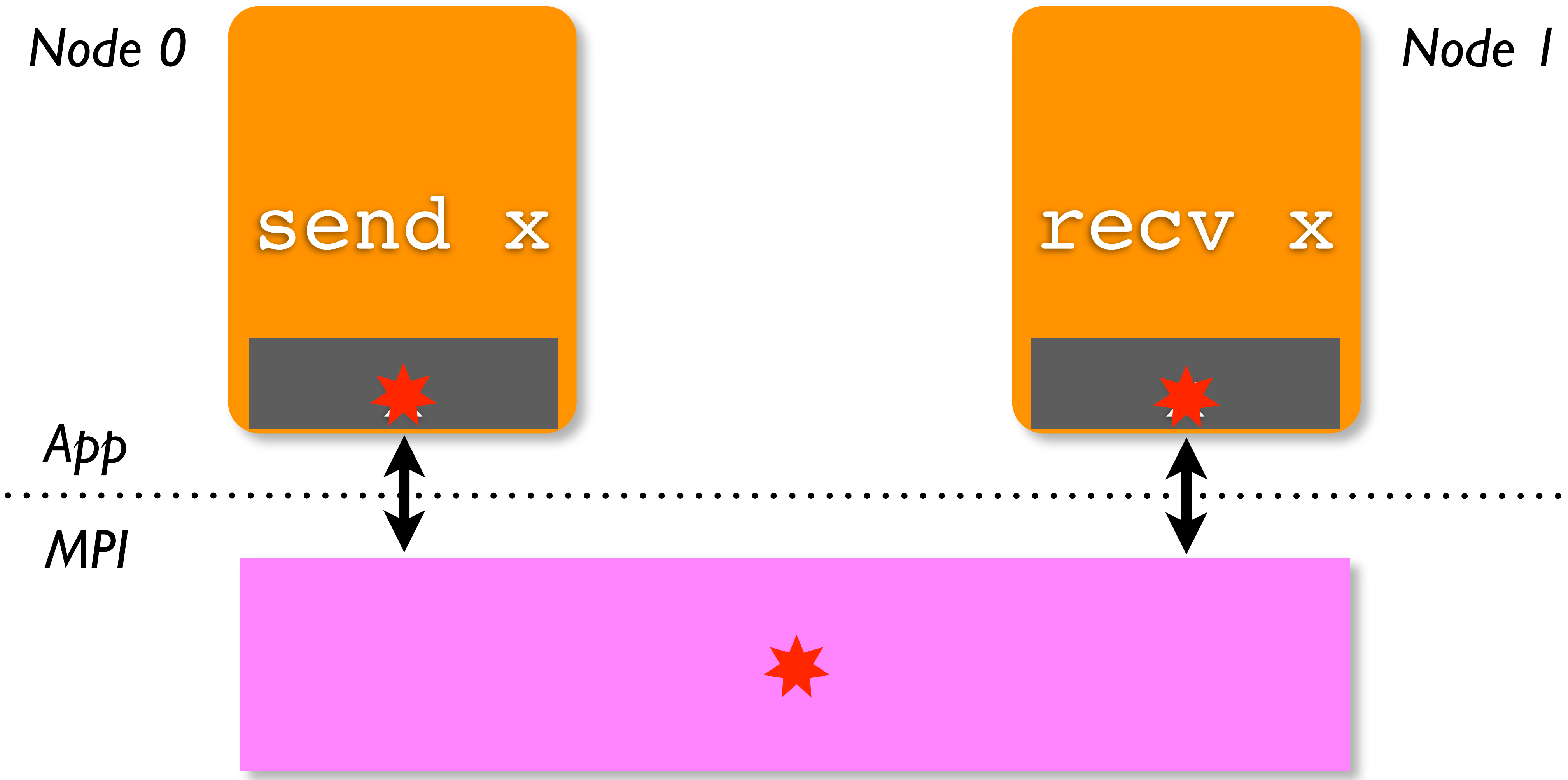
MPI Optimized for Shared Memory

`@communicate {x@1 <<= x@0}`



MPI Optimized for Shared Memory

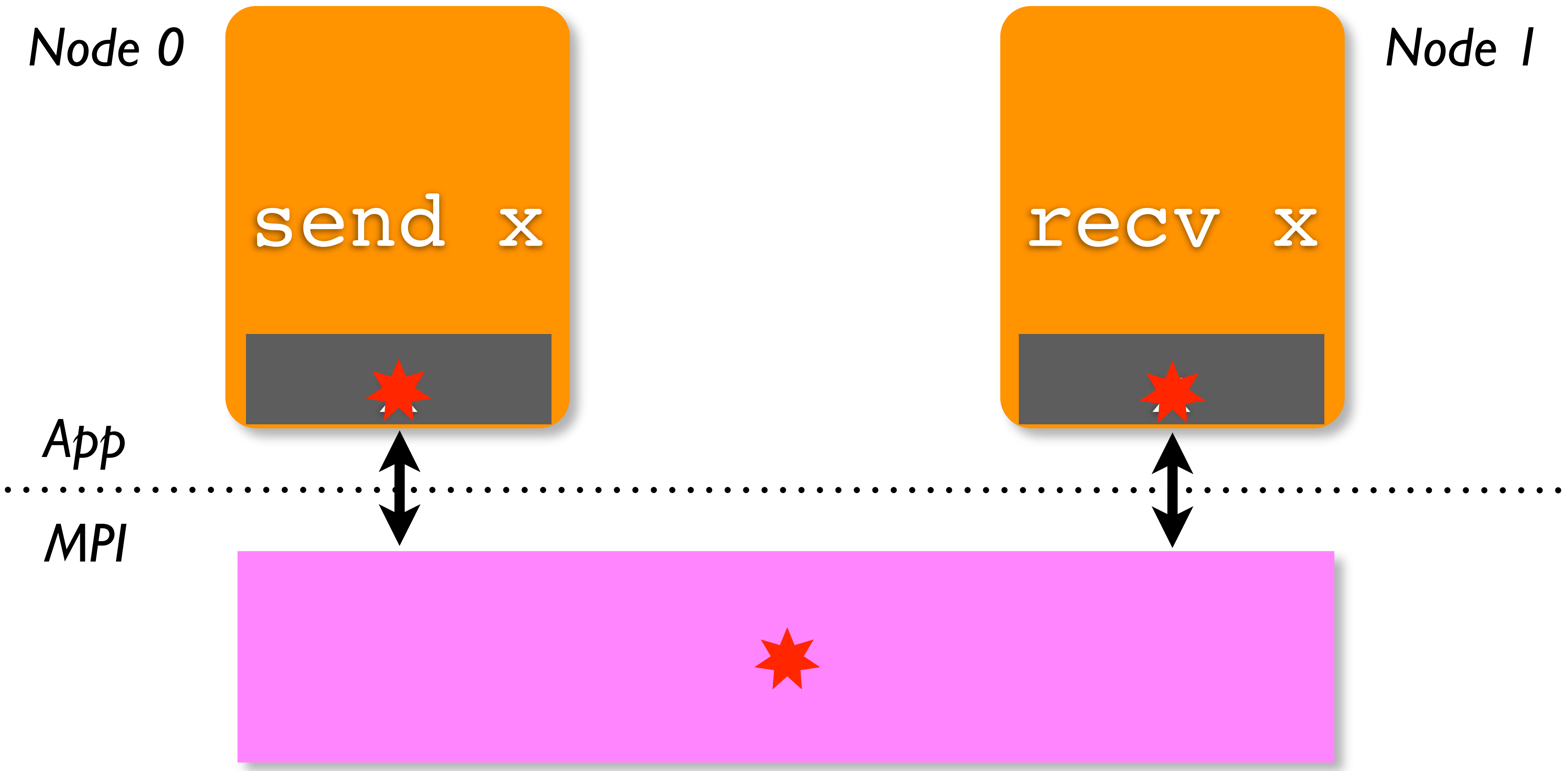
`@communicate {x@1 <<= x@0}`



Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

MPI Optimized for Shared Memory

```
@communicate {x@1 <<= x@0}
```

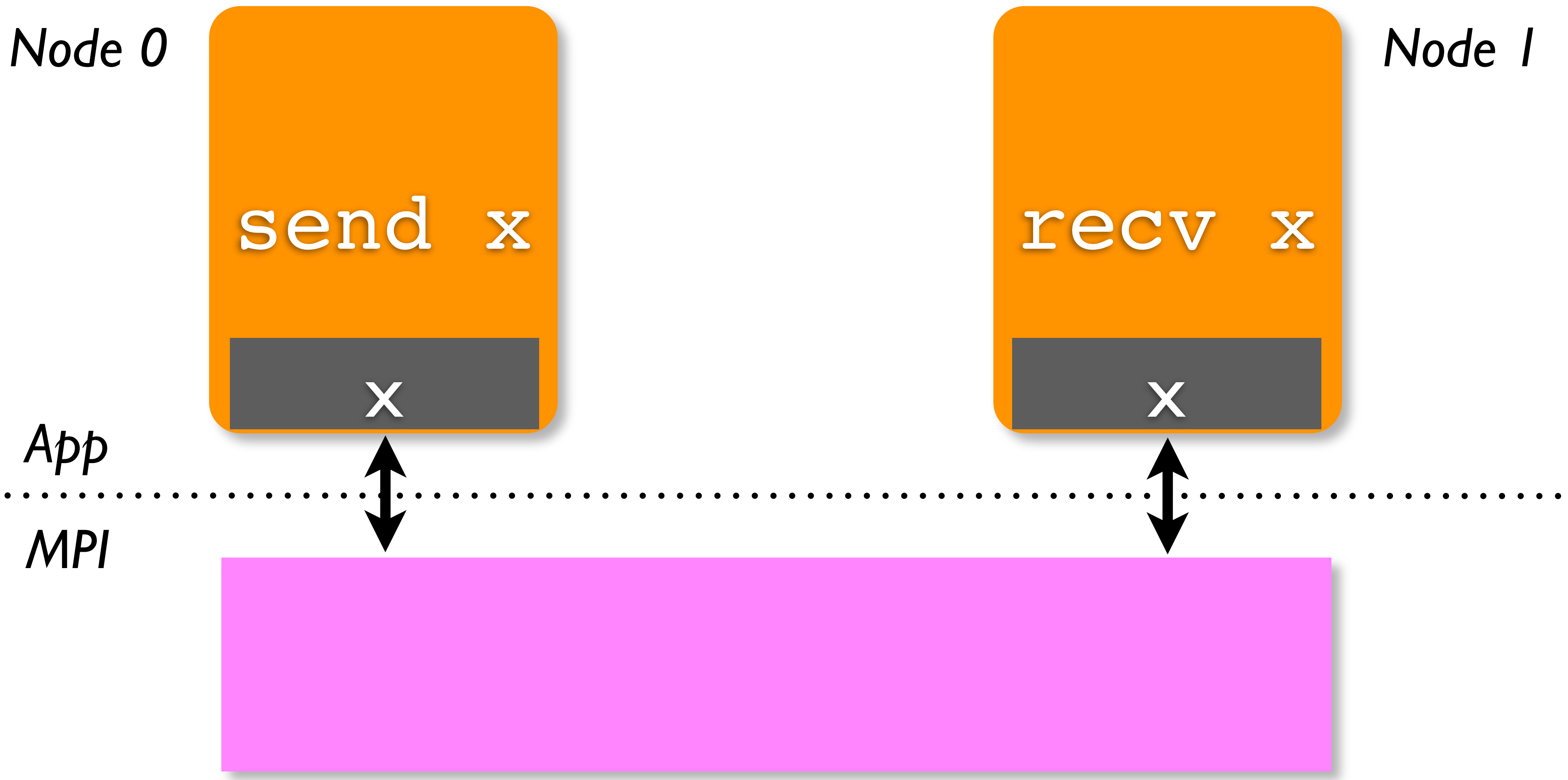


2 copies

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

Optimizing for Shared Memory

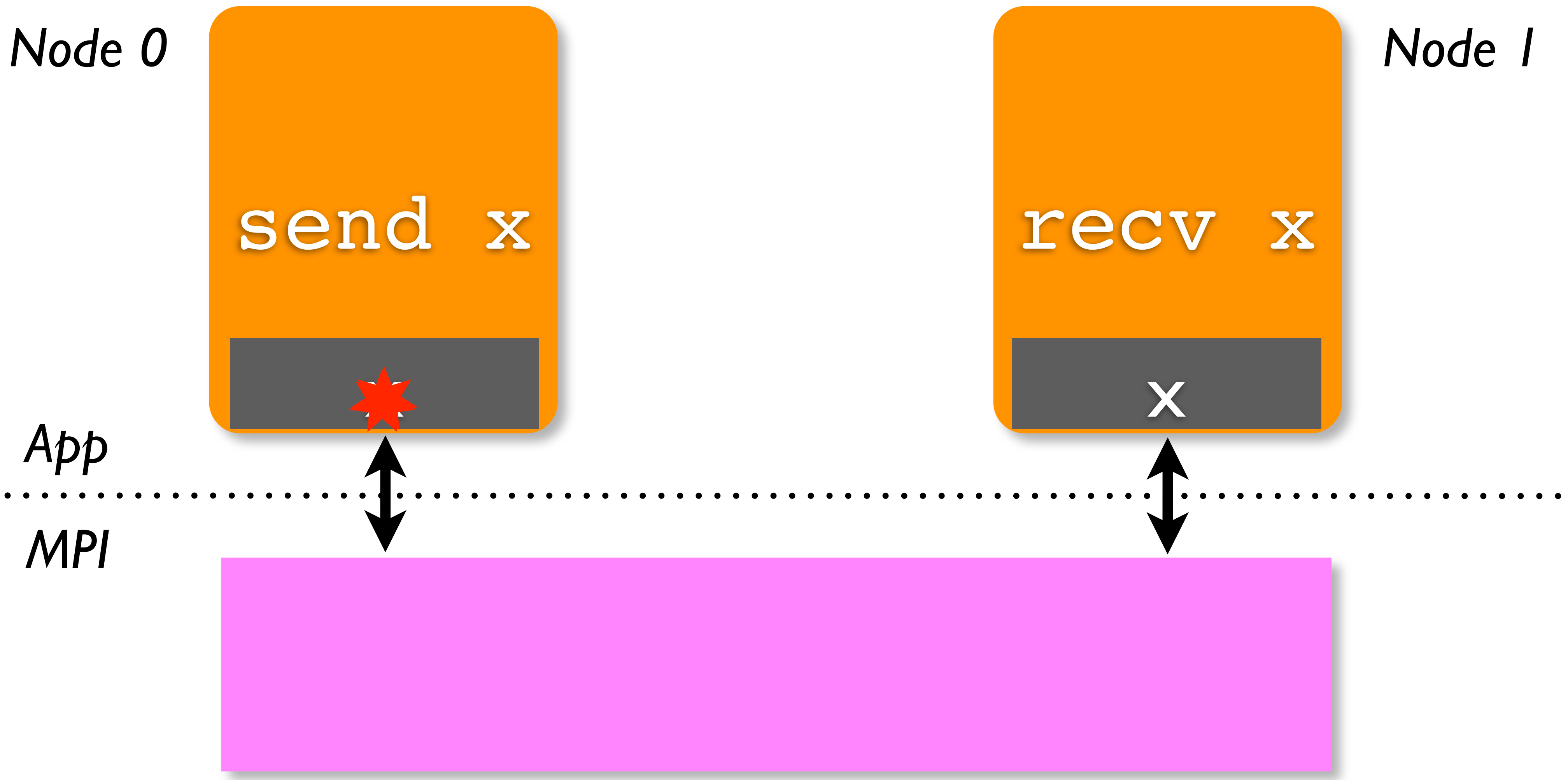
