Three parallel-programming models

- *Shared-memory* programming is like using a “bulletin board” where you can communicate with colleagues.
- *Message-passing* is like communicating via e-mail or telephone calls. There is a well defined event when a message is sent or received.
- *Data-parallel* programming is a “regimented” form of cooperation. Many processors perform an action separately on different sets of data, then exchange information globally before continuing en masse.

User-level communication primitives are provided to realize the programming model

- There is a mapping between language primitives of the programming model and these primitives

These primitives are supported directly by hardware, or via OS, or via user software.

In the early days, the kind of programming model that could be used was closely tied to the architecture.

Today—

- Compilers and software play important roles as bridges
- Technology trends exert a strong influence

The result is convergence in organizational structure, and relatively simple, general-purpose communication primitives.

A shared address space

In the shared-memory model, processes can access the same memory locations.

Communication occurs implicitly as result of loads and stores

This is convenient.

- Wide range of granularities supported.
- Similar programming model to time-sharing on uniprocessors, except that processes run on different processors
- Wide range of scale: few to hundreds of processors

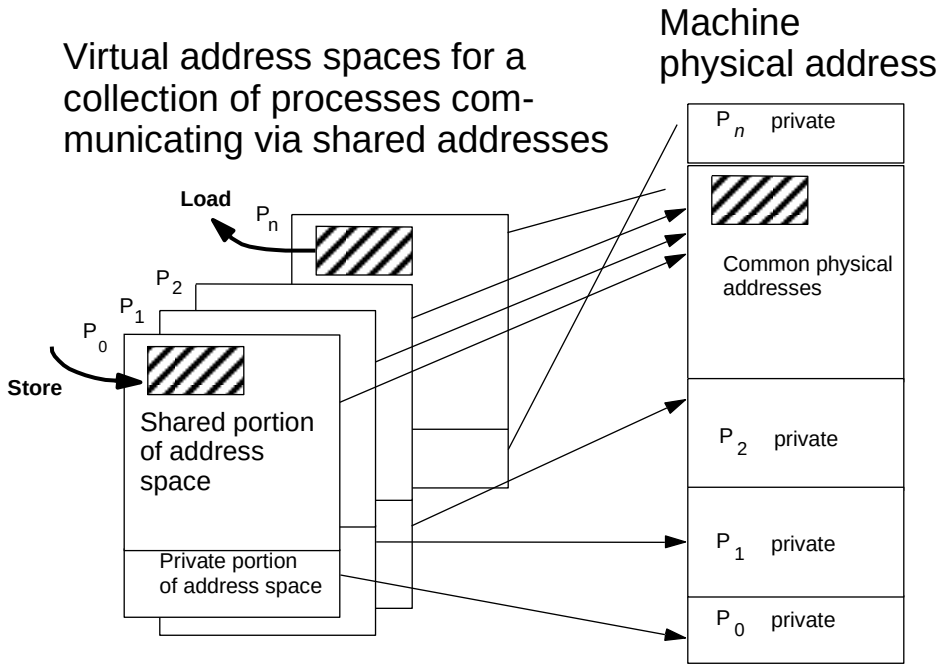
Good throughput on multiprogrammed workloads.

This is popularly known as the *shared memory* model, even though memory may be physically distributed among processors.

The shared-memory model

A process is a virtual address space plus

Portions of the address spaces of tasks are shared.



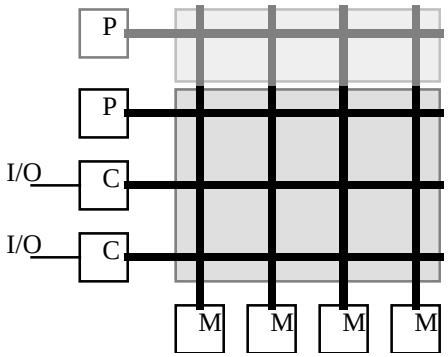
What does the private region of the virtual address space usually contain?

Conventional memory operations can be used for communication.

Special atomic operations are used for synchronization.

