Concurrent Models of Computation

Edward A. Lee

Robert S. Pepper Distinguished Professor, UC Berkeley

EECS 290n – Advanced Topics in Systems Theory

Concurrent Models of Computation

Spring 2009

Week 4: Message Passing Patterns

Message Passing Interface
MPI

MPI is a collaborative standard developed since the early 1990s with many parallel computer vendors and stakeholders involved.

Realized as a C and Fortran APIs.

First draft of MPI: J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

Anatomy of an MPI Program (in C)

```
/* On each processor, execute the following with different values for rank. */
int main(int argc, char *argv[]) {
    int rank, size, …;

    MPI_Init(&argc, &argv);
    // Find out which process this is (rank)
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // Find out how many processes there are (size)
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
      … code for one process …
    } else if (rank == RAMP2) {
      … code for another process …
   }
    MPI_Finalize();
    return 0;
}
```

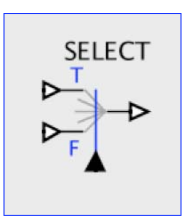
Lee 04: 3

MPI Implementation of Select Process

```
int control;
while (1) {
   MPI_Recv(&control, 1, MPI_INT, CONTROL_SOURCE, ...);
   if (control) {
       MPI_Recv(&selected, 1, MPI_INT, DATA_SOURCE1, ...);
   } else {
       MPI_Recv(&selected, 1, MPI_INT, DATA_SOURCE2, ...);
   }
   MPI_Send(&selected, 1, MPI_INT, DATA_SINK, ...);
}
```

Rank of the source or destination process

Data type of the handled message



Lee 04: 4

Vague MPI Send Semantics

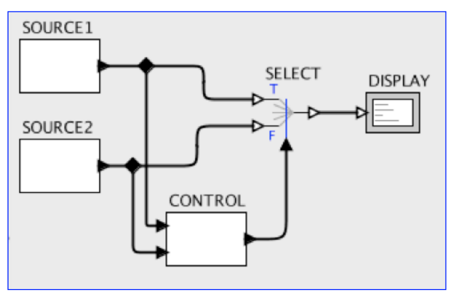
MPI_Send is a "blocking send," which means that it does not return until the memory storing the value to be sent can be safely overwritten. The MPI standard allows implementations to either copy the data into a "system buffer" for later delivery to the receiver, or to rendezvous with the receiving process and return only after the receiver has begun receiving the data.

Discussion: What do you think of this?

You can force a rendezvous style by using MPI_Ssend instead of MPI_Send

Lee 04: 5

What happens to this program under a rendezvous style of communication?



CONTROL Process:

```
MPI_Recv(&data1, 1, MPI_INT, SOURCE1, ...);
MPI_Recv(&data2, 1, MPI_INT, SOURCE2, ...);
while (1) {
  if (someCondition(data1, data2)) {
    MPI_Send(&trueValue, 1, MPI_INT, SELECT, ...);
    MPI_Recv(&data1, 1, MPI_INT, SOURCE1, ...);
  } else {
    MPI_Send(&falseValue, 1, MPI_INT, SELECT, ...);
    MPI_Recv(&data2, 1, MPI_INT, SOURCE2, ...);
  }
}
```

Lee 04: 6

Forcing Buffered Send: MPI_Bsend()

"A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signalled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations."

Message Passing Interface Forum (2008). MPI: A Message Passing Interface standard -- Version 2.1, University of Tennessee, Knoxville, Tennessee.

Irony

"The reluctance of MPI to mandate whether standard sends are buffering or not stems from the desire to achieve portable programs."

Message Passing Interface Forum (2008). MPI: A Message Passing Interface standard -- Version 2.1, University of Tennessee, Knoxville, Tennessee.

Buffer Size Control in MPI

MPI_Buffer_attach associates a buffer with a process. Any output can use the buffer, and MPI does not limit the buffering to the specified buffers.

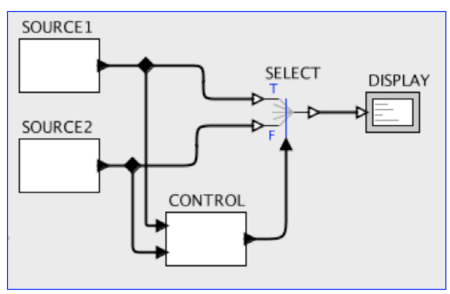
The MPI_Send procedure can return an error, so you can write processes that do something when buffers overflow. What should they do?

MPI provides few mechanisms to exercise control over the process scheduling (barrier synchronization seems to be about it).

A Design Question: How to accomplish the fork processes?

Option 1:
- Create a process for each fork that copies inputs to outputs (in what order?)

Option 2:
- Modify the SOURCE processes to do successive writes to SELECT and CONTROL (in what order?).