```
@communicate {x@1 <<= x@1}
```



Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

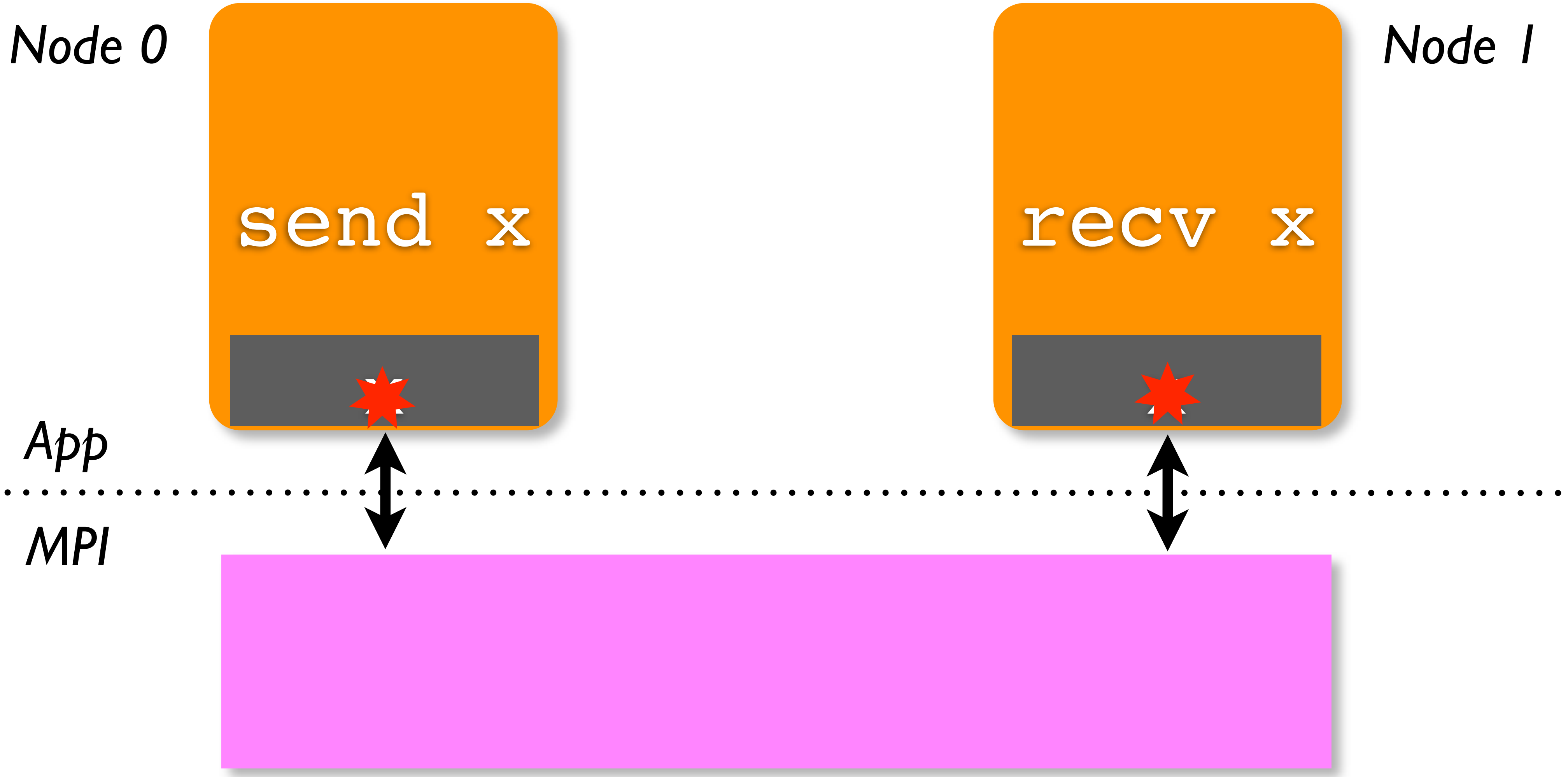
Optimizing for Shared Memory

`@communicate {x@1 <<= x@1}`



Optimizing for Shared Memory

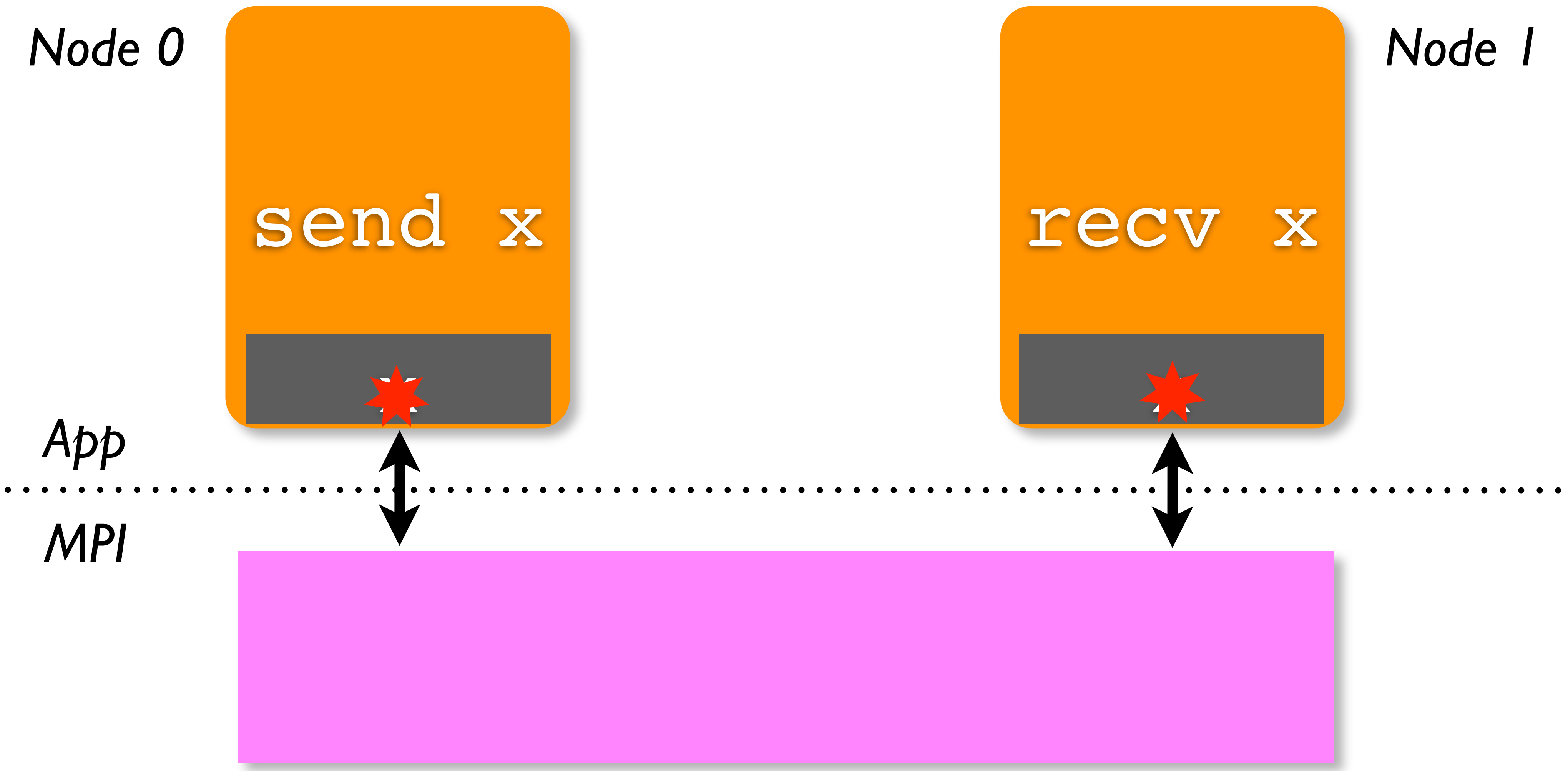
`@communicate {x@1 <<= x@1}`



Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

Optimizing for Shared Memory

```
@communicate {x@1 <<= x@1}
```

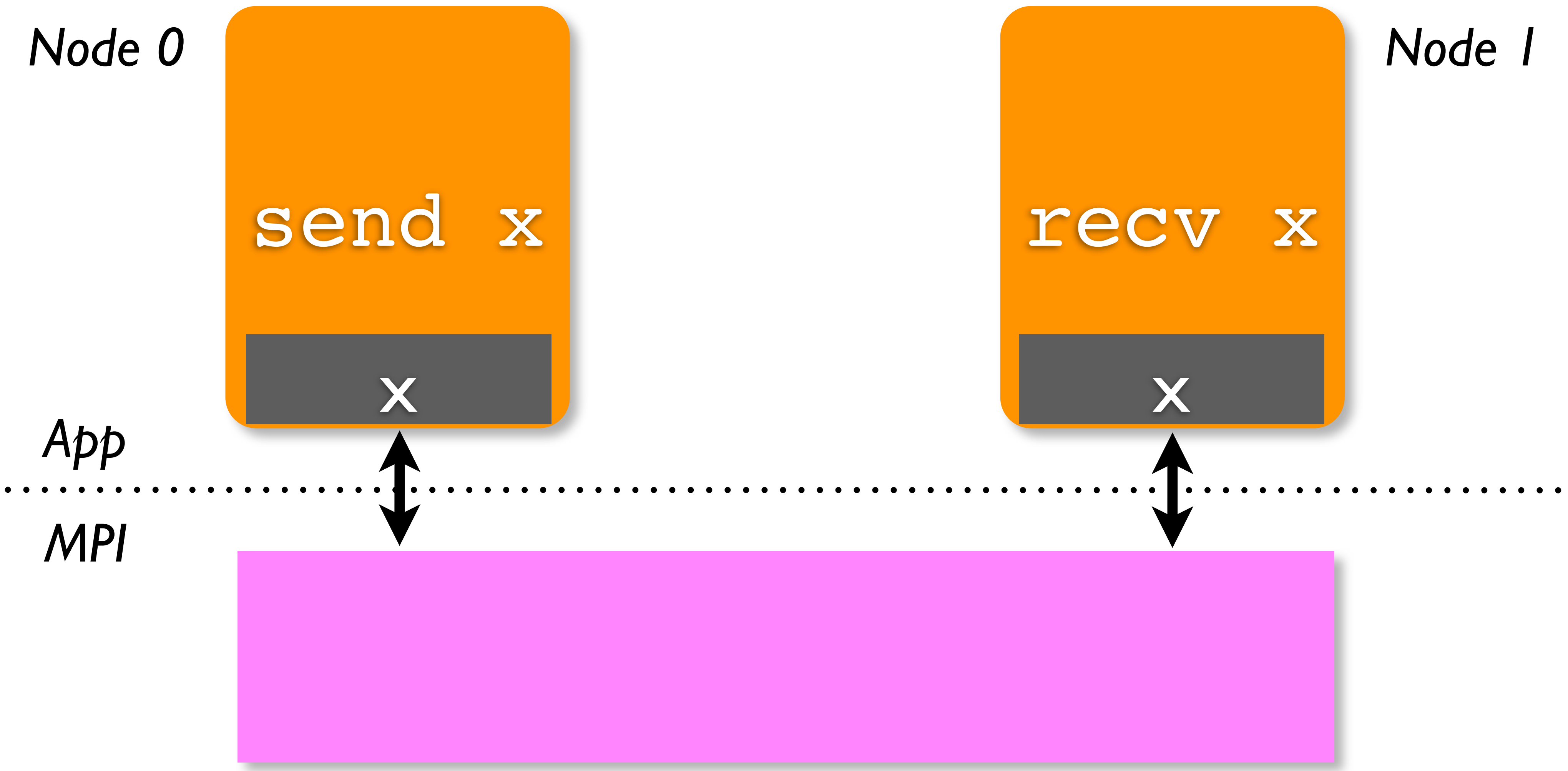


1 copy

(requires rendezvous or compiler intervention)