The interconnection structure

The interconnect in a shared-memory multiprocessor can take several forms.



It may be a *crossbar switch*.

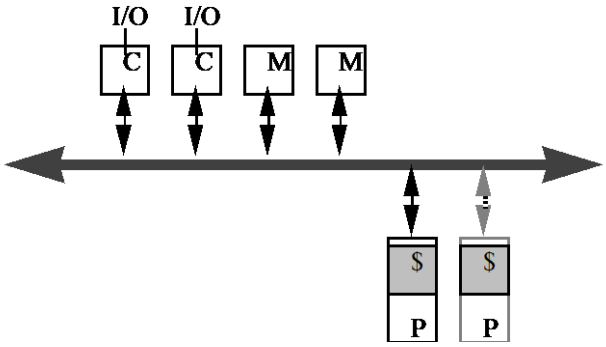
Each processor has a direct connection to each memory and I/O controller.

Bandwidth scales with the number of processors.

Unfortunately, cost scales with

This is sometimes called the “mainframe approach.”

At the other end of the spectrum is a *shared-bus* architecture.



All processors, memories, and I/O controllers are connected to the bus.

Such a multiprocessor is called a symmetric multiprocessor (SMP).

What are some advantages and disadvantages of organizing a multiprocessor this way? List them here.

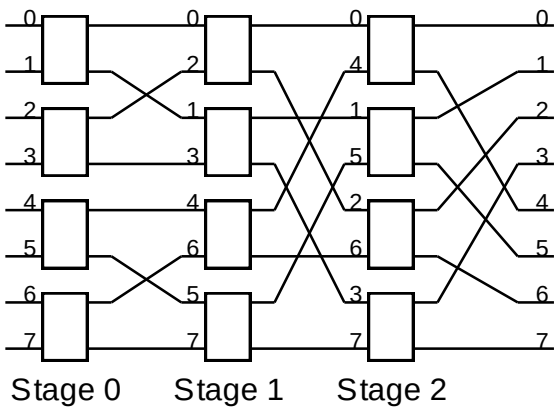
-
-
-

A compromise between these two organizations is a *multistage interconnection network.*

The processors are on one side, and the memories and controllers are on the other.

Each memory reference has to traverse the stages of the network.

Why is this called a compromise between the other two strategies?



For small configurations, however, a shared bus is quite viable.

Message passing

In a message-passing architecture, a complete computer, including the I/O, is used as a building block.

Communication is via explicit I/O operations, instead of loads and stores.

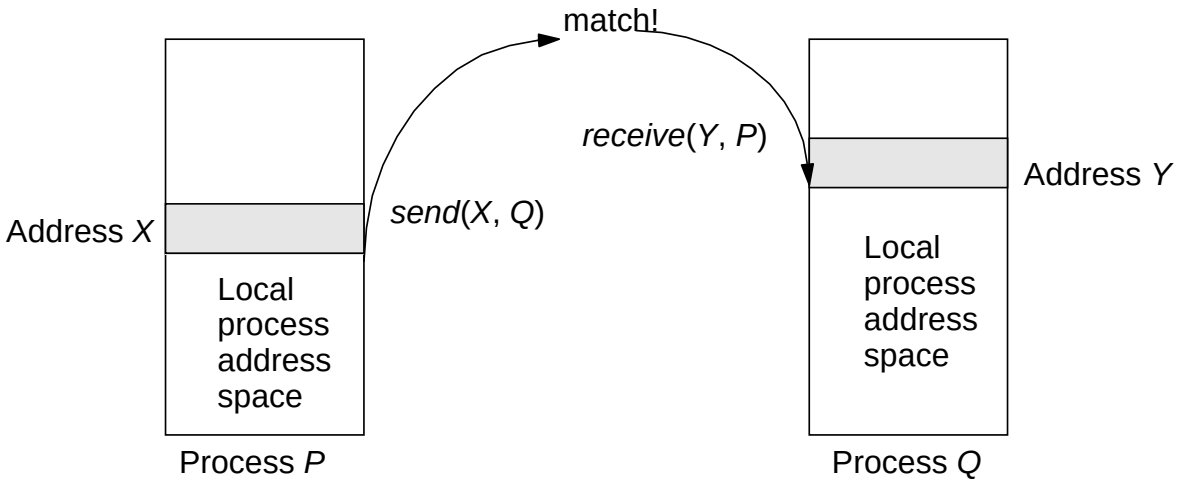
- A program can directly access only its private address space (in local memory).
- It communicates via explicit messages (*send* and *receive*).