MPI_Recv Semantics

MPI_Recv blocks until the message is received.

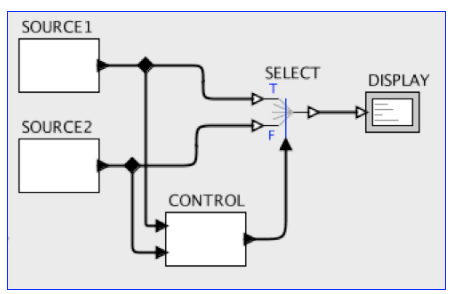
Communication is point-to-point: Sending and receiving processes refer to each other. According to the MPI standard: “[this] guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard MPI_ANY_SOURCE is not used in receives.”

MPI_ANY_SOURCE can be specified in a MPI_Recv()

Messages arrive in the same order sent.

Discussion: Suppose you wanted to implement Parks’ algorithm or Geilen and Basten?

How would you do it?



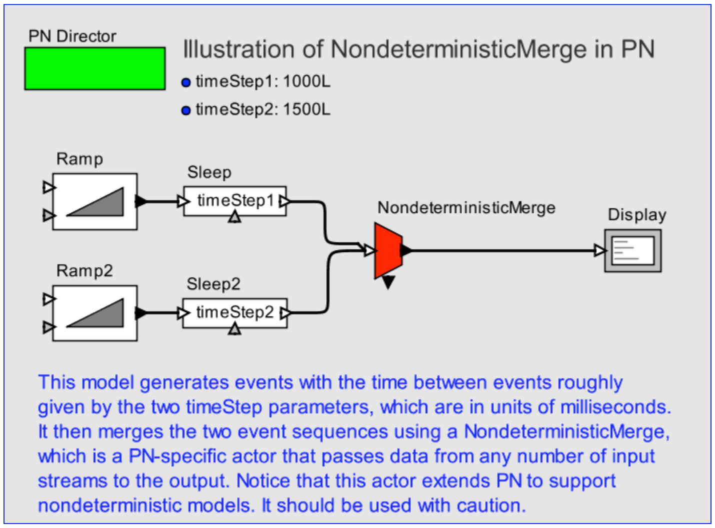
Threads and Fairness

MPI is used sometimes with threads, where a single process runs in multiple threads. This can

"Fairness MPI makes no guarantee of fairness in the handling of communication. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send, yet the message is never received, because it is each time overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multi-threaded process. Then it is possible that messages that match this receive are repeatedly received, yet the receive is never satisfied, because it is overtaken by other receives posted at this node (by other executing threads). It is the programmer's responsibility to prevent starvation in such situations."

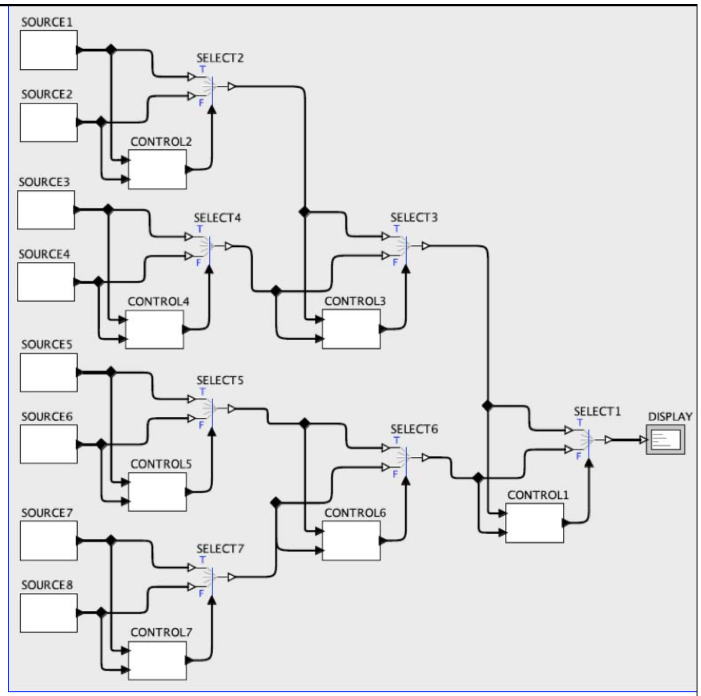
NondeterministicMerge in Ptolemy II is implemented in a multithreaded actor

Two threads perform blocking reads on each of two input channels and write to the same output port.



Scaling Up Designs

Collective operations enable compact representations of certain composite structures (not this one though, at least not in MPI).