Optimizing for Shared Memory

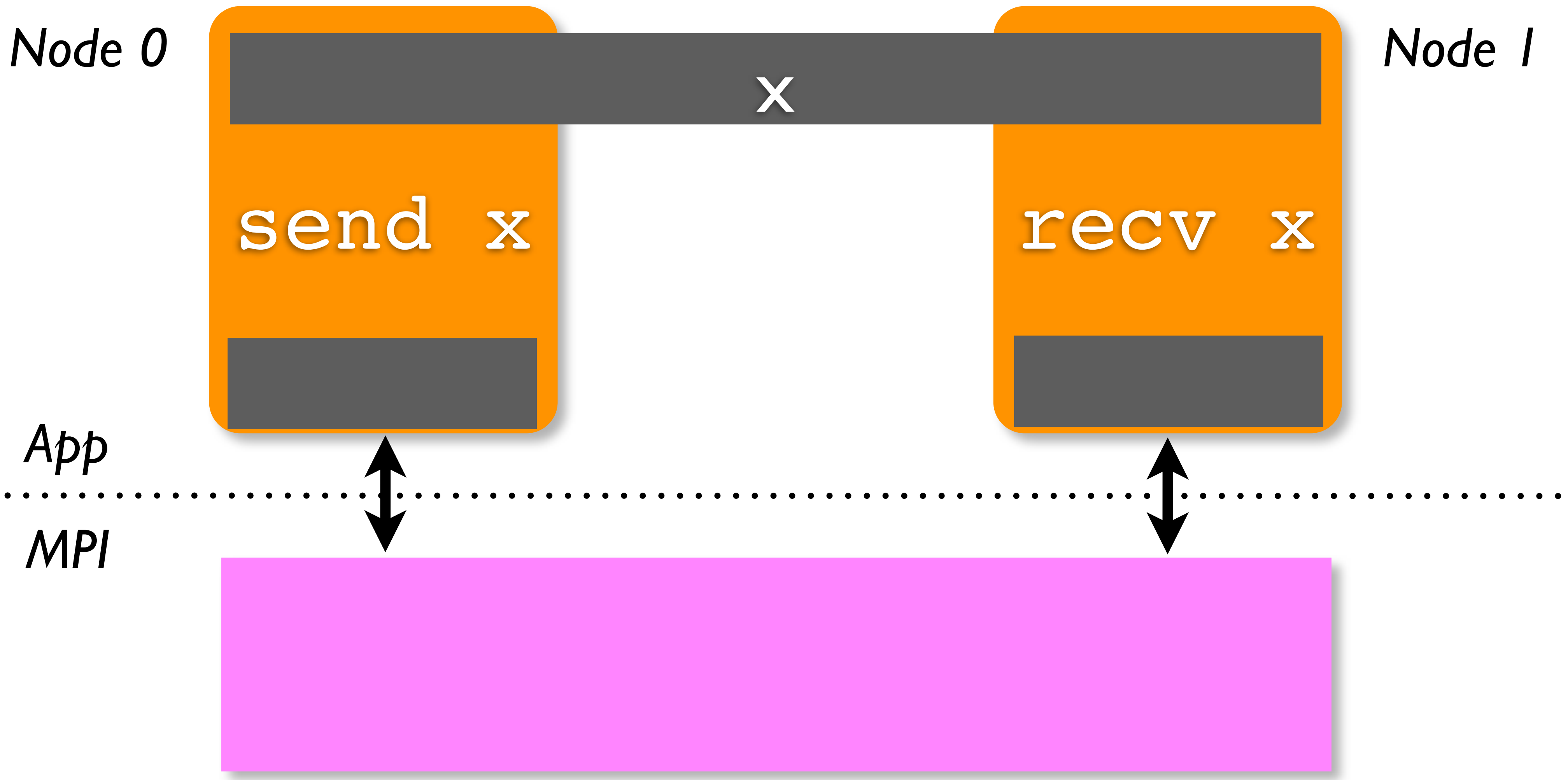
`@communicate {x@0 <<= x@1}`



Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

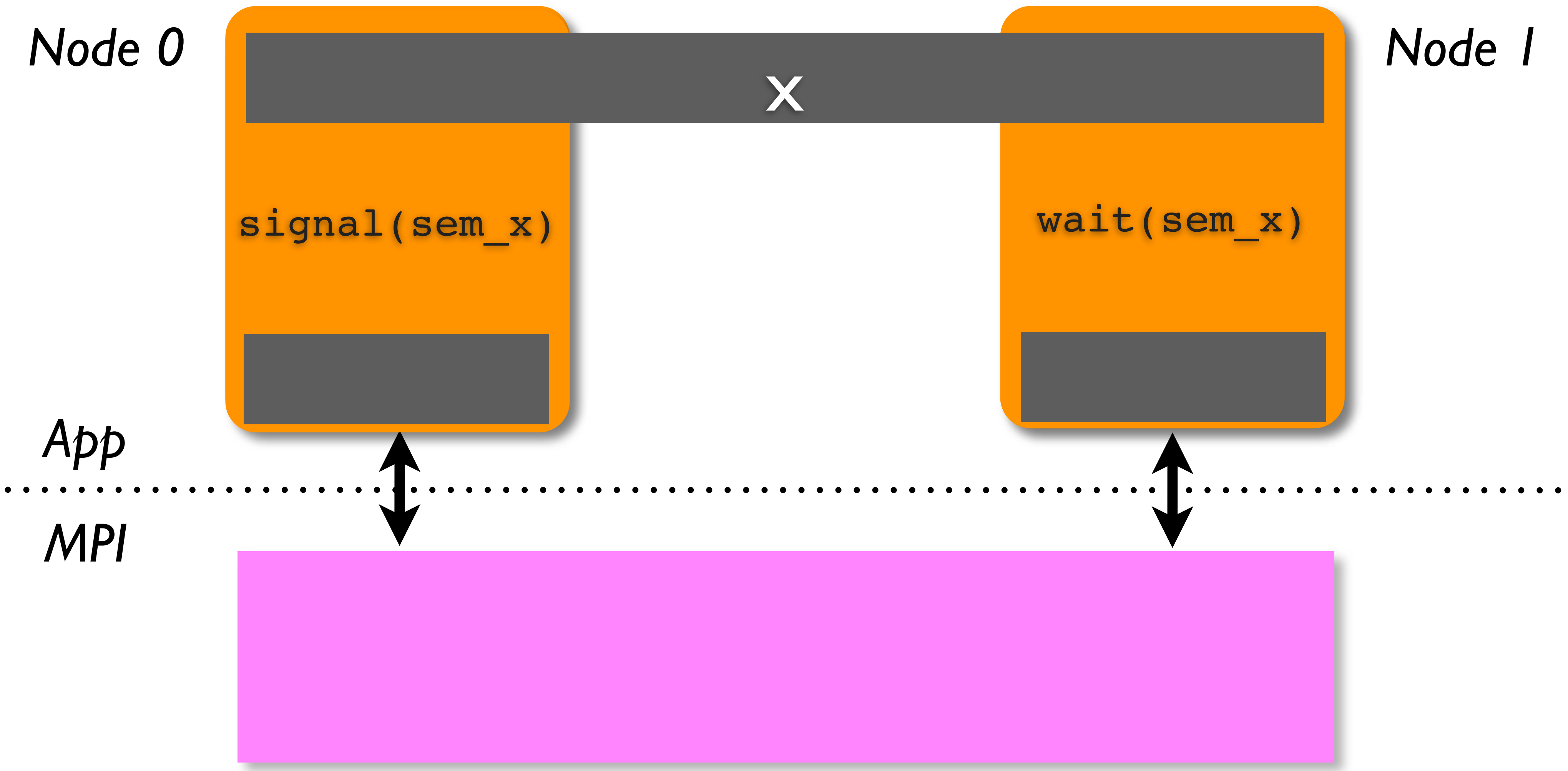
Optimizing for Shared Memory

`@communicate {x@0 <<= x@1}`



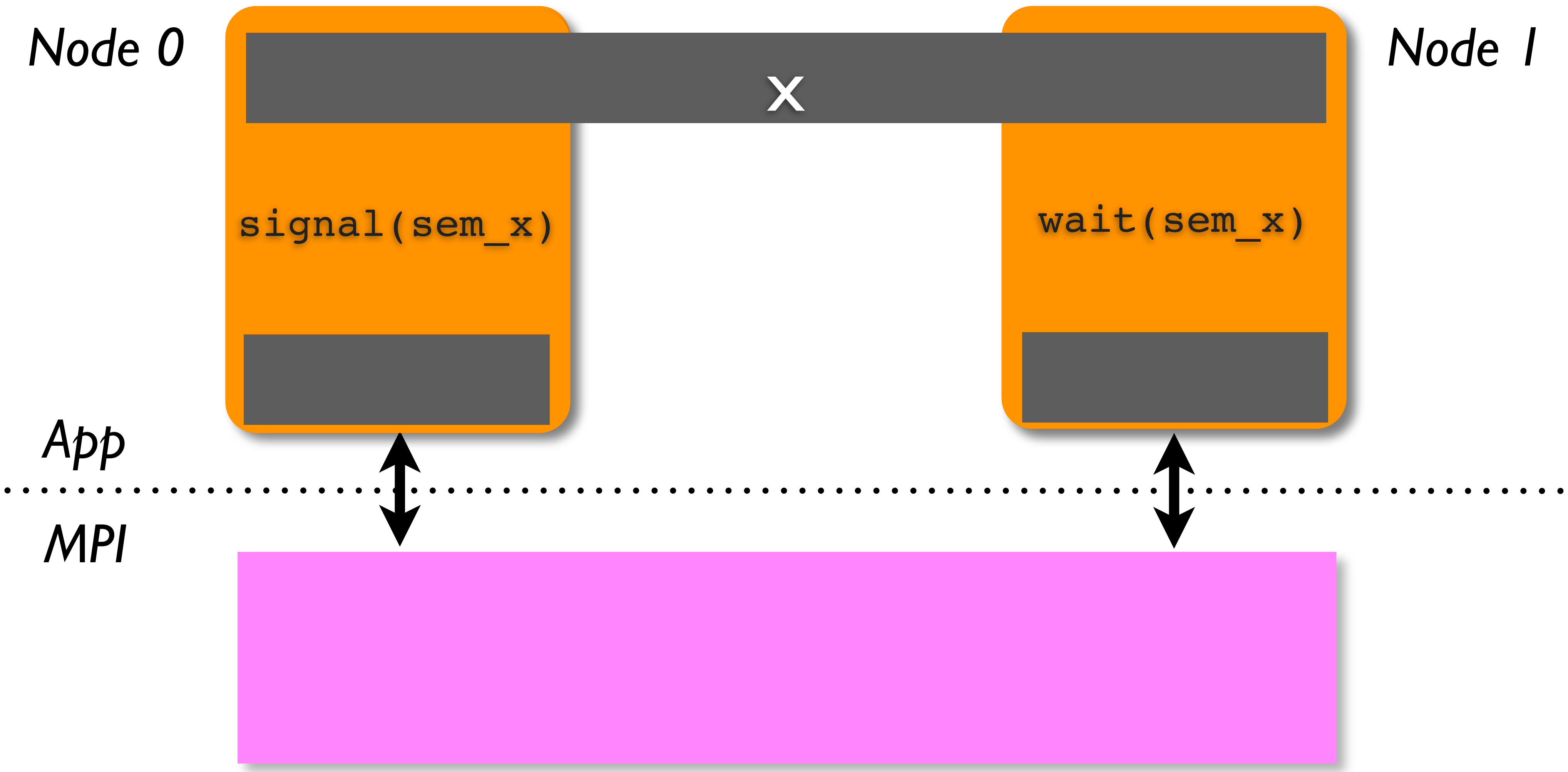
Optimizing for Shared Memory

`@communicate {x@0 <<= x@1}`



Optimizing for Shared Memory

```
@communicate {x@0 <<= x@1}
```

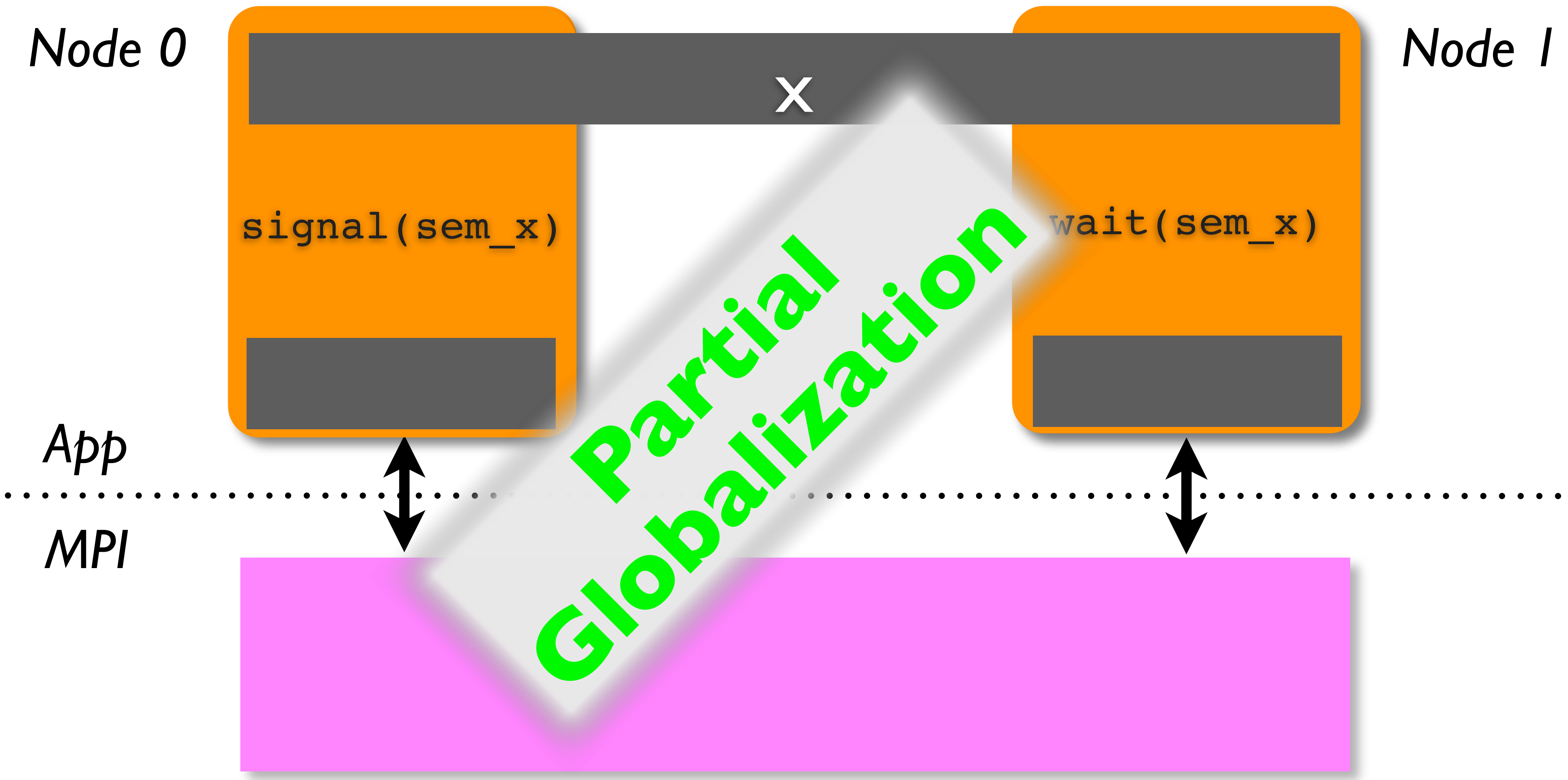


0 copy

(requires compiler intervention)

Optimizing for Shared Memory

`@communicate {x@0 <<= x@1}`



0 copy

(requires compiler intervention)

Steps for Optimizing Communication with Shared Memory

- Identify globalization candidates
- Ensure correctness
  - insert appropriate synchronization
- Minimize contention
  - minimize synchronization points
  - minimize synchronization overheads
    - *using a run-time trick*

Globalization Candidates

- Contiguous chunks of memory
  - excluding strided array sections, for example
  - contiguous array sections OK (but not implemented)