It is like a network of workstations (clusters), but more tightly integrated.

Easier to build than a scalable shared-memory machine.

Send-receive primitives

The programming model is further removed from basic hardware operations.



Library or OS intervention is required to do communication.

- *send* specifies a buffer to be transmitted, and the receiving process.
- *receive* specifies sending process, and a storage area to receive into.
- A memory-to-memory copy is performed, from the address space of one process to the address space of the other.
- There are several possible variants, including whether *send* completes—

  when the *receive* has been executed,

  when the send buffer is available for reuse, or

  when the message has been sent.
- Similarly, a *receive* can wait for a matching *send* to execute, or simply fail if one has not occurred.

There are many overheads: copying, buffer management, protection. Let's describe each of these. Submit your descriptions here.

- Why is there an overhead to copying, compared to a share-memory machine?

- Describe the overhead of buffer management.

- What is the overhead for protection?

Here's an example from the textbook of the difference between shared address-space and message-passing programming.

A shared-memory system uses the _____ model:

```
int a, b, signal;
…
void dosum(<args>) {
  while (signal == 0) {}; // wait until instructed to work
  printf("child thread> sum is %d", a + b);
  signal = 0;  // my work is done
}

void main() {
  signal = 0;
  thread_create(&dosum);  // spawn child thread
  a = 5, b = 3;
  signal = 1;             // tell child to work
  while (signal == 1) {}  // wait until child done
  printf("all done, exiting\n");
}
```

Message-passing uses the _____ model:

```
int a, b;
…
void dosum() {
  recvMsg(mainID, &a, &b);
  printf("child process> sum is %d", a + b);
}

void main() {
  if (fork() == 0)  // I am the child process
    dosum();
  else {            // I am the parent process
    a = 5, b = 3;
    sendMsg(childID, a, b);
```

```
    wait(childID);
    printf("all done, exiting\n");
  }
}
```

Here's the relevant section of documentation on the **`fork()`** function: "Upon successful completion, **`fork()`** and **`fork1()`** return 0 to the child process and return the process ID of the child process to the parent process."

Interconnection topologies

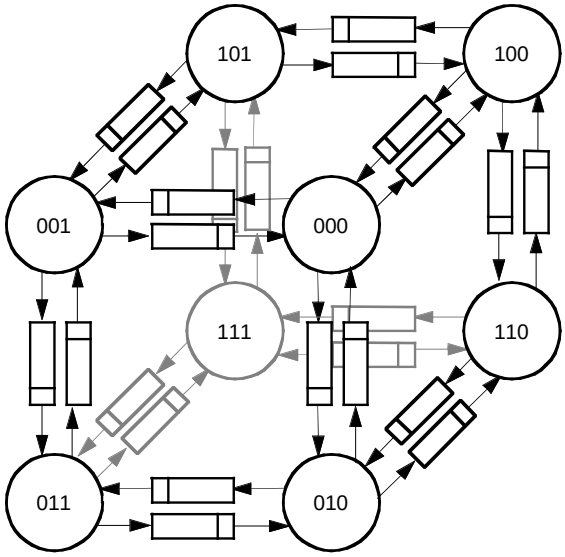
Early message-passing designs provided hardware primitives that were very close to the message-passing model.

Each node was connected to a fixed set of neighbors in a regular pattern by point-to-point links that behaved as FIFOs.