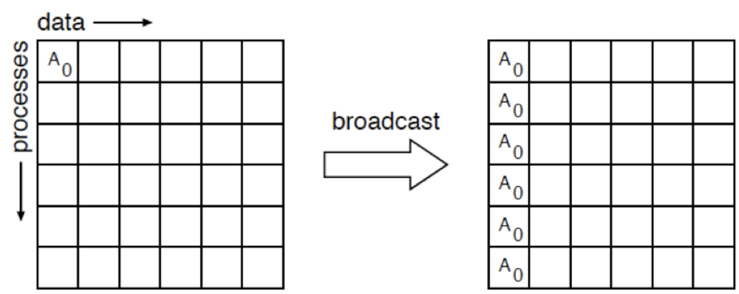


Collective Operations Provided by MPI

- Barrier synchronization in a group
- Broadcast to a group
- Gather from a group (to one member or all members)
- Scatter to a group
- Scatter/Gather all-to-all
- Reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all group members or one member
- Combined reduction and scatter operation
- Scan across all members of a group (also called prefix)

Message Passing Interface Forum (2008). MPI: A Message Passing Interface standard -- Version 2.1, University of Tennessee, Knoxville, Tennessee.

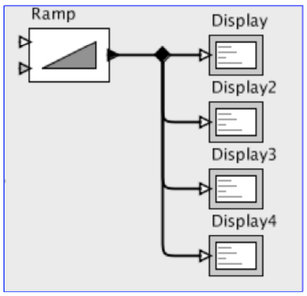
Broadcast

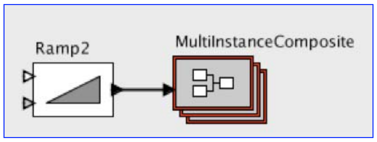


MPI_Bcast is like MPI_Send except that it sends to all members of the group. Data are copied.

Broadcast in Ptolemy II

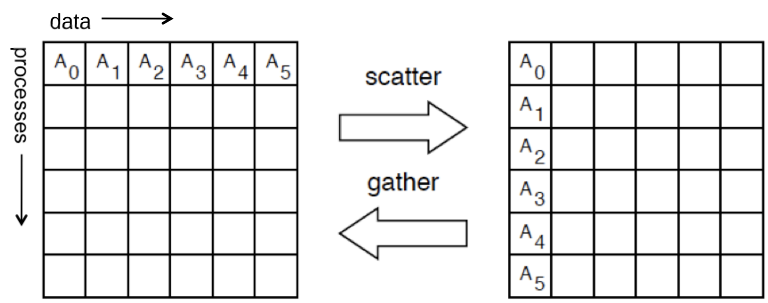
Choice of director defines the communication policy.





Message Passing Interface Forum (2008). MPI: A Message Passing Interface standard -- Version 2.1, University of Tennessee, Knoxville, Tennessee.