- Large buffers
  - communication inside loops
- Small local reuse

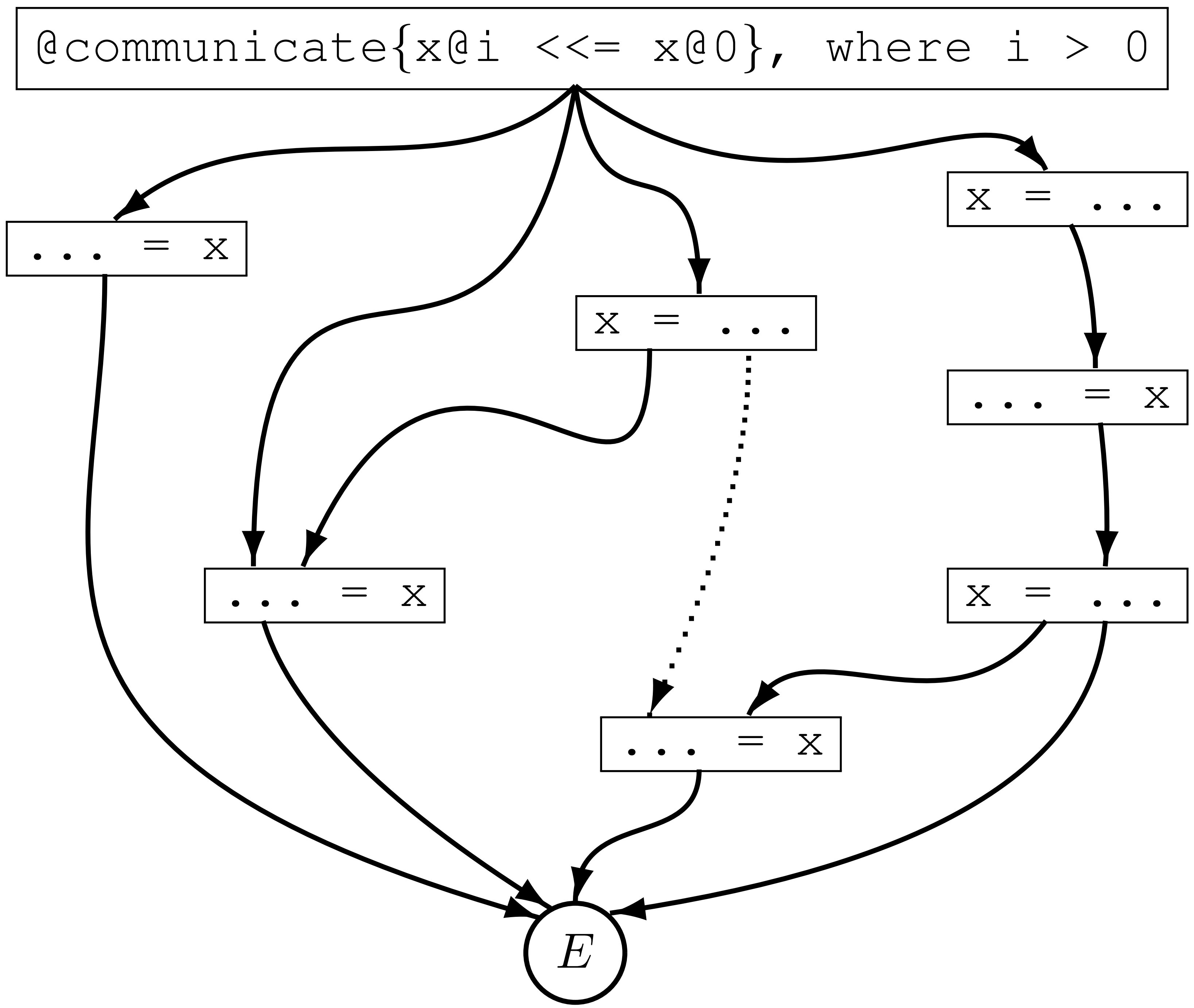
Ensuring Correctness

```
@communicate {x@i <<= x@i+1,
                 where i in Kanor::WORLD}
...
consume(A);       // consume communicated data
...
overwrite(A);     // reuse A for local data
...
consume(A);      // consume local data
```

Ensuring Correctness

```
@communicate {x@i <<= x@i+1,
              where i in Kanor::WORLD}
…
consume(A);      // consume communicated data
…
overwrite(A);    // reuse A for local data
…
consume(A);      // consume local data
```

Correctness Issues



Observations

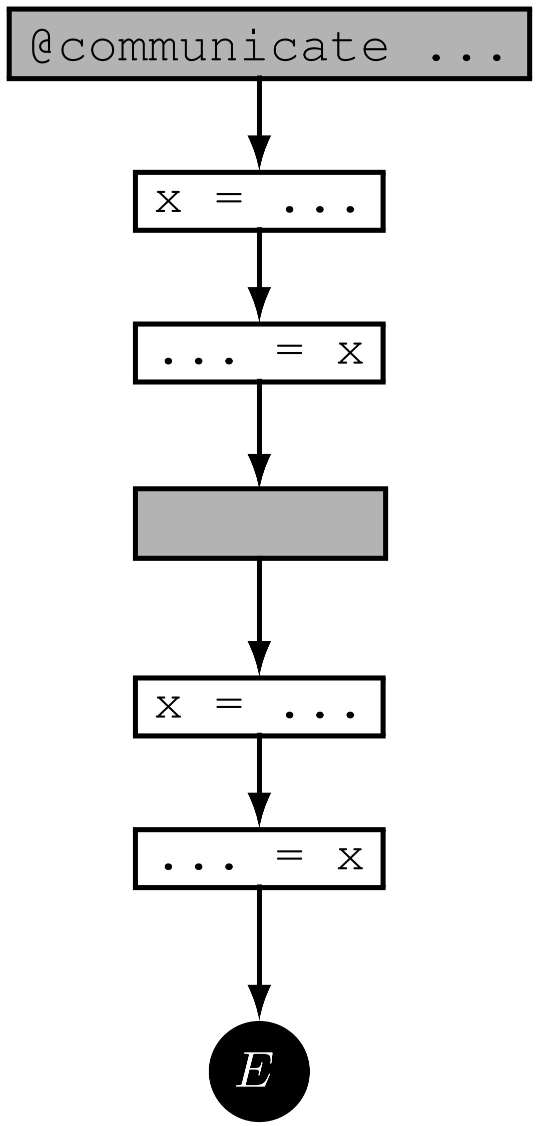
Definition: *Locking Set: The set of CFG nodes that lie on a path from a node containing local write into a globalized variable to a node containing read of that value*

Observations

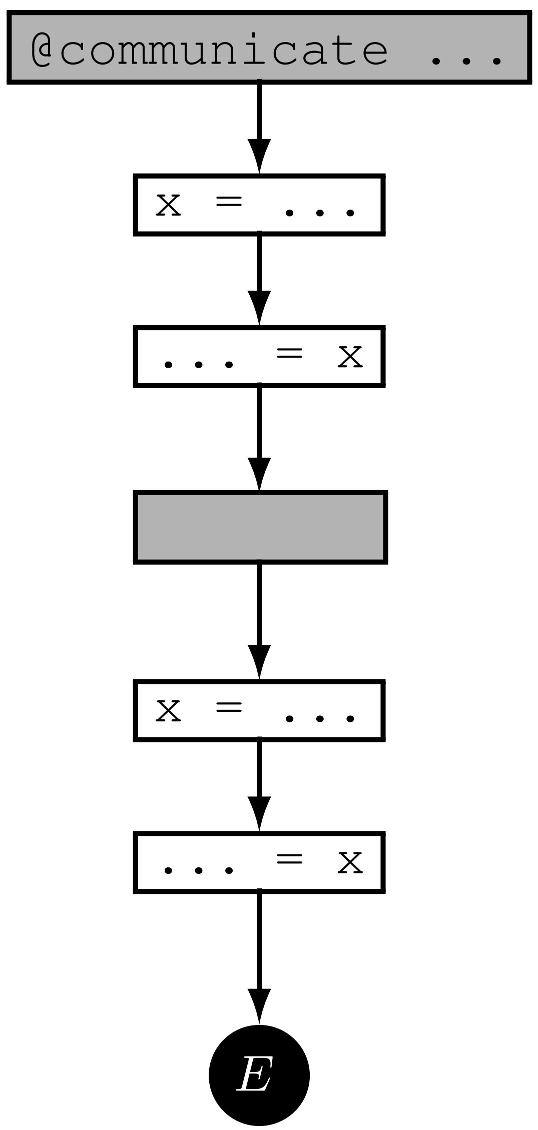
Definition: *Locking Set: The set of CFG nodes that lie on a path from a node containing local write into a globalized variable to a node containing read of that value*

