COMP90025 Parallel and Multicore Computing

Communication

Aaron Harwood

School of Computing and Information Systems
The University of Melbourne

2019 Semester II

Communication

Distributed memory architectures require message passing and this in turn leads to communication patterns. There are common communication patterns that support widely used parallel algorithms.
While parallel algorithm complexity is a governed by the machine's architecture upon which it executes, there are further aspects that can be considered, how messages are transmitted through the machine and what impact this can have on performance. These considerations need to address the communication patterns that are required by parallel algorithms.
Shared memory parallel programs don't explicitly use communication patterns though the underlying machine may, in the case of a distributed shared memory architecture.

A *message* is defined as a logic unit or packet of information. Messages may be instructions, data or control signals such as for synchronization.

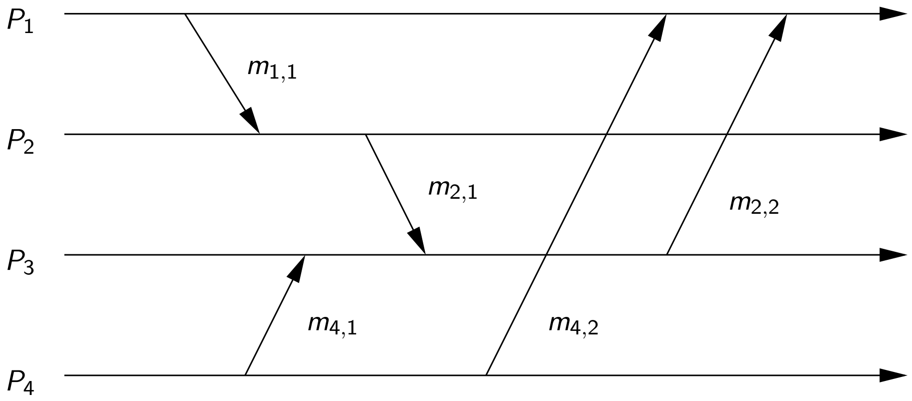

Figure: Communication between 4 processes or tasks.

Time to send a message

The time to send a message can be broken into a fixed tartup time, t_f, and a time per byte, t_b. The time to send the message is then

$$t_m = t_f + t_b\, l$$

where l is the number of bytes in the message.

The time t_f includes the time for the communication subsystem to prepare to send the message. This may include the time for the receiver to indicate that it is ready to receive the data, depending on the send/receive model being used.

- e.g., TPC's SYN + SYN/ACK + ACK 3-way handshake

For efficient message passing it is preferable to have $t_b\, l >> t_f$.

Total parallel run time

The parallel run time, t_p, is a combination of computation time, t_{comp}, and communication time, t_{comm}:

$$t_p = t_{comp} + t_{comm}.$$

For this, both times need to be in the same units. We usually convert t_p into seconds but sometimes we can convert t_{comm} into an equivalent number of computational steps.
The speedup can then be in the form:

$$S = \frac{t_s}{t_p} = \frac{t_s}{t_{comp} + t_{comm}}.$$

Delay hiding

- Communication and computation can happen at the same time
- Part of the art of parallel programming is making sure that all processes have enough data to keep them busy.
- It sometimes helps to forward results incrementally as they are generated
    - May incur extra fixed set-up costs, t_f

Granularity

The total time spent for computations divided by the total time spent communicating is the *granularity* of a task.

$$\text{granularity} = \frac{\text{computation time}}{\text{communication time}}$$

If the granularity is unity then the task spends half of its time in computations and half of its time communicating. Increasing the granularity can be done by decreasing the distribution, i.e. combining tasks into a single task; but this leads to decreased parallelism. A single task spends all of its time in computation and no communication time - apart from the initial communication to obtain the input parameters and the final communication to provide the result.

