Other parallel architectures

Dataflow architectures:
This organization is fundamentally different than all we have seen before, because it is data driven rather than control driven.

- An instruction is ready for execution whenever its operands have arrived (i.e., have been evaluated).
- There is no concept of control flow, thus no PC.
- One way of thinking about dataflow computers: Each operand specifies an instruction (and potentially another operand) rather than vice versa.
- For effective programming, requires very different programming languages—called functional programming.

A dataflow program graph is made up of “actors” (operators) connected by arcs. Each input or output arc can carry a “token” (value).

An actor is enabled (ready for execution) whenever tokens are present on each of its input arcs, but not on any of its output arcs.

An instruction in a dataflow computer is an activity template consisting of—

- the name of the operation
- “receivers”: places waiting to be filled in with operand values
- destinations, which are (activity template, operand #) ordered pairs.

This dataflow graph
At the left is a diagram of a dataflow machine.

Note that an instruction, along with its data, is directed to a free processor by the arbitration network.

The distribution network takes care of passing the results to the instructions where they are needed.
Here is how matching occurs.

- A message, or *token*, from the network consists of data and an address, or *tag*.
- When an instruction completes, its tag is compared against those in the token store.
- If a matching token (the other operand) is present, the matching token is extracted and the instruction is issued for execution.
- The instruction executes, and computes a result.
- The result is sent to each of the destinations specified in the instruction. (This happens whether the successor instructions are local or on a remote processor.)

**Systolic architectures**

A systolic architecture replaces a single processor with an array of regular processing elements.

The idea is to orchestrate data flow for high throughput with less memory access.

Systolic arrays are different from pipelining.
• Nonlinear array structure,
• multidirectional data flow,
• each PE may have (small) local instruction and data memory

Systolic arrays are different from SIMD: Each PE may do something different.

Initial motivation: VLSI enables inexpensive special-purpose chips.

They represent algorithms directly by chips connected in regular pattern.

Here is a systolic array for computing a 1D convolution.

\[
y(i) = w_1 \times x(i) + w_2 \times x(i + 1) + w_3 \times x(i + 2) + w_4 \times x(i + 3)
\]

Each box in the above diagram represents a computational unit performing a specific function.

Each time the clock beats, all units accept inputs, compute results, and generate outputs.

Practical architectures like iWarp provide many functions so that many algorithms can be implemented.
A generic parallel architecture

A node consists of

- processor(s),
- memory system,
- plus *communication assist*, a network interface and communication controller.

The network is scalable.

Convergence allows lots of innovation, now within this framework.

**Fundamental design issues**

At any layer, there is an interface (contract between hardware & software) aspect and performance aspects

- *Naming*: How are logically shared data and/or processes referenced?
- *Operations*: What operations are provided on these data
- *Ordering*: How are accesses to data ordered and coordinated?
• *Replication:* How are data replicated to reduce communication?

• *Communication Cost:* Latency, bandwidth, overhead, occupancy

We need to understand the programming model, since that sets requirements for how the hardware and software work together.

**Programming-model requirements**

*Sequential programming model*

Contract

• *Naming:* Can name any location in virtual address space.

  Hardware (and perhaps compilers) does translation to physical addresses.

• *Operations:* Loads and stores.

• *Ordering:* Sequential program order.

Performance

• Optimizations rely on dependencies on single location (mostly): compiler and hardware preserve the *dependence order.*

• Compiler: reordering and register allocation

• Hardware: out of order, write buffers

The illusion of program order is preserved while actually executing the program in a *weaker* dependence order.
SAS programming model

- **Naming**: Any process can name any variable in shared space.
- **Operations**: loads and stores, plus synchronization operations needed for ordering.
- **Simplest Ordering Model**:
  - Within a process/thread: sequential program order
  - Across threads: some interleaving (as in time-sharing)
  - Additional ordering through synchronization
  - Again, compilers/hardware can violate orders without getting caught
- Different, more subtle ordering models also possible (discussed later).

**Synchronization**

- Mutual exclusion (locks)
  
  Ensure certain operations on certain data can be performed by only one process at a time.
  
  Like a room that only one person can enter at a time.
  