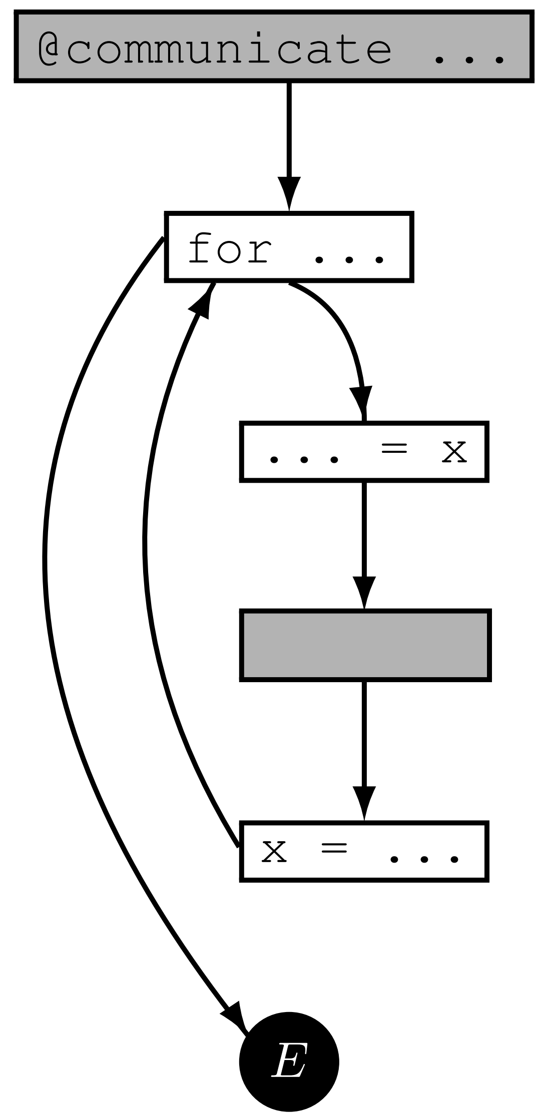
Theorem: *If the locking set belongs to a critical section then the partitioned address space semantics are maintained*

Correctness: Examples

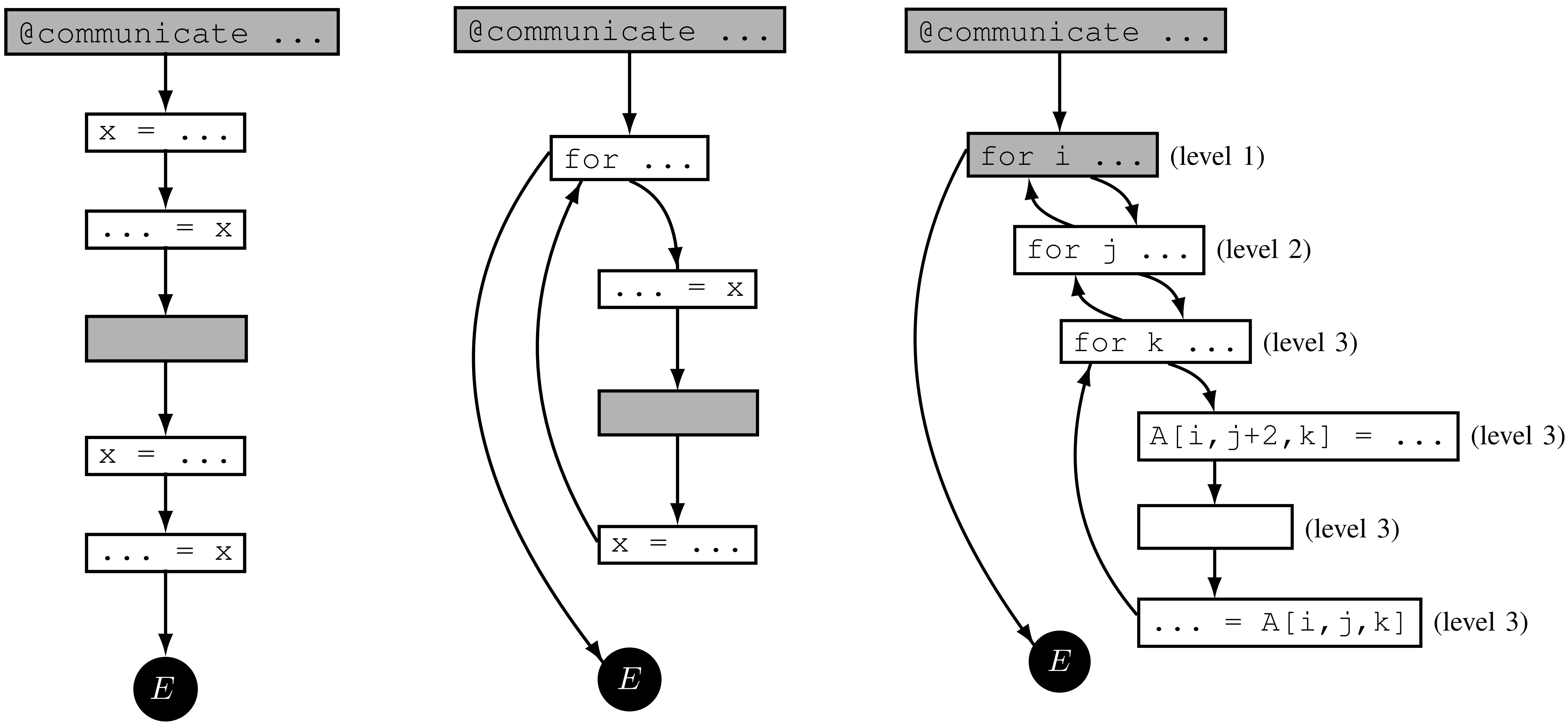


Correctness: Examples





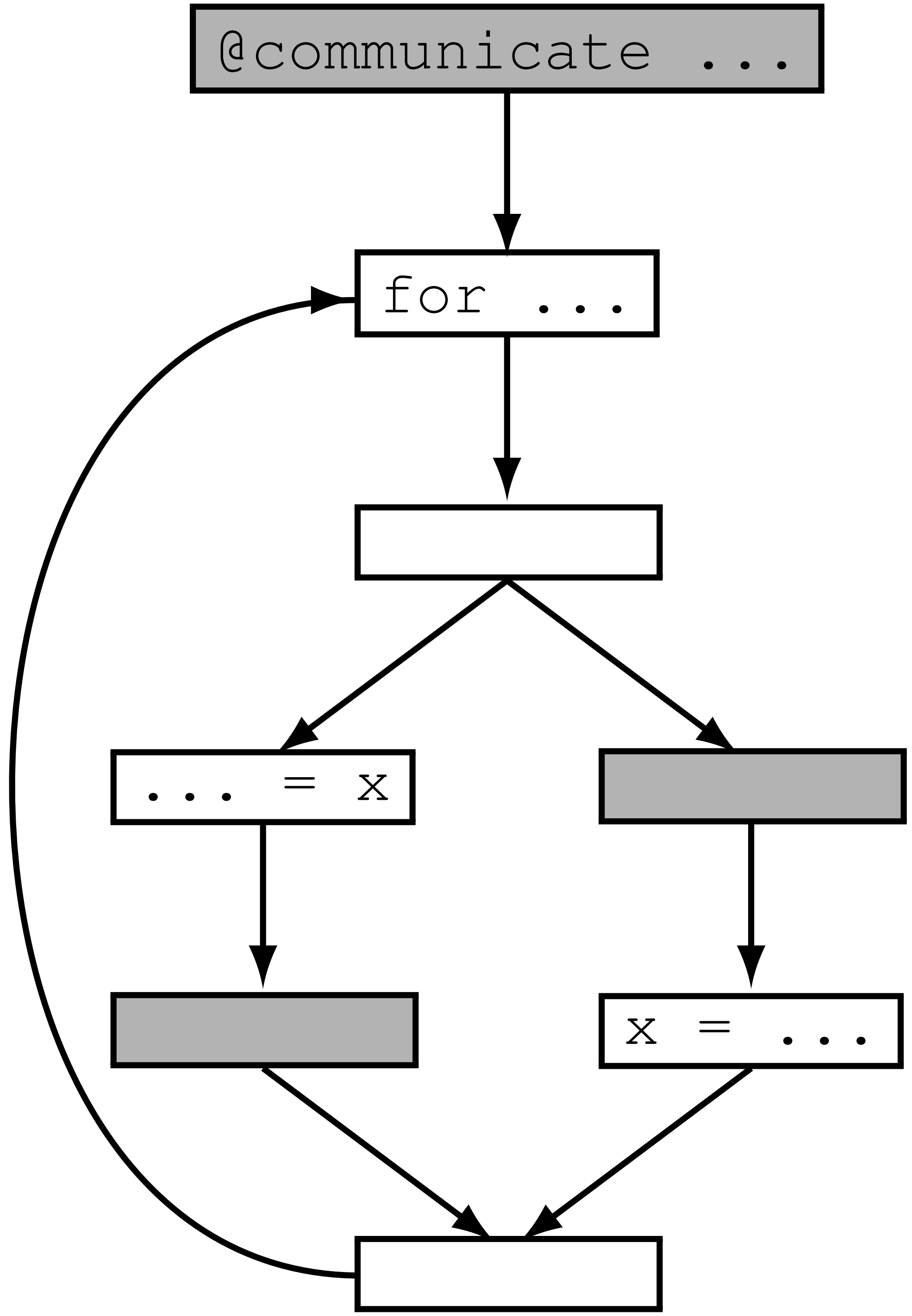
Correctness: Examples



Overall Algorithm

- Identify globalization candidates
- For each globalized variable
  - compute the *locking set*
  - divide the locking set into *connected components*, C_i
    - CFG edge into C_i ⇒ insert `lock_acquire`
    - CFG edge out of C_i ⇒ insert `lock_release`

Example of sub-optimal Behavior



Copy-on-conflict

```
void acquire_or_copy (Buffer& a, Lock& lock)
{
  if (Localized[a]) return NULL;
  Condition cond;
  enum {COPY_THRD, LOCK_THRD} notifier;
  a_cpy = new Buffer;

  Thread l_thrd =
    spawn(acquire_lock, lock, cond, &notifier);
  Thread c_thrd =
    spawn(buf_copy, a, a_cpy, cond, &notifier);
  wait(cond);

  if (notifier == LOCK_THRD) {
    c_thrd.kill();
    free(a_cpy);
  } else {
    l_thrd.kill();
    if (lock.held()) lock.release();
    delete a;
    a = a_cpy;
    Localized[a] = true;
  }
}
```

Copy-on-conflict

```
void acquire_or_copy (Buffer& a, Lock& lock)
{
  if (Localized[a]) return NULL;
  Condition cond;
  enum {COPY_THRD, LOCK_THRD} notifier;
  a_cpy = new Buffer;

  Thread l_thrd =
    spawn(acquire_lock, lock, cond, &notifier);
  Thread c_thrd =
    spawn(buf_copy, a, a_cpy, cond, &notifier);
  wait(cond);

  if (notifier == LOCK_THRD) {
    c_thrd.kill();
    free(a_cpy);
  } else {
    l_thrd.kill();
    if (lock.held()) lock.release();
    delete a;
    a = a_cpy;
    Localized[a] = true;
  }
}
```

Experimental Evaluation

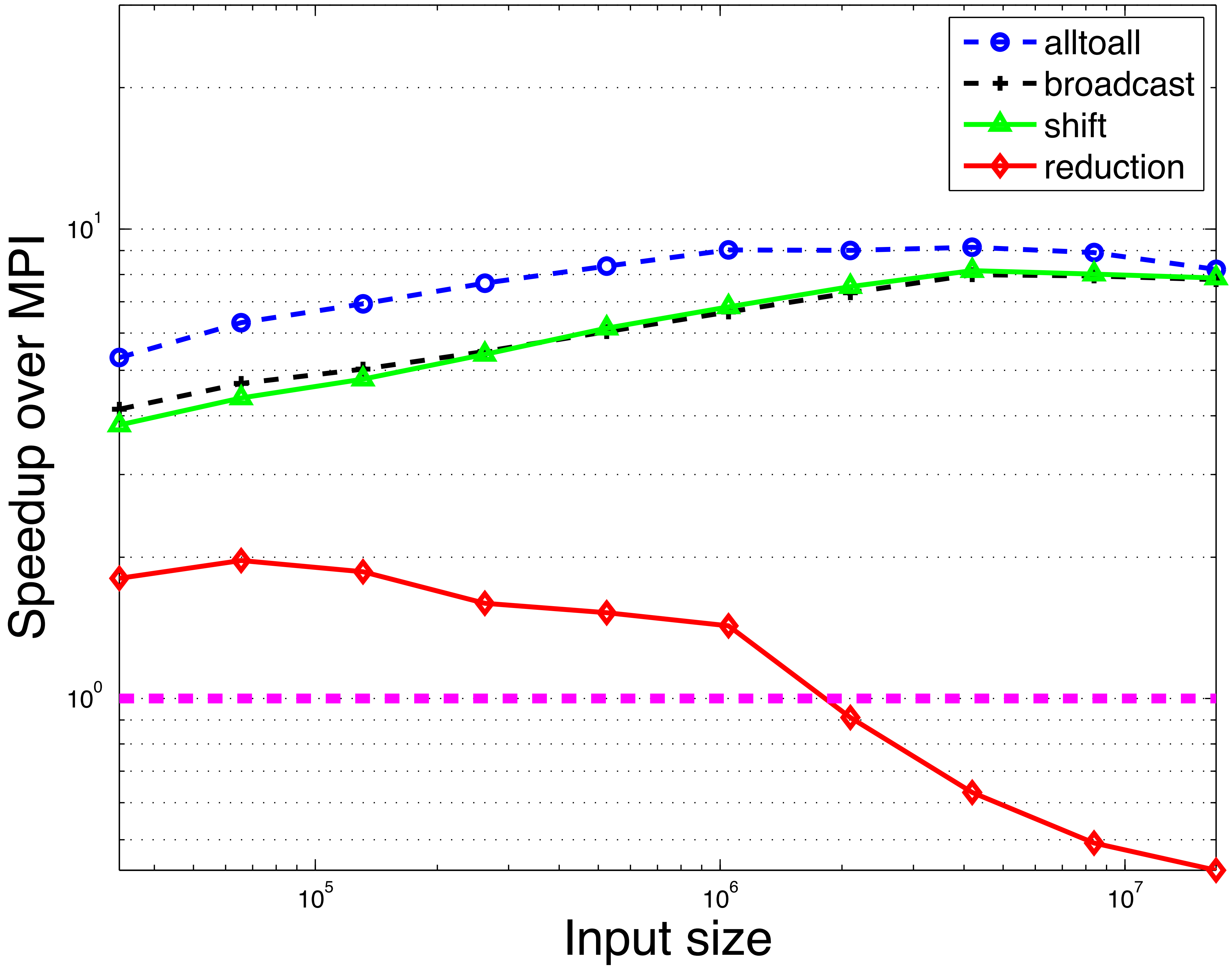
Op	*Kanor*	*MPI*	*Shared Memory*
all	`A[j]@i <<= A[i]@j` `where i,j in WORLD`	`MPI_Alltoall (...)`	`barrier();`
b'cast	`A@i <<= A@0` `where i in WORLD`	`MPI_Bcast(A, ..., ..., 0, ...);`	`barrier();`

shift	`A@i <<= A@i+1`	`if (Rank == (numprocs - 1)) dest = 0;`
		`else dest = Rank + 1;`
		`MPI_Send(A, array_size, ...);`
		`MPI_Recv(A, array_size, ...);`
			`barrier();`

reduce	`A@0 <<op<< A@i`
	`where i in WORLD`
		`MPI_Reduce (...)`
		`// or specialized code for`
		`// tree-reduction of ``op''`
			`// loop for tree-reduction`
			`for (i ...) {`
			`A[i] = op(..);`
			`}`