Communication primitives

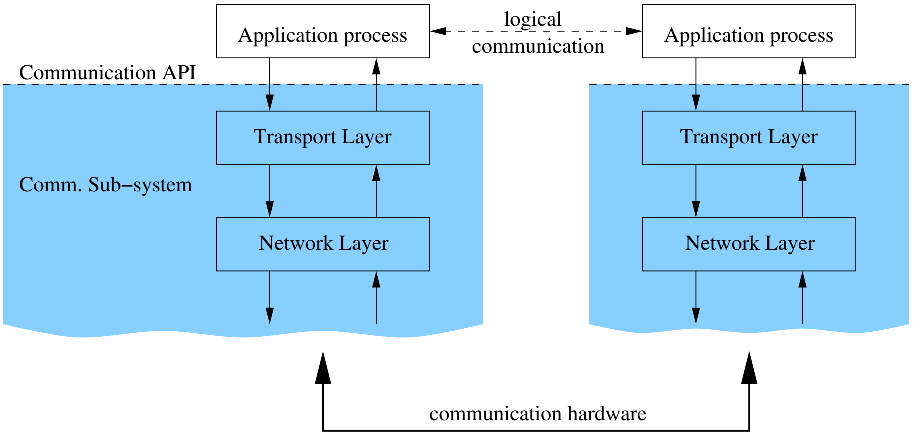

Figure: Generalized communication between two application processes.

The basic communication APIs consists of `send()` and `receive()` primitives. A sender uses `send()` to send data to a receiver that uses `receive()` to receive the data.

Details of the primitive arguments will vary, but the following arguments are common:

- destination identifier,
- message type,
- data type and contents.

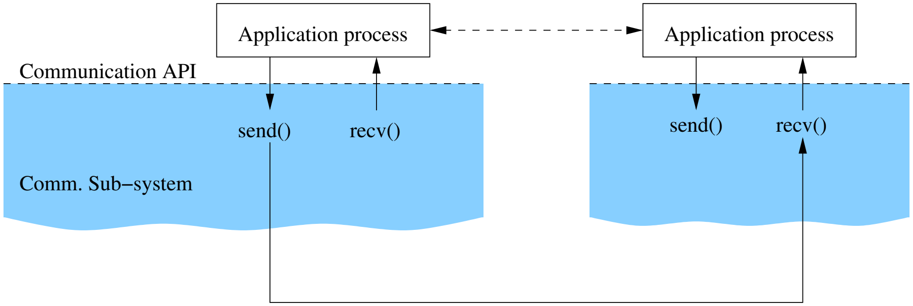

Figure: Send and receive primitives.

Function calls associated with sending and receiving data are not usually provided as separate threads. This means that the caller will *block* until the function returns.

In a general sense functions are called *blocking* or *non-blocking*. A blocking function will not return until the service has been completed. A non-blocking function will return immediately, without requiring the service to be completed.

There are various assumptions made when deciding when a service has been "completed". In general, the application itself may block if there are insufficient resources to continue.

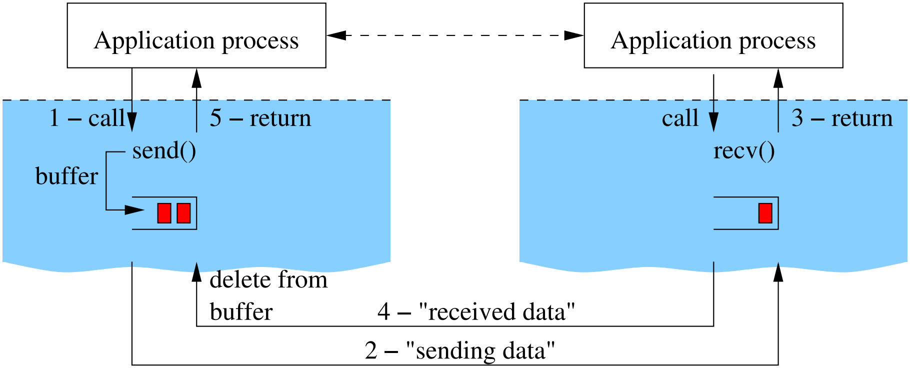

Figure: Blocking send.