Gather/Scatter



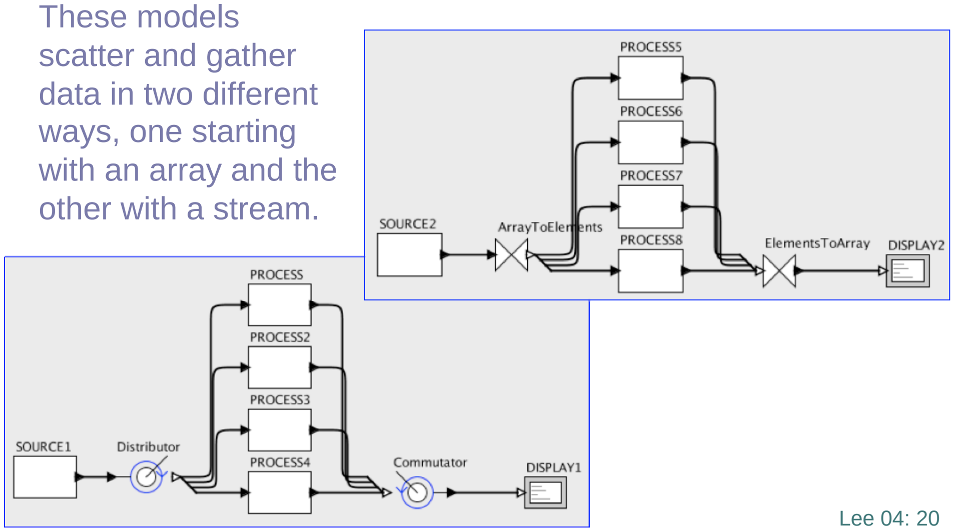
E.g., For gather, processes execute

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
recvcount, recvtype, receivingProcessID, communicator);
```

At the receiving process, this results in recvbuf getting filled with items sent by each of the processes (including the receiving processes).

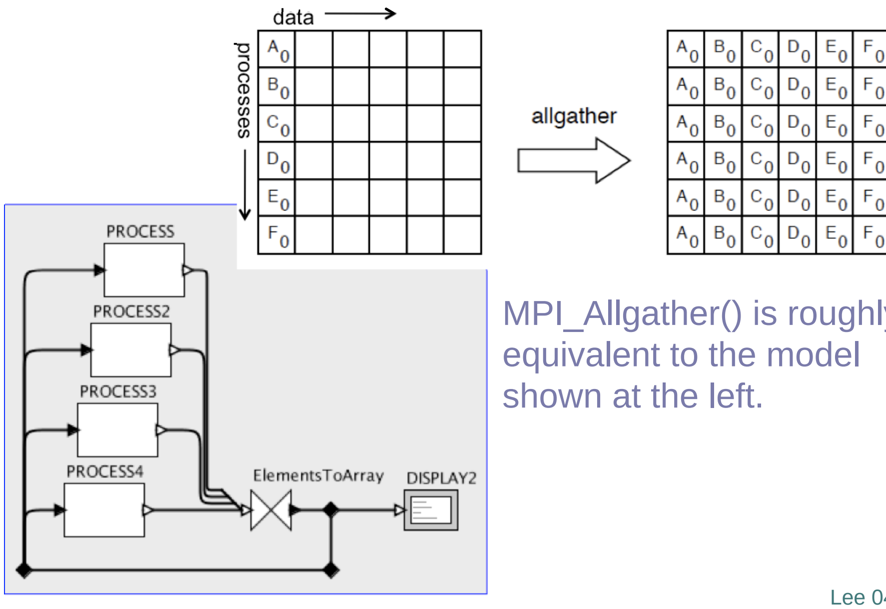
Ptolemy II Mechanisms a bit like Gather/Scatter

These models scatter and gather data in two different ways, one starting with an array and the other with a stream.



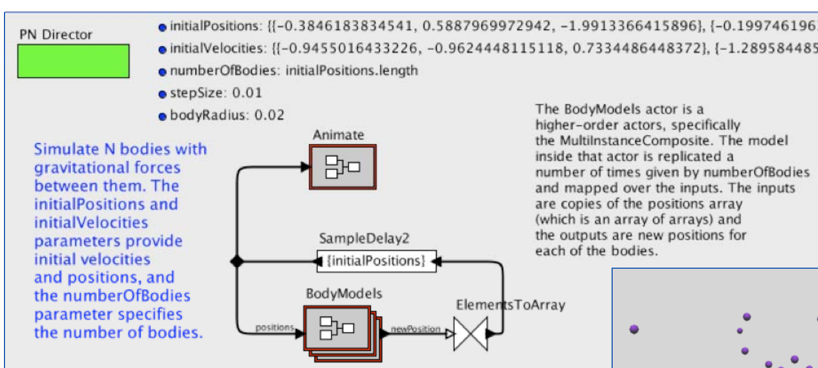
Message Passing Interface Forum (2008). MPI: A Message Passing Interface standard -- Version 2.1, University of Tennessee, Knoxville, Tennessee.

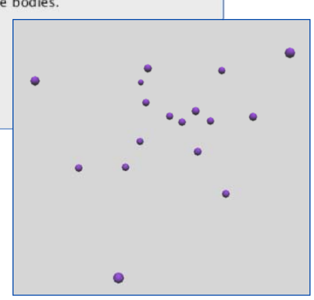
Gather to all



MPI_Allgather() is roughly equivalent to the model shown at the left.

Application of Gather to All: Gravitation Simulation

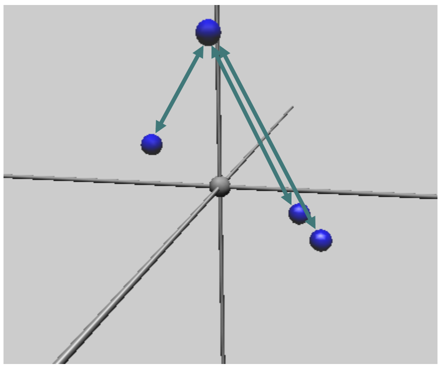


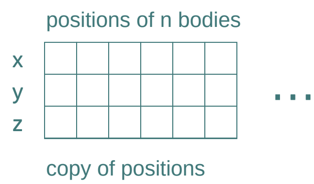


Here, BodyModels receives an array of positions and broadcasts it to one process per body. Each process computes the position of the body at the next time step and the ElementsToArray gathers these into an array.

How the Gravitation Simulation is a Gather-to-all Pattern

3-D gravitational simulation of *n* bodies

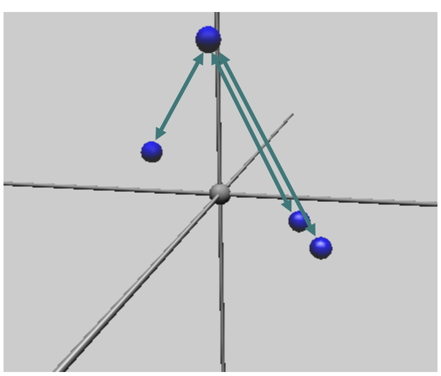


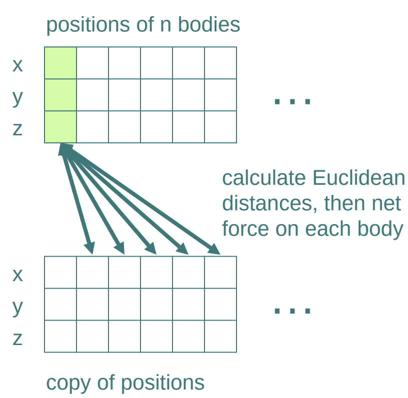


Thanks to Rodric Rabbah, IBM Watson Center, for suggesting this example.

How the Gravitation Simulation is a Gather-to-all Pattern

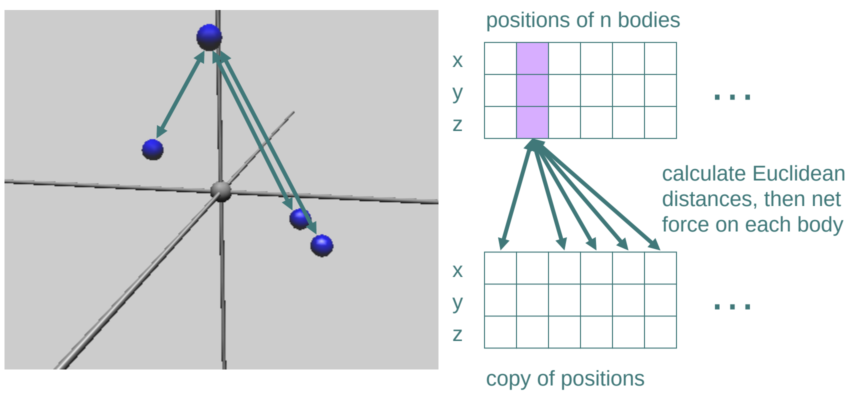
3-D gravitational simulation of *n* bodies





How the Gravitation Simulation is a Gather-to-all Pattern

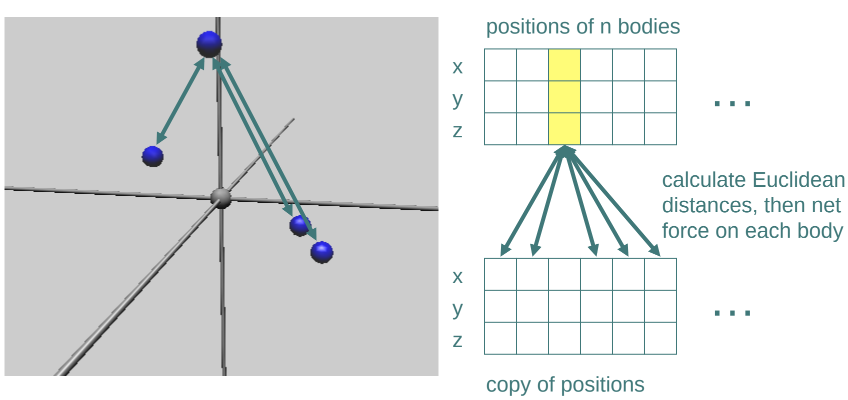
3-D gravitational simulation of *n* bodies



Lee 04: 25