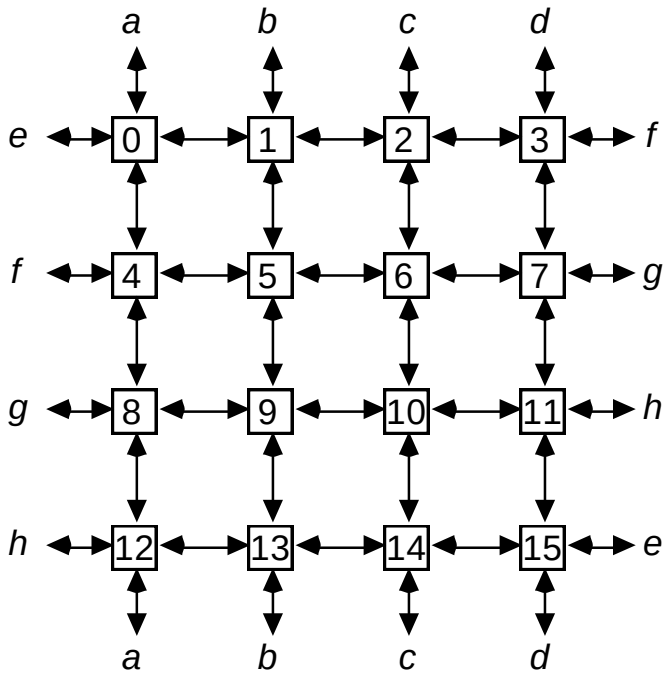
A common design was a *hypercube*, which had $2 \times n$ links per node, where n was the number of dimensions.

The diagram shows a 3D cube.

One problem with hypercubes was that they were difficult to lay out on silicon.



Because of that, 2D meshes eventually supplanted hypercubes.



Here is an example of a 16-node mesh. Note that the last element in one row is connected to the first element in the next.

If the last element in each row were connected to the first element in the same row, we would have a *torus* instead.

Early message-passing machines used a FIFO on each link.

- Thus, software sends were implemented as synchronous hardware operations at each node.

  What was the problem with this, for multi-hop messages?

- Synchronous ops were replaced by DMA, enabling non-blocking operations
  - A DMA device is a special-purpose controller that transfers data between memory and an I/O device without processor intervention.
  - Messages were buffered by the message layer of the system at the destination until a *receive* took place.
  - When a *receive* took place, the data was

The diminishing role of topology.

- With store-and-forward routing, topology was important.

Parallel algorithms were often changed to conform to the topology of the machine on which they would be run.

- Introduction of pipelined ("wormhole") routing made topology less important.

In current machines, it makes less difference how far the data travels.

This simplifies programming; cost of interprocessor communication is essentially independent of which processor is receiving the data.

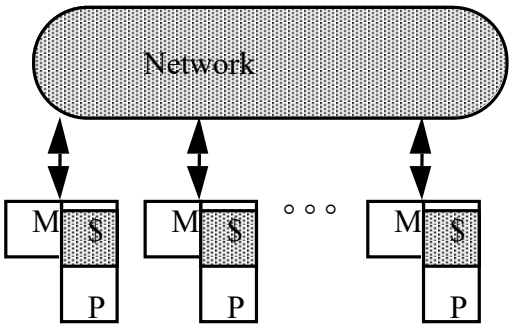
Toward architectural convergence

In 1990, there was a clear distinction between message-passing and shared-memory machines. Today, there isn't a distinct boundary.

- Message-passing operations are supported on most shared-memory machines.
- A shared virtual address space can be constructed on a message-passing machine, by sharing *pages* between processors.
  - When a missing page is accessed, a page fault occurs.
  - The OS fetches the page from the remote node via message-passing.

At the machine-organization level, the designs have converged too.

The block diagrams for shared-memory and message-passing machines look essentially like this:



In shared memory, the network interface is integrated with the memory controller.

It initiates a transaction to access memory at a remote node.

In message-passing, the network interface is essentially an I/O device.

What does Solihin say about the ease of writing shared-memory and message-passing programs on these architectures?

- Which model is easier to program for initially?
- Why doesn't it make much difference in the long run?