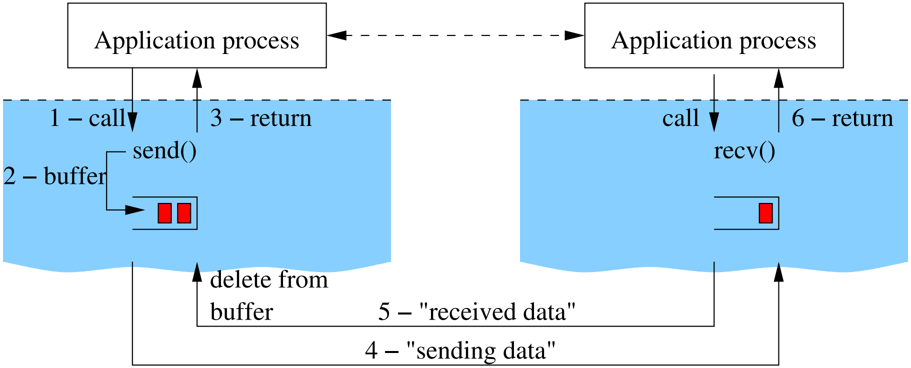

Figure: Local-blocking send.

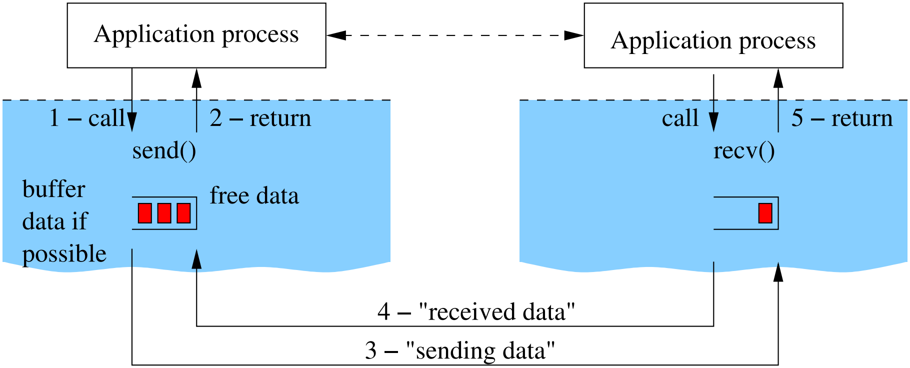

Figure: Non-blocking send.

The `recv()` is also either blocking or non-blocking. A blocking call will not return until a message has been received. A non-blocking call will return immediately with either a message (if one was buffered and waiting to be received) or no message.

There may be several messages waiting in the receive buffer to be received. The `recv()` function will usually receive one message per call.

Use of a non-blocking `recv()` function is similar to polling a device for I/O. It wastes CPU cycles if called too many times without a message being received.

The significant difference between blocking and both local-blocking and non-blocking is that the sender can be **sure** that the receiver has received the data when a blocking send is used. This is useful for loose synchronization.

Blocking and non-blocking connote *synchronous* and *asynchronous* communication.

The communication API needs to stipulate various axioms or quality of service about the message passing service provided. For example, if message A is sent and subsequently message B is sent to a receiver with non-blocking sends, is the receiver guaranteed to receive A before B?

Table: Communication patterns

communication	source	destination
unicast	one node	one node
broadcast	one node; one item	all nodes; item per node
gather	all nodes; item per node	one node; N items
scatter	one node; N items	all nodes; item per node
gather/broadcast	all nodes; item per node	all nodes; N items per node
gather/scatter	all nodes; N items per node	all nodes; N items per node
reduce	all nodes; arithmetic combining	one node; one item
reduce/broadcast	all nodes; arithmetic combining	all nodes; same item in each
prefix	all nodes; arithmetic combining	all nodes; different item in each

The unicast sends a single data item from a single node to a single node.