  No ordering guarantees.
- Event synchronization

  Ordering of events to preserve dependencies, e.g. producer —> consumer of data

  3 main types:
  
  - point-to-point (involving two processes)
  - global (involving all processes)
  - group (involving a subset of processes)

Message-passing programming model

- **Naming**: Processes can name private data directly.

  No shared address space
• **Operations**: Explicit communication through *send* and *receive*.
  - Send transfers data from private address space to another process
  - Receive copies data from process to private address space
  - Must be able to name processes

• **Ordering**:
  - Program order within a process
  - Send and receive can provide point-to-point synchronization between processes
  - Mutual exclusion inherent. Why?

• Can construct global address space:
  - Process number + address within process address space.
  - But no direct operations on these names.

**Replication**
Very important for reducing data transfer and communication.

• In a *uniprocessor*, caches do it automatically. This reduces communication
  If the processor uses the data while it’s in the cache, further

  When data is written, something must be done to assure that later reads get the new value.

• In the *message-passing* model, a receive replicates, giving the data a new name. In contrast to uniprocessor caching, this replication is done explicitly.

• In the *shared-address-space* model, replication occurs on several levels.
• A load brings in data transparently, so replication is transparent. What does that mean?

• Hardware caches also replicate.
• The OS can replicate, by copying pages or objects.

Since there is no explicit renaming, many copies of the data may have the same name.
This introduces a coherence problem.

**Communication Performance**

The performance of various communication operations influence their use.

Programmers and compilers make choices based on performance.

Fundamentally, there are three important characteristics:

- *Latency*, the time taken for an operation.
- *Bandwidth*, the rate of performing operations.
- *Cost*, the impact on the execution time of the program.

If processor does one thing at a time,

\[
\text{bandwidth} \propto 1 \div \text{latency}, \text{ and } \text{cost is latency} \times \# \text{ of operations performed.}
\]

However, the situation is more complex in modern computer systems, because they do more things at once.

We'll focus on communication or data transfer across nodes.

*Simple example:*

Suppose a component can perform an operation in 100 ns.
Its simple bandwidth is 10
Now, suppose it is pipelined with a depth of 10. What is its peak bandwidth?
The rate is determined by the slowest stage of the pipeline, not by the overall latency.

The delivered bandwidth on application depends on the initiation frequency. To see this, …

Suppose application performs 100M of these operations. What is the cost?

- op count \times op latency gives 10 sec (upper bound)
- op count / peak op rate gives 1 sec (lower bound)

This assumes full overlap of latency with useful work. In this case, the cost is just the *issue* cost.

If an application can do 50 ns. of useful work before depending on the result of the operation, the cost to the application is the other 50 ns. of latency.

*A linear model of data-transfer latency*

\[
\text{Transfer time}(n) = T_0 + \left( n \div B \right)
\]

where

- \( T_0 \) is the
- \( n \) is the
- \( B \) is the transfer rate of the component moving the data

This model is useful for message-passing, memory access, vector ops etc.

As \( n \) increases, bandwidth approaches asymptotic rate \( B \).

How quickly it approaches depends on \( T_0 \).

It is easily shown that the data size at which half the peak bandwidth is obtained (the half-power point) is

\[
n_{1/2} = T_0 \div B
\]
But a linear model not enough

- When can the next transfer be initiated?
- Can the cost be overlapped?

To determine this, we need to know how the transfer is performed.

*A model for communication cost*

Communication time per message =

\[
\text{Overhead} + \text{Occupancy} + \text{Network delay} + \text{Size/Bandwidth} + \text{Contention}
\]

\[= o_v + o_c + l + n/B + T_c\]

Each component along the way has occupancy and delay

- Overall delay is sum of delays.
- Overall occupancy \((1/\text{Bandwidth})\) is biggest of occupancies.

This is a very general model for data transfer. It applies to cache misses too.

**Summary of Design Issues**

Functional and performance issues apply at all layers.

Functional issues are naming, operations and ordering.

Performance issues are organization, latency, bandwidth, overhead, and occupancy.

Replication and communication are deeply related.

Goal of architects: design against frequency and type of operations that occur at communication abstraction, constrained by tradeoffs from levels above or below.