Three parallel-programming models

- *Shared-address* 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 address-space 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.

But this term is ambiguous. Why?

*The shared address-space model*

A process is a virtual address space plus

Portions of the address spaces of tasks are shared.

Virtual address spaces for a collection of processes communicating via shared addresses

Machine physical address

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

CS&G call this 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).

- Latency larger than for uniprocessor
- Low incremental cost
- Bus is bandwidth bottleneck

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 SAS 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.
- Optionally, a **send** may include a **tag**.
  - In this case, the **receive** may also specify a matching rule, e.g.,
    - "match only a specific tag from a specific processor," or
    - "match any tag from any processor."
- 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.

- Why is there an overhead to copying, compared to an SAS machine?

- Describe the overhead of buffer management.

- What is the overhead for protection?

**Interconnection topologies**

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

So 2D meshes were also used.
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, the hardware was close to the programming model; send and receive were synchronous operations.

What was the problem with this?

- This was 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 little difference in time 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, that distinction is less clear.

- 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:

![Diagram](image)

In shared memory, the network
interface is integrated with the
memory controller, to conduct a
transaction to access memory at a
remote node.

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

DMA transfers across the network.
Similarly, some switch-based LANs provide scalable interconnects that approach what loosely coupled multiprocessors offer.

**Data-parallel processing**

**Programming model**

- Operations performed in parallel on each element of a data structure
- Logically single thread of control, performs sequential or parallel steps
- Conceptually, a processor is associated with each data element.

**Architectural model**

- Array of many simple, cheap processors (“processing elements”—PEs) with little memory each
- Processors don’t sequence through instructions
  - Attached to a control processor that issues instructions
- Specialized and general communication, cheap global synchronization.

Here is a block diagram of an array processor.

The original motivations were to

- match the structure of simple differential equation solvers
- centralize high cost of instruction fetch/sequencing

Each instruction is either an operation on local data elements, or a communication operation.

For example, to average each element of a matrix with its four neighbors,
• a copy of the matrix would be shifted across the PEs in each of the four directions, and
• a local accumulation would be performed in each PE.

The control processor is actually an ALU that controls the other processors.

• Executes conditional branch instructions.
• Loads and stores index registers. (Some or all of the index registers can also be used by the PEs.)
• *Broadcasts* other instructions to the PEs.

Each PE has access to only one memory unit.

• Memory is effectively *interleaved* so that each processor can reference memory at once.
• But each processor can reference only specific memory addresses—e.g. addresses ending in 0101.
• Special instructions are provided to move data between the memories of different processors.

Here is how this kind of memory can be used to hold three arrays:
**Matrix multiplication—a sample program**

Suppose we want to calculate the product of two $N \times N$ matrices, $C := A \times B$. We must perform this calculation:

$$c_{ij} := \sum_{k=0}^{N-1} a_{ik} b_{kj}$$

Each of the $N$ processors operates on a single element of the array.

Operations can be carried out for all indices $k$ in the interval $[0, N-1]$ simultaneously.

*Notation:* $(0 \leq k \leq N-1)$
\textbf{for } i := 0 \textbf{ to } N-1 \textbf{ do} \\
\textbf{begin} \\
\{Initialize the elements of the \}
\text{ith row to zero (in parallel)}\}
C[i, k] := 0, \(0 \leq k \leq N-1); \\
\textbf{for } j := 0 \textbf{ to } N-1 \textbf{ do} \\
\{Add the next term to each of } N \text{ sums} \}
C[i, k] := C[i, k] + A[i, j] \times B[j, k], \(0 \leq k \leq N-1); \\
\textbf{end}; \\
\textbullet \quad \text{The program simultaneously computes all the elements in an entire row of the result matrix.} \\
\textbullet \quad \text{Only } N^2 \text{ array multiplications are required by the algorithm, compared with } N^3 \text{ scalar multiplications in the corresponding serial algorithm.} \\
\textbullet \quad \text{An instruction is needed to simultaneously multiply each element of the } j \text{th row of } B \text{ by } a_{ij}, \text{ in other words, to multiply a vector by a scalar. Thus it is desirable to broadcast a single element simultaneously to all processors.} \\

\textit{The instruction set of an array processor}

Our example array processor has a parallel one-address (single accumulator) instruction set.