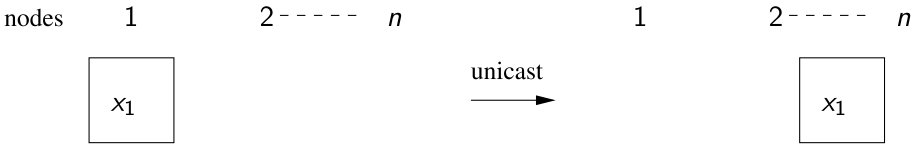

Figure: Unicast.

The broadcast sends a single data item from a single node to all nodes.

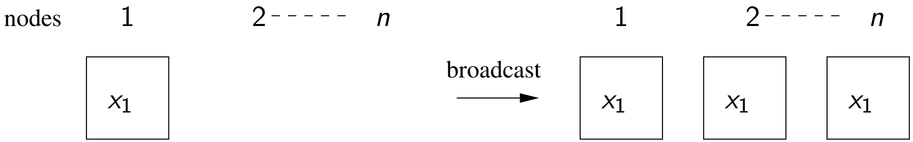

Figure: Broadcast.

The gather sends a single data item from each node to the same, single node.

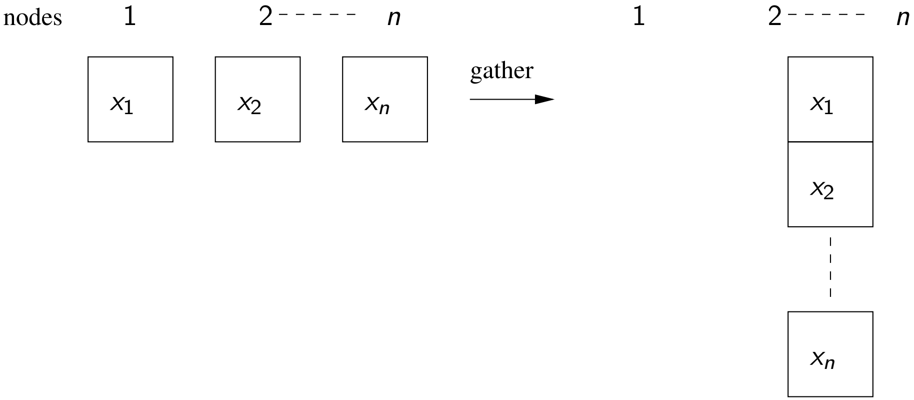

Figure: Gather.

The scatter sends each element from an array of data to a different node.

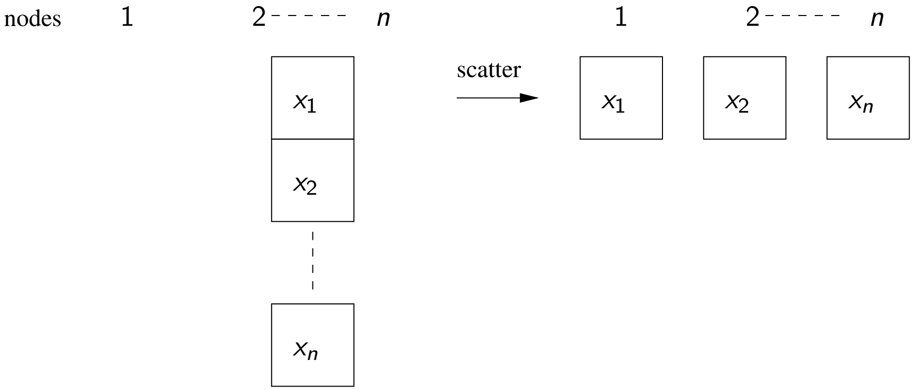

Figure: Scatter.

The gather/broadcast sends a data item from each node to every node.