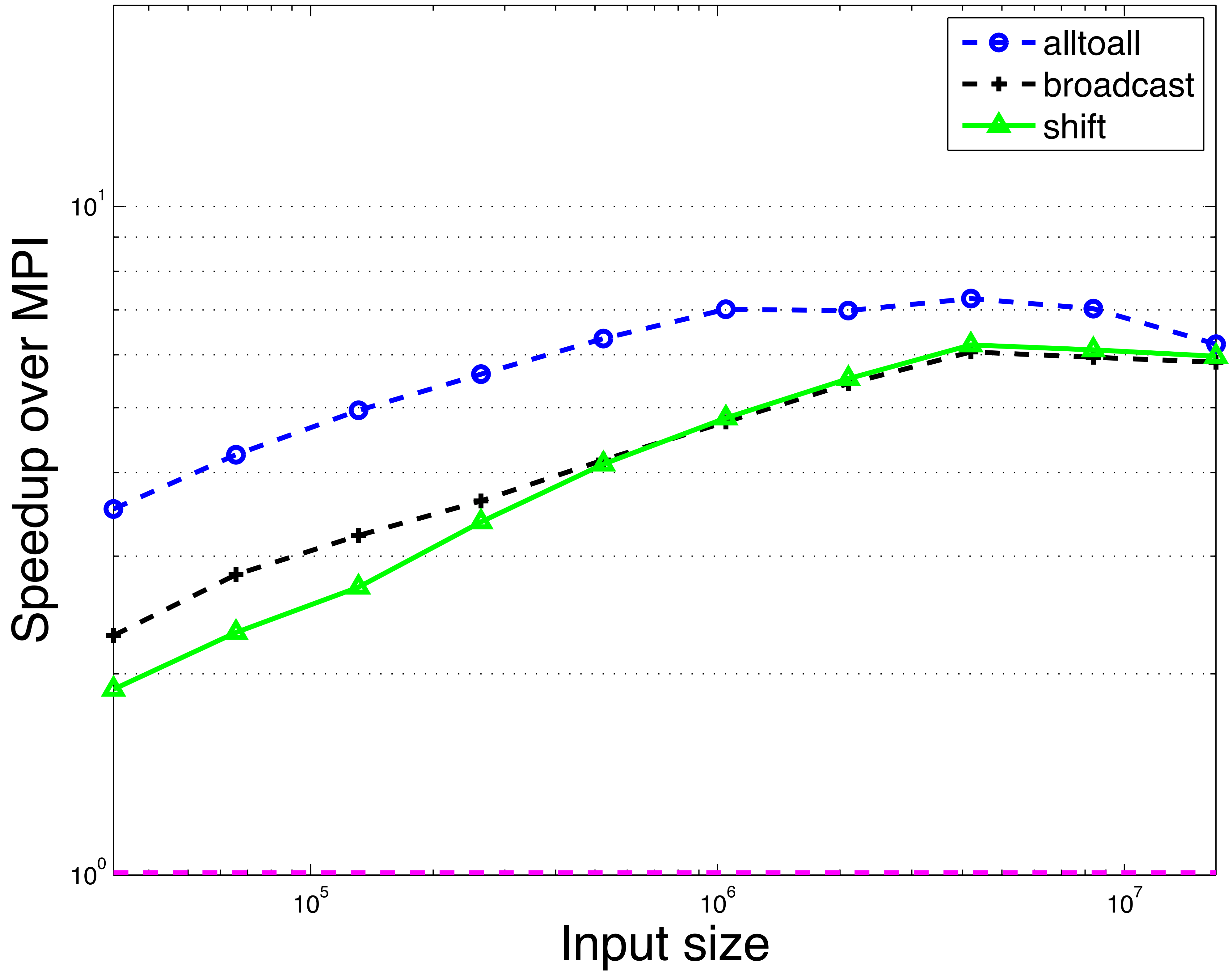
- 8-core AMD Opteron, Gentoo Linux, OpenMPI 1.4.3
- Case 1: No local writes
- Case 2: Local writes
    - 2a: lock successfully acquired
    - 2b: buffer copied locally before the lock could be acquired
- Case 3: Forced copying (overlapping live ranges)

Case I



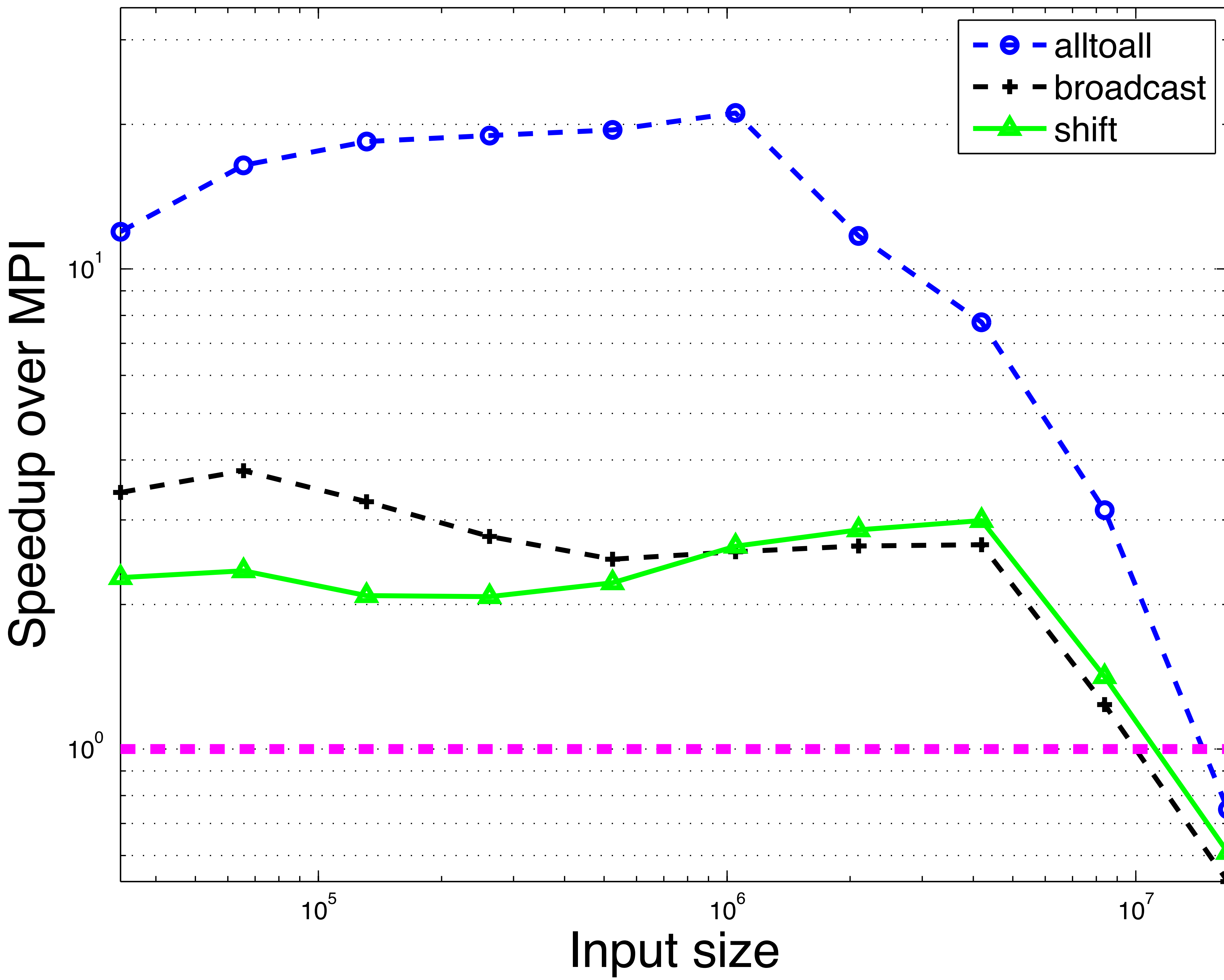
alltoall
broadcast
shift
reduction
Speedup over MPI
Input size
10^1
10^0
10^5
10^6
10^7