How the Gravitation Simulation is a Gather-to-all Pattern

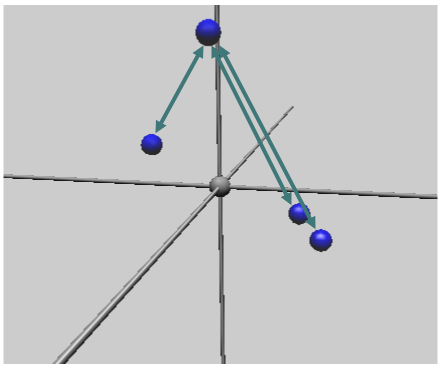
3-D gravitational simulation of *n* bodies



Lee 04: 26

How the Gravitation Simulation is a Gather-to-all Pattern

3-D gravitational simulation of *n* bodies



$$F(t) = ma(t)$$
$$a(t) = F(t)/m$$
$$v(t) = \int_0^t a(\tau)d\tau + v(0)$$
$$p(t) = \int_0^t v(\tau)d\tau + p(0)$$

A simple (naïve) approximation:

$$v(t+\Delta) = v(t) + \Delta a(t)$$
$$p(t+\Delta) = p(t) + \Delta v(t)$$

Each process computes this approximation.

Message Passing Interface Forum (2008). MPI: A Message Passing Interface standard -- Version 2.1, University of Tennessee, Knoxville, Tennessee.

All to all Gather/Scatter

data →

processes ↓

A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5

alltoall ⇒

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

Exercise: Realize this pattern in Ptolemy II.

Reduction Operations

Reduce operations gather data from multiple processes and reduce them using an associative operation (like sum, maximum, …). The operation need not be commutative. The order of reduction is by process ID (called “rank” in MPI).

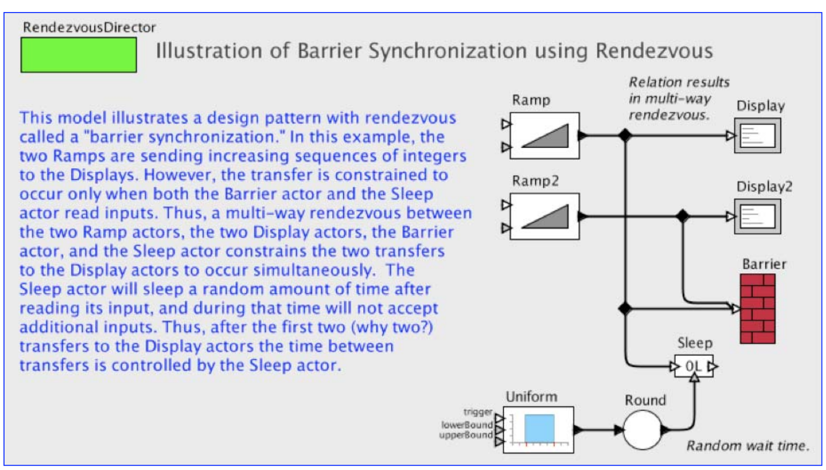
Result may be returned to one process or to all.

E.g.,

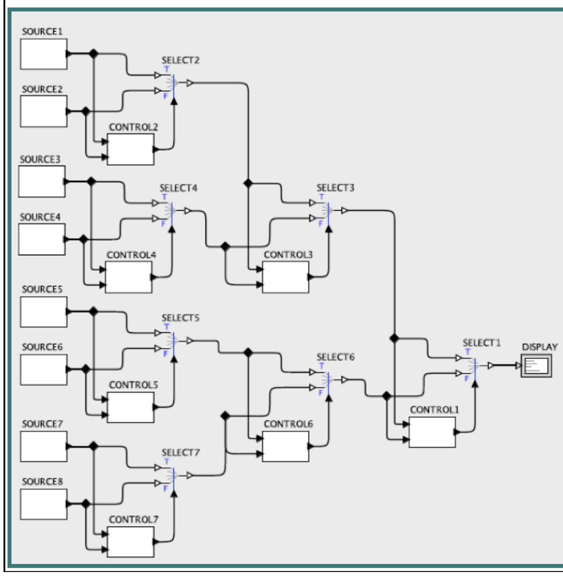
```
MPI Reduce(sendbuf, recvbuf, count, type, operation
receivingProcessID, communicator);
```


A Rather Different MPI Pattern: Barrier Synchronization

MPI_Barrier() blocks until all members of a group have called it. Ptolemy II equivalent uses the Rendezvous director:



Not provided Directly by MPI: Sorting Trees

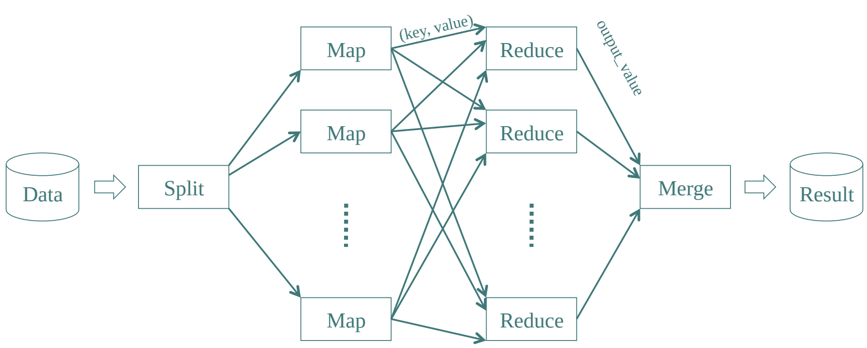


Consider collecting time-stamped trades from commodities markets around the world and merging them into a single time-stamped stream. The CONTROL actors could compare time stamps, with logic like this:

```
data1 = topPort.get();
data2 = bottomPort.get();
while (true) {
  if (data1.time < data2.time)) {
    output.send(true);
    data1 = topPort.get();
} else {
    output.send(false);
    data2 = bottomPort.get();}
}
```

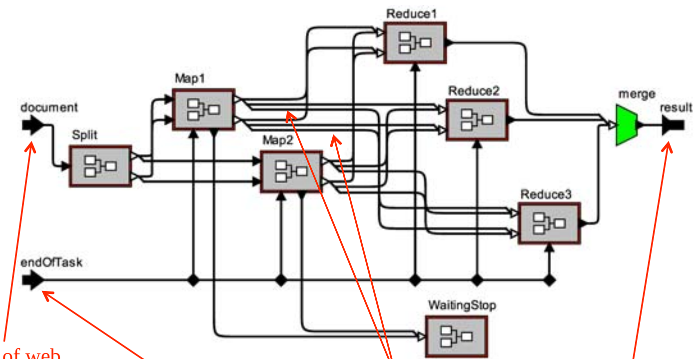

Not provided directly by MPI: Map/Reduce

Dean, J. and S. Ghemawat (2004). {MapReduce}: Simplified Data Processing on Large Clusters. Symposium on Operating System Design and Implementation (OSDI).



This pattern is intended to exploit parallel computing by distributing computations that fit the structure. The canonical example constructs an index of words found in a set of documents.

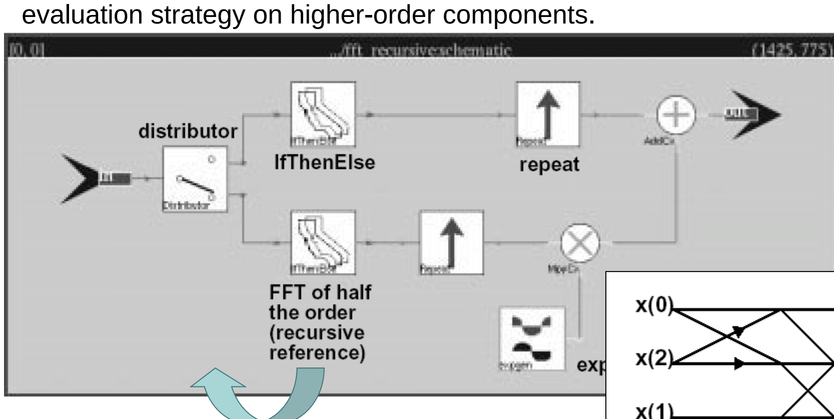
A MapReduce Model in Ptolemy II



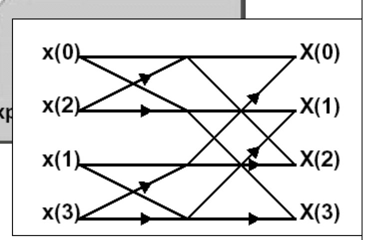
Inputs of web documents	End of all documents	Word-value pairs	Merged word-counting outputs
`"contents of the first document"`	`false`	`{"contents", 1}`	`{"contents", 3}`
`"contents of the second document"`	`false`	`{"of", 1}`	`{"first", 1}`
`...`	`...`	`...`	`...`
`"contents of the last document"`	`true`	`{"document", 1}`	`{"document", 3}`

Not provided by MPI: Recursion

FFT implementation in Ptolemy Classic (1995) used a partial evaluation strategy on higher-order components.



recursive reference



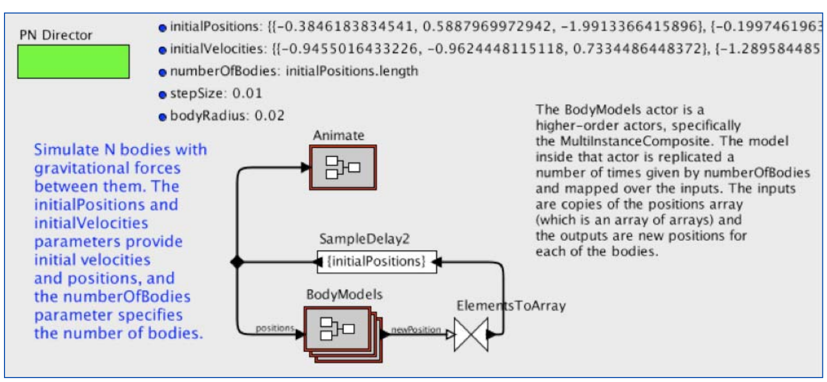
Not provided by MPI: Dynamically Instantiated Processes