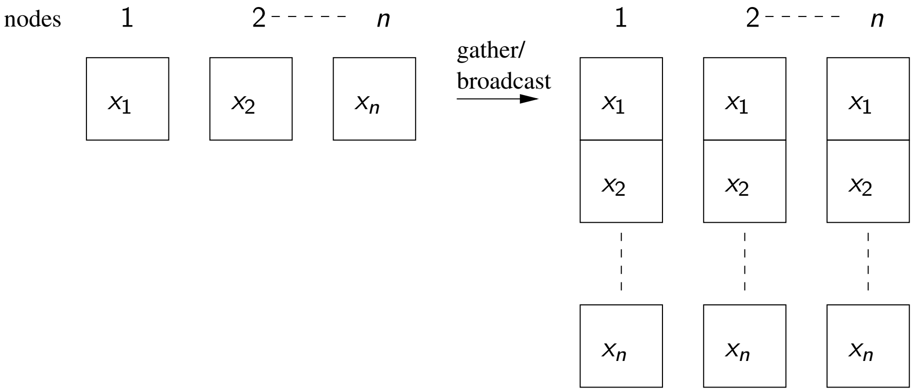

Figure: Gather/broadcast.

The gather/scatter sends each element from arrays at every node to a different node.

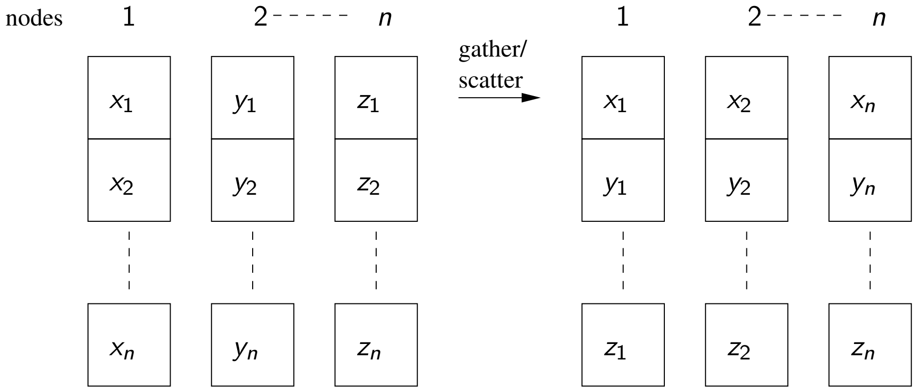

Figure: Scatter/gather.

The reduce sends the result of a mathematical function over the data items from each node to a single node.

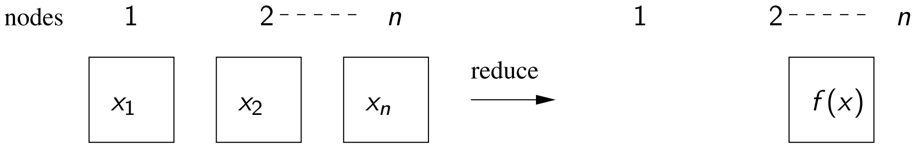

Figure: Reduce.

The reduce sends the result of a mathematical function over the data items from each node to all nodes.

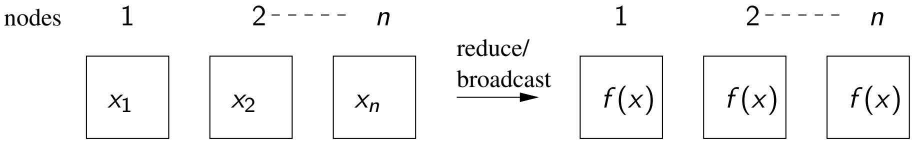

Figure: Reduce/broadcast.

The prefix sends the result of a different mathematical function over the data items to each of the nodes.

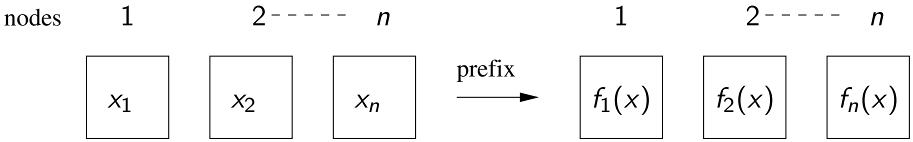

Figure: Prefix.