\textbullet \quad \text{In an ordinary one-address instruction set, a LOAD causes the accumulator to be loaded.} \\
\textbullet \quad \text{In an array computer, a LOAD causes all the accumulators in the computer (usually 64 or 256) to be loaded.} \\
\textbullet \quad \text{Similarly, an ADD, etc., causes all accumulators to perform an addition. (Actually, as we will see later, there is a way to stop some of the accumulators from performing the operation.)} \\

Here are the formats of some (array) arithmetic and indexing instructions:
Vector instructions

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector load</td>
<td>ACC[k] := A[0, k], (0 ≤ k ≤ N–1)</td>
</tr>
<tr>
<td>Vector load (indexed)</td>
<td>ACC[k] := A[index[i], k], (0 ≤ k ≤ N–1)</td>
</tr>
<tr>
<td>Vector store</td>
<td>A[0, k] := ACC[k], (0 ≤ k ≤ N–1)</td>
</tr>
<tr>
<td>Vector add</td>
<td>ACC[k] := ACC[k] + A[0, k], (0 ≤ k ≤ N–1)</td>
</tr>
<tr>
<td>Vector multiply</td>
<td>ACC[k] := ACC[k] × A[0, k], (0 ≤ k ≤ N–1)</td>
</tr>
<tr>
<td>Broadcast scalar</td>
<td>ACC[k] := ACC[index[i]], (0 ≤ k ≤ N–1)</td>
</tr>
</tbody>
</table>

Indexing instructions

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter index constant</td>
<td>INDEX[i] := 1</td>
</tr>
<tr>
<td>Load index</td>
<td>MEMORY[Y]</td>
</tr>
<tr>
<td>Increment index constant</td>
<td>INDEX[i] := INDEX[i] + 1</td>
</tr>
<tr>
<td>Compare index, branch if low</td>
<td>if INDEX[i] &lt; INDEX[j] then goto LABEL</td>
</tr>
</tbody>
</table>

The BCAST (broadcast) instruction copies a value from one accumulator into all the other accumulators. Most algorithms that use this instruction would be very inefficient on a uniprocessor!

Note that this is not an exhaustive list of instructions. There would undoubtedly be a SUB, a DIV, and indexed forms of all the other array instructions.

Sample assembly-language program

We will code the inner loop from our matrix-multiplication program.

    for j := 0 to N–1 do
        (Add the next term to each of N sums)
        \[ C[i, k] := C[i, k] + A[i, j] \times B[j, k], (0 \leq k \leq N–1); \]

First, we need to initialize for the loop, by placing the loop index \( j \) and the loop limit \( N \) in registers. (Note that in this instruction set, we can only compare quantities that are in registers.)
1. ENXC \( j, 0 \) Loop counter into reg. \( j \)
2. ENXC \( \text{lim}, N \) Loop limit into reg. “lim”

(In a real program, the index registers we are calling \( j \) and “lim” would actually have numbers—registers 1 and 2, for example.)

Since each of the elements of the \( j \)th row of \( B \) need to be multiplied by \( A[i, j] \), we must arrange to put \( A[i, j] \) into all the accumulators.

3. JLOOP LOAD \( A[i] \) Load row number pointed to by index register \( i \).
4. BCAST \( j \) Broadcast from \( j \)th accumulator to all other accumulators.

Next we multiply by the \( j \)th row of \( B \) and add the products to the elements of the \( i \)th row of \( C \).

5. MUL \( B[j] \) Multiply \( A[i, j] \times B[j, k] \)
6. ADD \( C[i] \) Add \( C[i, k] \).
7. STO \( C[i] \) Save sums in memory for use in next iteration.

Finally, we need to take care of the loop control.

8. ICNX \( j, 1 \) Increment register \( j \).
9. CPNX \( j, \text{lim}, \text{JLOOP} \) If \( j < \text{lim} \), then \text{goto} \text{JLOOP}

Masking for conditional branching

This type of programming works great as long as the same instructions need to be performed on all items in the accumulator, but suppose that we want to operate on only some elements?

There’s only one instruction stream, so we can’t take a conditional branch for the elements in some accumulators and not branch for other accumulators.

However, we can design hardware to \textit{deactivate} some of the processors.
**Strategy:** Always execute both branches of the if. Deactivate, or mask out, some of the processors during the then portion, and deactivate the rest during the else portion. Here are the steps.

1. Mask out the processors whose accumulator is zero (or, whose accumulator is positive, negative, etc.).
2. Perform the then clause.
3. Mask out the processors whose accumulator is not zero (positive, negative, etc.).
4. Perform the else clause.
5. Turn on all processors.

Let us assume that a special register, called MASK, can be loaded from any of the index registers (or vice versa).

Comparisons (<, ≤, =, ≠, >, ≥) leave their results in an index register, from which the MASK register can be loaded. (This assumes # of processors is ≤ # of bits in an index register.)