Case 2a

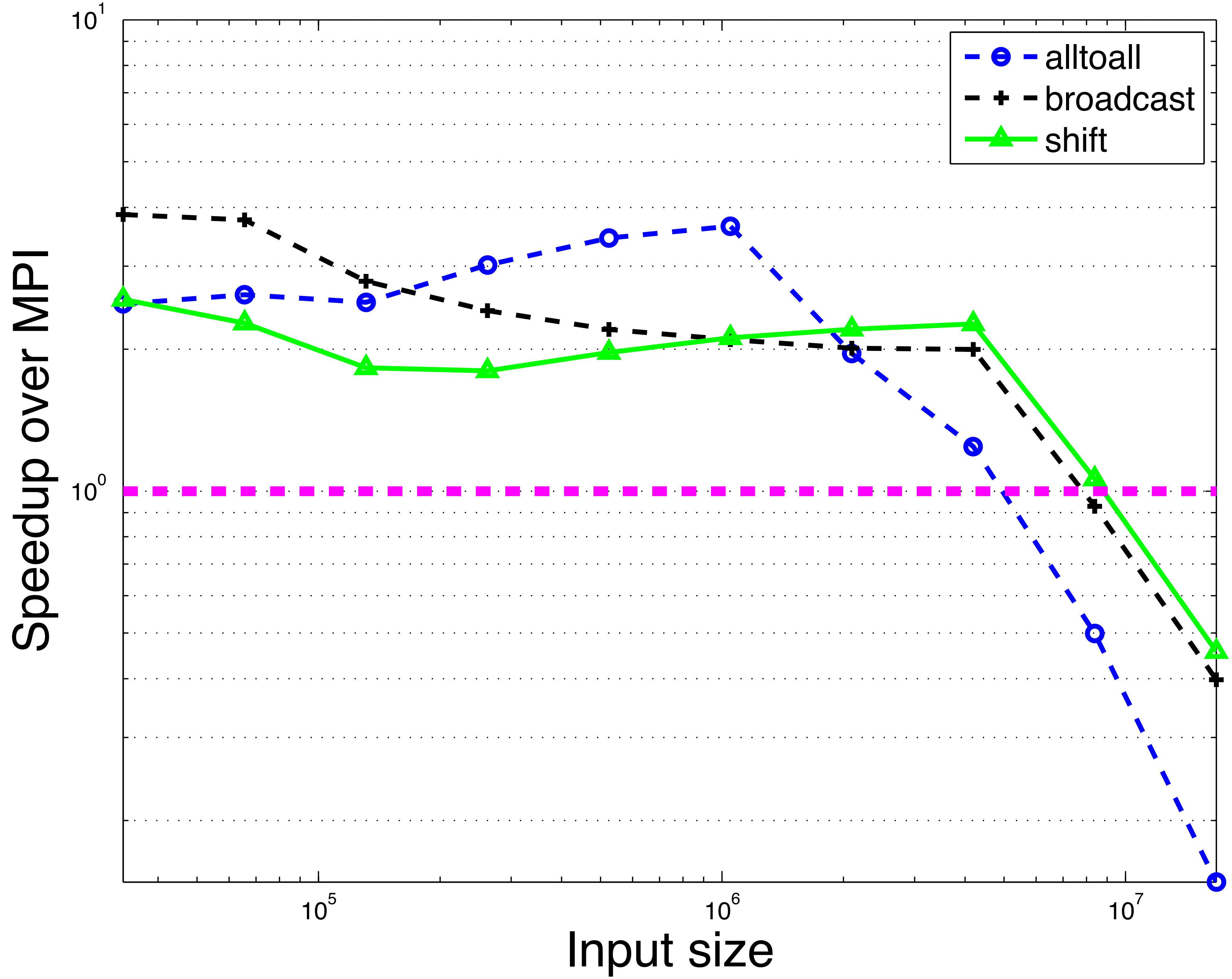


Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

Case 2b


alltoall
broadcast
shift
Speedup over MPI
10^1
10^0
10^5
10^6
10^7
Input size

Case 3


alltoall
broadcast
shift
Speedup over MPI
Input size
10^1
10^0
10^5
10^6
10^7

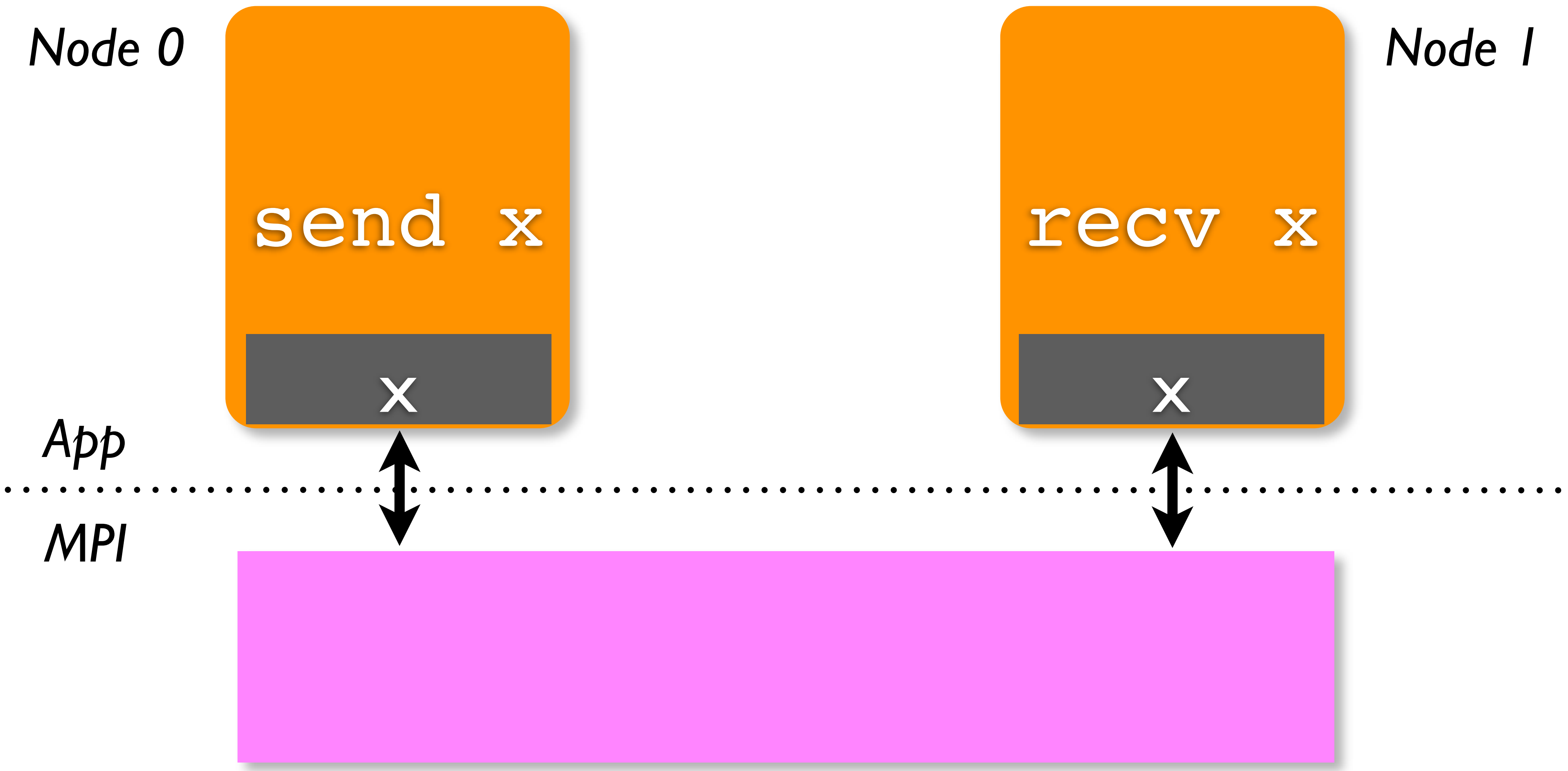
Concluding Remarks

- Parallel programming with partitioned address spaces has advantages
- Appropriate abstraction makes parallel programming more accessible to intermediate-level programmers
  - Kanor demonstrates the effectiveness of this approach
- Advantages of shared memory can be obtained through compiler optimizations
  - our compiler algorithms and experimental evaluation substantiate this claim

End

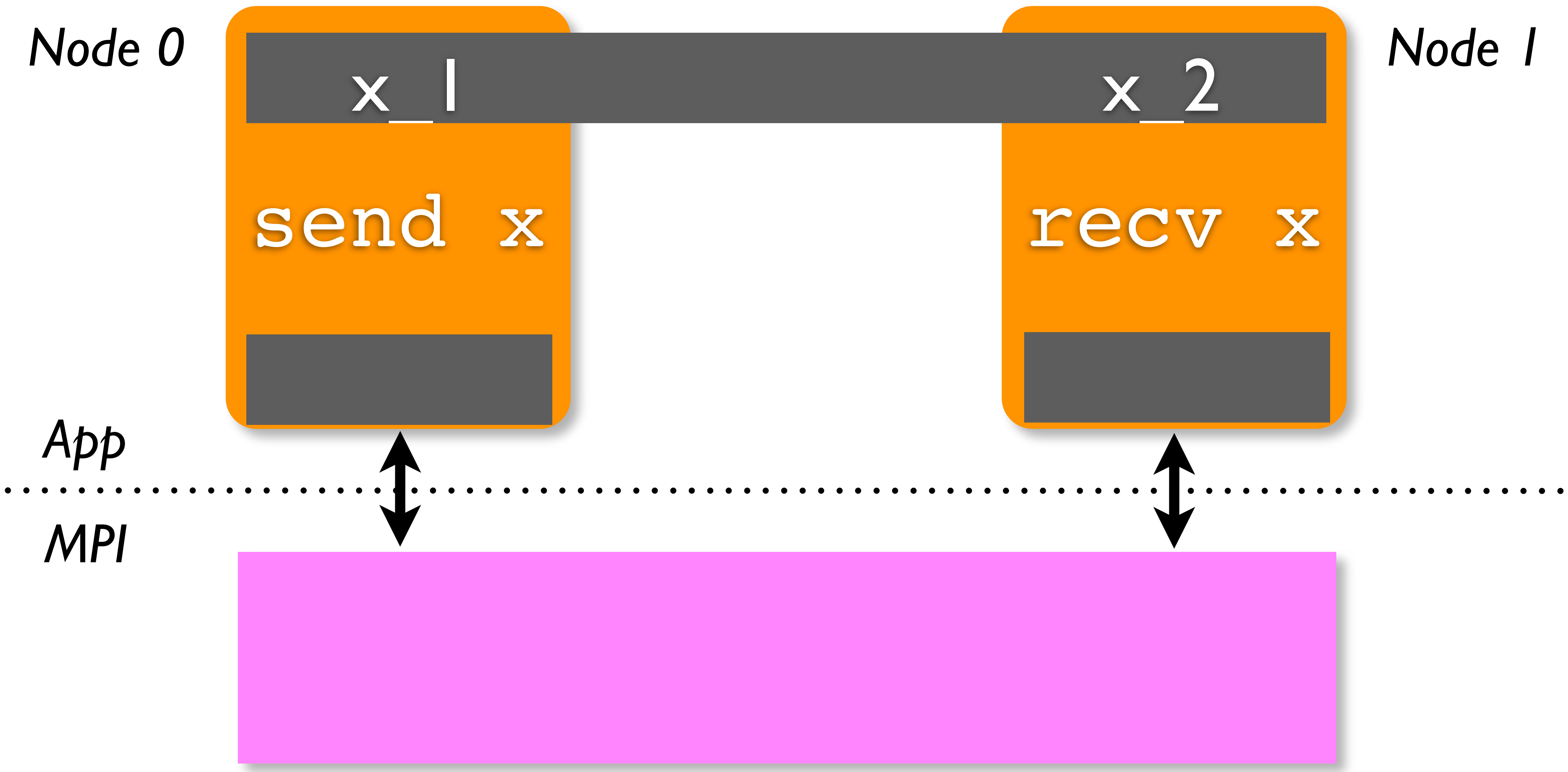
Optimizing for Shared Memory

`@communicate {x@0 <<= x@1}`



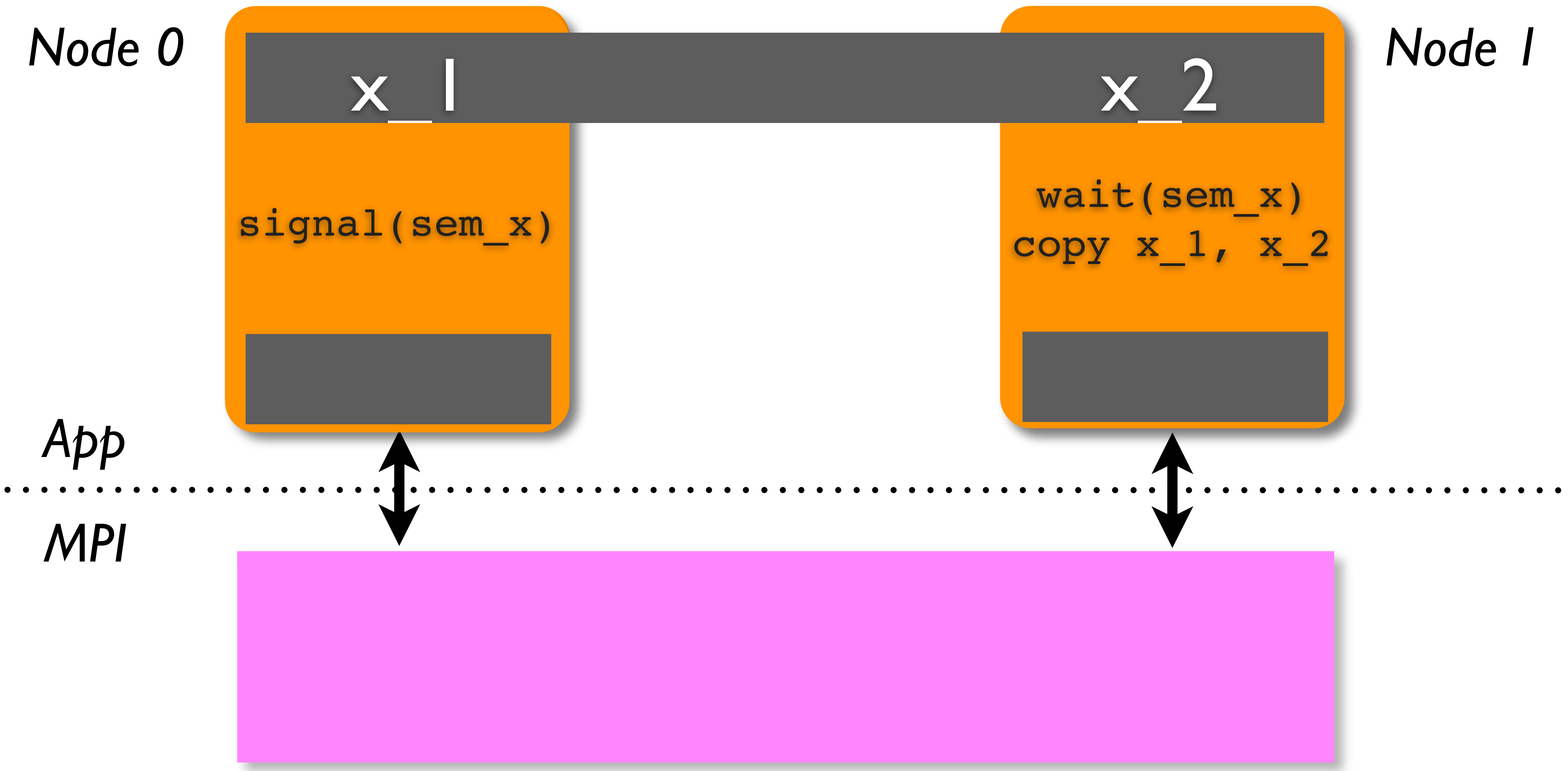
Optimizing for Shared Memory

```
@communicate {x@0 <<= x@1}
```