```
Process SIFT in QI => QO;
  Vars PRIME;
  repeat
    GET(QI) → PRIME; PUT(PRIME,QO)
    doco FILTER(PRIME,QI)→QI; CONTINUE closeco
  forever
Endprocess;
```

Recall Kahn & MacQueen (1977). Above, a new instance of FILTER is spliced into the pipeline ahead of this process each time a new input arrives.

Kahn, G. and D. B. MacQueen (1977). Coroutines and Networks of Parallel Processes. Information Processing, North-Holland Publishing Co.

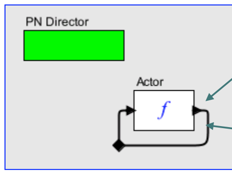
Patterns as Higher-Order Components



BodyModels here is an instance of MultiInstanceComposite, an actor in Ptolemy II that has two parameters: one specifying the number of instances, and one specifying the model to instantiate. This is a “higher-order-component” because it operates on components, not just data.

Reexamining Kahn MacQueen Blocking Reads

or “do we need MPI_Probe()?”



type A

sequence in A^{**}

Recall: Semantics of a PN Model is the Least Fixed Point of a Monotonic Function:

- Chain: $C = \{ f(\bot), f(f(\bot)), \ldots, f^n(\bot), \ldots\}$
- Continuity: $f(\vee C) = \vee \hat{f}(C)$

Limits

Kahn-MacQueen Blocking Reads vs. Kahn Continuity

Following Kahn-MacQueen [1977], actors are threads that implement *blocking reads*, which means that when they attempt to read from an empty input, the thread stalls.

- This restricts expressiveness more than continuity

PN Implementation in Ptolemy II

Body of a process:

```
while (!stopRequested()) {
  …
  if (inputPort.hasToken(channelNo)) {
    …
    Token input = inputPort.get(channelNo);
    …
  }
}
```

When using the PN Director, hasToken() always returns true. Why?

Blocking reads realize *sequential* Functions [Vuillemin]

Let $f : A^n \to A^m$ be an n input, m output function.

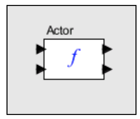
Then f is *sequential* if it is continuous and for any $a, b \in A^n$ where $a \leq b$ there exists an $i \in \{1, \ldots n\}$, where:

$$a\,|_{\{i\}} = b\,|_{\{i\}} \Rightarrow f(a) = f(b)$$

Intuitively: At all times during an execution, there is an input channel that blocks further output. This is the Kahn-MacQueen blocking read!

Continuous Function that is not Sequential

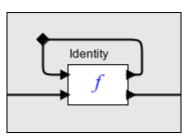
Two input identity function is not sequential:



Let $f : A^2 \to A^2$ such that for all $a \in A^2$, $f(a) = a$.
Then f is not sequential.

Cannot Implement the Two-Input Identity with Blocking Reads

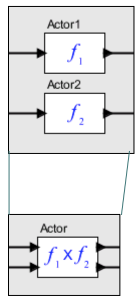
Consider the following connection:



This has a well-defined behavior, but an implementation of the two-input identity with blocking reads will fail to find that behavior.

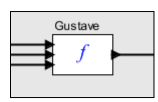
Sequential Functions do not Compose

If $f_1 : A \to B$ and $f_2 : C \to D$ are sequential then $f_1 \times f_2$ may or may not be sequential. Simple example: suppose f_1 and f_2 are identity functions in the following:



Lee 04: 43

Gustave Function
Non Sequential but Continuous



Let $A = T^{**}$ where $T = \{t, f\}$.

Let $f : A^3 \to N^{**}$ such that for all $a \in A^3$,

$$f(a) = \begin{cases} (1) & \text{if } ((t),(f),\bot) \sqsubseteq a \\ (2) & \text{if } (\bot,(t),(f)) \sqsubseteq a \\ (3) & \text{if } ((f),\bot,(t)) \sqsubseteq a \end{cases}$$

This function is continuous but not sequential.

Lee 04: 44

Linear Functions [Erhard]

Function $f: A \rightarrow B$ on CPOs is *linear* if for all joinable sets $C \subseteq A$, $\hat{f}(C)$ is joinable and

$$\vee \hat{f}(C) = f(\vee C)$$

Intuition: If two possible inputs can be extended to a common input, then the two corresponding outputs can be extended to the common output.

Fact: Sequential functions are linear.

Fact: Linear functions are continuous (trivial)

Lee 04: 45

Stable Functions [Berry]

Function $f: A \rightarrow B$ on complete semilattices (CPOs where every subset has a greatest lower bound) is *stable* if it is continuous and for all joinable sets $C \subseteq A$, $\hat{f}(C)$ is joinable and

$$\wedge \hat{f}(C) = f(\wedge C)$$

← NOTE: meet! not join!

Intuition: If two possible inputs do not contain contradictory information, then neither will the two corresponding outputs.

Fact: Sequential functions are stable.

Lee 04: 46

Summary

- MPI is an underspecified standard (buffering issues)
- MPI programs are not modular
- Collective operations in MPI are useful
- There are useful collective operations not specified in MPI
- Collective operations can be viewed as higher-order components.
- Constraint to blocking reads makes process networks non-compositional.
- Constraint to blocking reads precludes implementing certain continuous functions (but are any of those useful?)

Lee 04: 47