Branches can depend upon—

- whether all bits in some index register are 1.
- whether any bits in some index register are 1.
- whether no bits in some index register are 1 (i.e., all bits are 0).

Below is the set of instructions for manipulating index registers and the mask.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector compare less</td>
<td>CLSS A, i</td>
<td>if ACC[k] &lt; A[k] then kth bit of INDEX[i] is set to 1, otherwise reset to 0.</td>
</tr>
<tr>
<td>Vector compare equal</td>
<td>CEQL A, i</td>
<td>if ACC[k] = A[k] then kth bit of INDEX[i] is set to 1, otherwise reset to 0.</td>
</tr>
<tr>
<td>Branch all</td>
<td>BRALL i, LOOP</td>
<td>Branch to LOOP if INDEX[i] has all 1 bits.</td>
</tr>
<tr>
<td>Branch any</td>
<td>BRANY (i, j), LOOP</td>
<td>Branch to LOOP if INDEX[(i)] has any 1 bits.</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>Branch none</td>
<td>BRNON (i, j), LOOP</td>
<td>Branch to LOOP if INDEX[(i)] has all 0 bits.</td>
</tr>
<tr>
<td>Logical and of index</td>
<td>AND (i, j)</td>
<td>INDEX[(i)] := INDEX[(i)] and INDEX[(j)]</td>
</tr>
<tr>
<td>Logical or of index</td>
<td>OR (i, j)</td>
<td>INDEX[(i)] := INDEX[(i)] or INDEX[(j)]</td>
</tr>
<tr>
<td>Complement index</td>
<td>CMP (i)</td>
<td>INDEX[(i)] := not INDEX[(i)]</td>
</tr>
<tr>
<td>Load mask from index</td>
<td>LDMSK (i)</td>
<td>MASK := INDEX[(i)]</td>
</tr>
<tr>
<td>Store mask in index</td>
<td>STMSK (i)</td>
<td>INDEX[(i)] := MASK</td>
</tr>
</tbody>
</table>

**Sample program:** Array “normalization”—replace \(A[i, j]\) by \(A[i, j] / A[0, j]\) provided \(A[0, j] \neq 0\):

\[
\text{for } i := N-1 \text{ to } 0 \text{ do}
\]
\[
\text{if } A[0, j] \neq 0 \text{ then } A[i, j] := A[i, j] / A[0, j], \quad (0 \leq j \leq N-1);
\]

We can compute which elements of \(A[0, j]\) are zero before entering the loop. The program now reads—

\[
\text{var } M[0 \ldots N-1]: \text{boolean}; \quad \{ M \text{ is the mask register. } \}
\]
\[
\text{begin}
\]
\[
M[k] := A[0, k] \neq 0, \quad (0 \leq k \leq N-1);
\]
\[
\text{for } i := N-1 \text{ to } 0 \text{ do } A[i, j] := A[i, j] / A[0, j], \quad (M[j]);
\]
\[
\text{end};
\]

First, we set up the mask:

1. SUB ACC \( ACC[k] := ACC[k] - ACC[k], \quad 0 \leq k \leq N-1 \). Clear all ACC regs.
2. CEQL A, M \( \text{Set } k \text{th bit of INDEX[M] to } \text{ACC}[k] = A[0, k]. \text{ This sets bits of INDEX[M] corresponding to } 0 \text{ elements of A.} \)
3. CMP M \( \text{INDEX[M]} := \text{not INDEX[M]. Sets bits of INDEX[M] corresponding to non-zero elements of A.} \)
4. LDMSK M Load mask reg. from INDEX[M].

Now we initialize for the loop:

5. ENXC i, 1 INDEX[i] := 1
6. LDNX lim, N Loop limit into reg. “lim”

Here comes the loop! The LOAD, DIV, and STO operations are only performed for those processors whose mask bits are 1.

10. ICNX i, 1 INDEX[i] := INDEX[i] + 1
11. CPNX i, lim, ILOOP Branch to ILOOP if not yet finished.

Vector processors

Array processors were eclipsed in the mid-'70s by the development of vector processors.

As we mentioned in Lecture 1, vector processors use a pipeline to operate on different data elements concurrently.

This eliminates the need for the array elements to be in specific places in memory.

The first vector machine was the CDC Star-100. It was a memory-memory architecture.

The Cray-1 (1976) was a load-store architecture with vector registers.

In the mid-80s, array processors made a comeback with the Connection Machine (-1 & -2), made up of thousands of bit-serial PEs.

But its advantage lasted only till the development of single-chip floating-point processors integrated with caches.
The SIMD approach evolved into the SPMD approach (single program, multiple data). All processors execute copies of the same program.