Optimizing for Shared Memory

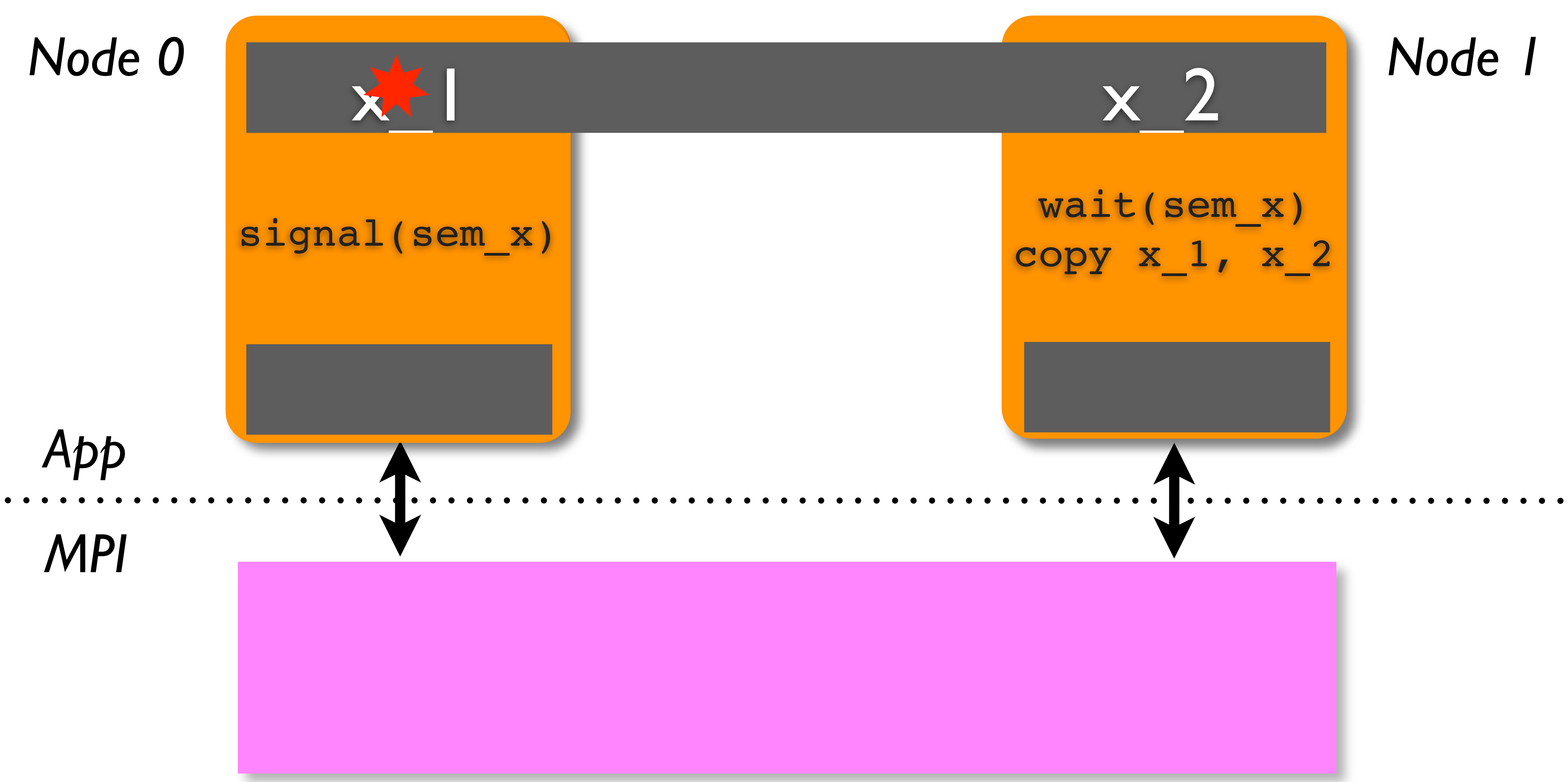
`@communicate {x@0 <<= x@1}`



Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

Optimizing for Shared Memory

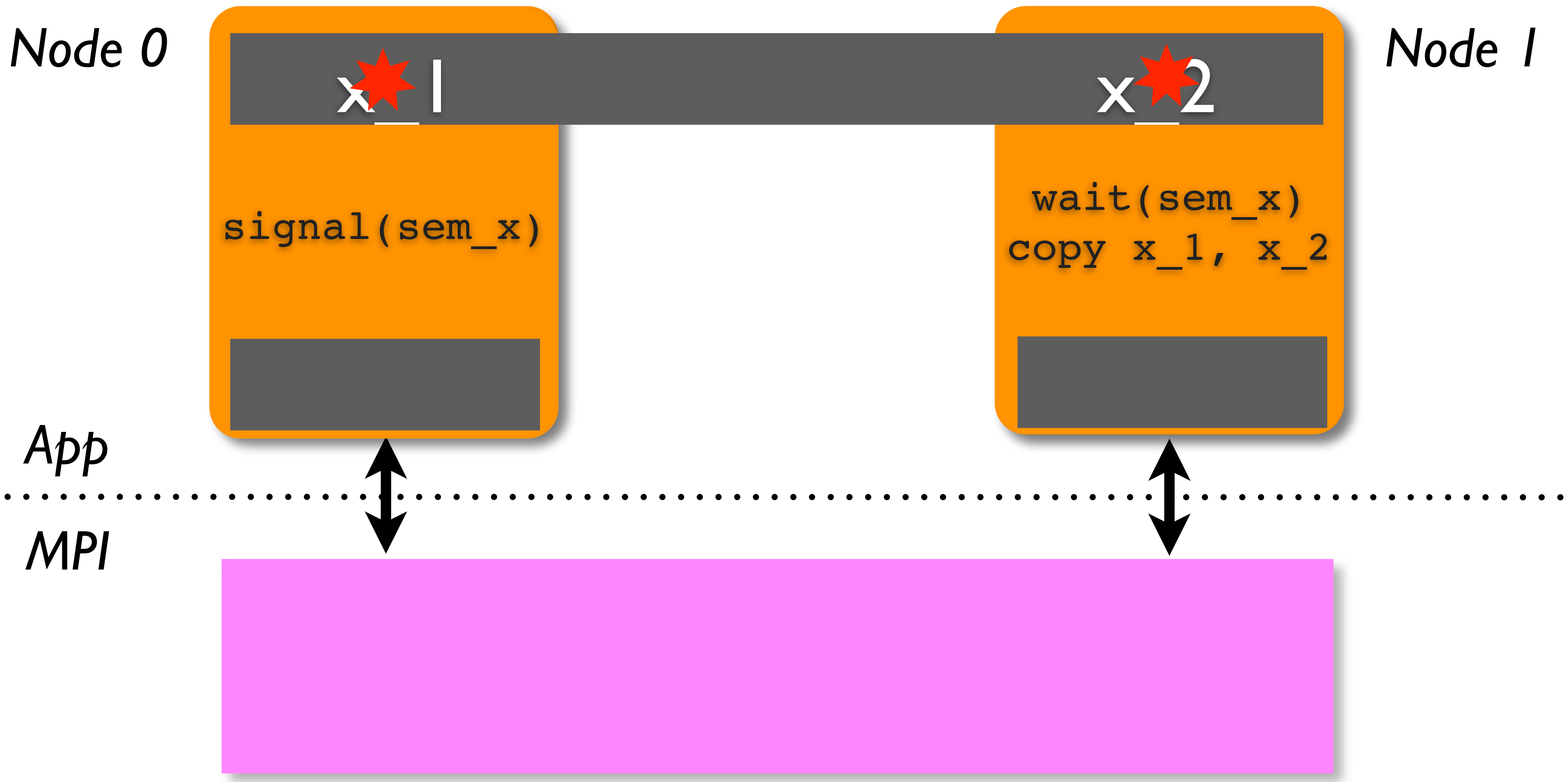
```
@communicate {x@0 <<= x@1}
```



Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

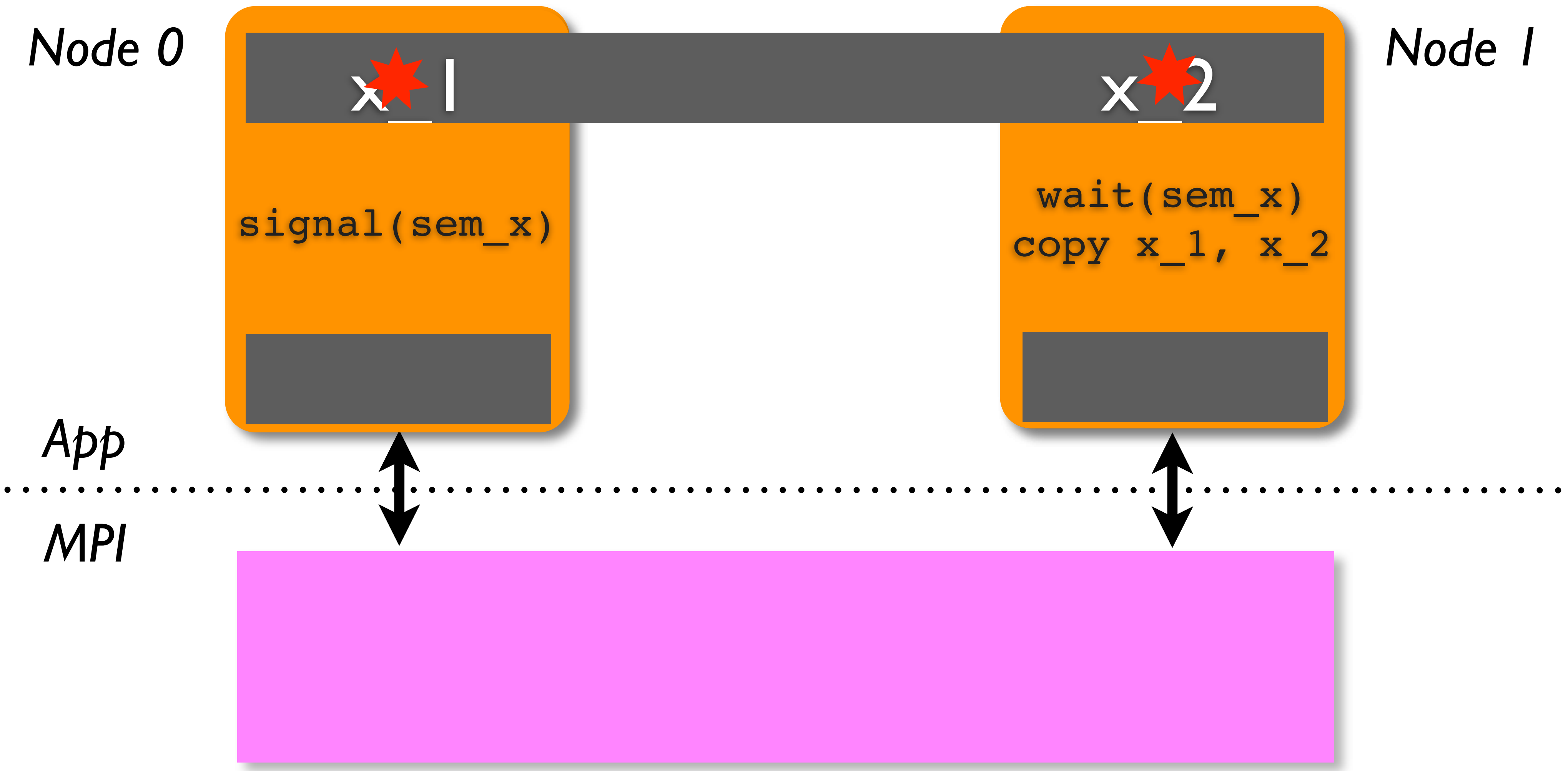
Optimizing for Shared Memory

`@communicate {x@0 <<= x@1}`



Optimizing for Shared Memory

`@communicate {x@0 <<= x@1}`



1 copy

(requires compiler intervention)

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

Computing all Paths from s to t

```
Algorithm: PATHS
Input: Directed graph G(V, E)
       Start node s
       End node t
Output: Set P of nodes that lie on any path from s to t
------------------------------------------------------------
P ← φ
for each node n in G do
    n.color ← "white"
Q ← [s]
while not Q.empty do
    q ← Q.extract
    for each edge (q, v) ∈ E do
        if v.color ≠ "red" then
            v.color ← "red"
            Q.add(v)
Q ← [t]
while not Q.empty do
    q ← Q.extract
    for each edge (v, q) ∈ E do
        if v.color ≠ "black" then
            if v.color = "red" then
                P ← P ∪ {v}
            v.color ← "black"
            Q.add(v)
return P
```

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011

Computing Locking Sets

Algorithm: COMPUTE-LOCKING-SET

Input: CFG $G(V, E)$ of code region over which variable `x` is globalized, with level-annotated nodes; dependence levels, l_x, for dependencies involving `x`; dep. distances, d_x, for dependencies involving `x`;

Output: Locking set L

$L = \phi$

for *each node pair* (w, r) *with an entry in* l_x **do**

  if $d_x(w, r) = 0$ **then**

    if $l_x(w, r) = 0$ **then**

      $L \leftarrow L \cup \text{PATHS}(G, w, r)$

    else

      $G'(V', E') \leftarrow G$ without any looping back-edges at level $l_x(w, r)$ and lower

      $L \leftarrow L \cup \text{PATHS}(G', w, r)$

  else if $d_x(w, r) = 1$ **then**

    $h \leftarrow$ head node of loop at level $l_x(w, r)$

    $G'(V', E') \leftarrow G$ restricted to levels $l_x(w, r)$ and higher

    $L \leftarrow L \cup \text{PATHS}(G', w, h) \cup \text{PATHS}(G', h, r)$

  else

    $G'(V', E') \leftarrow G$ restricted to levels $l_x(w, r)$ and higher

    $L \leftarrow L \cup \text{PATHS}(G', w, r)$

return L

Arun Chauhan, Zero-copy communication in partitioned address space programs on shared memory, HiPC 2011